Type Inference and Optimisation for an Impure World.

Ben Lippmeier

May, 2010

A thesis submitted for the degree of Doctor of Philosophy
of the Australian National University

Declaration

The work in this thesis is my own except where otherwise stated.

Ben Lippmeier

Thanks

Thanks to my family and friends for putting up with me and my occasional disappearances.

Sincere thanks to my supervisor Clem Baker-Finch, to Simon Marlow for an internship at GHC HQ, and to Bernie Pope for reading draft versions of this thesis.

Shouts out to the FP-Syd crew, for pleasant monthly diversions when I probably should have been writing up instead.

Thanks to the General, the Particular and FRATER PERDURABO, who taught me that great truths are also great lies.

Thanks to the people who receive no thanks.

All Hail Discordia.

Abstract

We address the problem of reasoning about the behaviour of functional programs that use destructive update and other side effecting actions. All general purpose languages must support such actions, but adding them in an undisciplined manner destroys the nice algebraic properties that compiler writers depend on to transform code.

We present a type based mutability, effect and data sharing analysis for reasoning about such programs. Our analysis is based on a combination of Talpin and Jouvelot's type and effect discipline, and Leroy's closure typing system. The extra information is added to our System-F style core language and we use it to guide code transformation optimisations similar to those implemented in GHC. We use a type classing mechanism to express constraints on regions, effects and closures and show how these constraints can be represented in the same way as the type equality constraints of System-Fc.

We use type directed projections to express records and to eliminate need for ML style mutable references. Our type inference algorithm extracts type constraints from the desugared source program, then solves them by building a type graph. This multi-staged approach is similar to that used by the Helium Haskell compiler, and the type graph helps to manage the effect and closure constraints of recursive functions.

Our language uses call-by-value evaluation by default, but allows the seamless integration of call-by-need. We use our analysis to detect when the combination of side effects and call-by-need evaluation would yield a result different from the call-by-value case. We contrast our approach with several other systems, and argue that it is more important for a compiler to be able to reason about the behaviour of a program, than for the language to be purely functional in a formal sense.

As opposed to using source level state monads, effect typing allows the programmer to offload the task of maintaining the intended sequence of effects onto the compiler. This helps to separate the conceptually orthogonal notions of value and effect, and reduces the need to refactor existing code when developing programs. We discuss the Disciplined Disciple Compiler (DDC), which implements our system, along with example programs and opportunities for future work.

Layout

The layout and formatting of this thesis was influenced by the work of Edward R. Tufte, especially his book "Visual Explanations". To reduce the need for the reader to flip pages back and forth when progressing through the material, I have avoided using floating figures, and have tried to keep related diagrams and discussion on the same two page spread. For this reason some diagrams have been repeated, and some sections contain extra white space.

Contents

Thanks **5**

Abstract **7**

Layout **9**

1 Introduction **17**

1.1 Prelude 17

1.2 The Problem 18

1.3 Why destructive update matters 20

1.3.1 Efficient data structures require destructive update 20

1.3.2 Destructive update helps to broadcast new values 25

1.4 What is purity? 30

1.5 Linear and uniqueness typing 38

1.6 State monads 40

1.7 Ref types invite large refactorisation exercises 47

1.8 Practical limitations of lazy evaluation 49

1.9 A way forward 55

2 Type System **57**

2.1 Update without imperative variables 58

2.2 Region Typing 60

2.2.1 Regions and aliasing 60

2.2.2 Region classes 62

2.2.3 Functions, allocation and non-material regions 63

2.2.4 Updating data requires it to be mutable 64

2.2.5 Primary regions and algebraic data 65

2.2.6 Thinking about regions 66

2.3 Effect typing 67

2.3.1 Effects and interference 67
2.3.2 Effect information in types 68
2.3.3 Effects and currying 70
2.3.4 Top level effects 71
2.3.5 Effects in higher order functions 72
2.3.6 Constraint strengthening and higher order functions . . . 72
2.3.7 Observable effects and masking 78
2.3.8 Recursive effects 80
2.3.9 Constant regions and effect purification 81
2.3.10 Purification in higher order functions 82
2.3.11 Strict, spine lazy and element lazy lists 84
2.3.12 Lazy and Direct regions 84
2.3.13 Liftedness is not a capability 86
2.4 The problem with polymorphic update 88
2.4.1 Fighting the value restriction 88
2.4.2 Don't generalise variables free in the store typing 89
2.4.3 Generalisation reduces data sharing in System-F 91
2.4.4 Restricting generalisation with effect typing 92
2.4.5 Observation criteria 93
2.4.6 Effect typing versus arbitrary update 94
2.5 Closure typing 96
2.5.1 Dangerous type variables 97
2.5.2 Closure typing and hidden communication 98
2.5.3 Material regions and sharing 101
2.5.4 Material regions and algebraic data types 102
2.5.5 Strong, mixed and absent region variables 104
2.5.6 Pure effects and empty closures 104
2.5.7 Closure trimming 105
2.6 Type classing 107
2.6.1 Copy and counting 107
2.6.2 Type classes for copy and update 108
2.6.3 Shape and partial application 110
2.6.4 Shape constraints and rigid type variables 111
2.6.5 Shape constraints and immaterial regions 113
2.7 Type directed projections 115

2.7.1 Default projections 116
2.7.2 Ambiguous projections and type signatures 117
2.7.3 Pull back projections 117
2.7.4 Custom projections 119
2.8 Comparisons with other work 121
2.8.1 FX. 1986 – 1993.
Gifford, Lucassen, Jouvelot and Talpin. 121
2.8.2 C++ 1986
Bjarne Stroustrup. 122
2.8.3 Haskell and unsafePerformIO. 1990
Simon Peyton Jones *et al.* 122
2.8.4 Behaviors and Trace Effects. 1993
Nielson and Nielson *et al* 123
2.8.5 λ_{var}. 1993 – 1994
Odersky, Rabin, Hudak, Chen 123
2.8.6 MLKit. 1994
Tofte, Talpin, Birkedal 124
2.8.7 Functional Encapsulation. 1995. Gupta 124
2.8.8 Objective Caml. 1996
Leroy, Doligez, Garrigue, Rémy and Jérôuillon. 124
2.8.9 Ownership Types. 1998
Clarke, Potter and Noble 125
2.8.10 Calculus of Capabilities and Cyclone. 1999
Crary, Walker, Grossman, Hicks, Jim, and Morrisett . . . 126
2.8.11 BitC. 2004
Shapiro, Sridhar, Smith, Doerrie 127
2.8.12 Monadic Regions. 2006
Fluet, Morrisett, Kiselyov, Shan 127

3 Type Inference **129**

3.1 Binding order and constraint based inference 130
3.2 Source language and constraint slurping 132
3.2.1 Normal types 135
3.2.2 Free variables of types 136
3.2.3 Dangerous variables 136
3.2.4 Material and immaterial variables 139
3.2.5 The *map* example 141
3.2.6 Annotated source language 142
3.2.7 Slurping and constraint trees 144

3.2.8 Types of programs and declarations . . . 145
3.2.9 Kinds of types and constraints . . . 146
3.2.10 Types of terms . . . 147
3.2.11 Example: type constraints . . . 154
3.2.12 Constraint sets and equivalence classes . . . 155
3.2.13 Example: type graph . . . 156
3.3 Constraint reduction . . . 158
3.3.1 Constraint entailment . . . 158
3.3.2 Unification . . . 159
3.3.3 Head read . . . 160
3.4 Generalisation . . . 162
3.4.1 Tracing . . . 162
3.4.2 Loop checking . . . 162
3.4.3 Packing . . . 163
3.4.4 Loop breaking . . . 164
3.4.5 Cleaning . . . 164
3.4.6 Quantification . . . 165
3.4.7 Late constraints and post-inference checking . . . 166
3.5 Projections . . . 168
3.5.1 Example: vectors . . . 170
3.5.2 Ambiguous projections and type signatures . . . 173
3.6 Constraint ordering and mutual recursion . . . 175
3.6.1 Comparison with Helium. 2002 Heeren, Hage, Swierstra. . . . 181
3.6.2 Comparison with other constraint systems. Pottier, Sulzmann, Odersky, Wehr *et al* . . . 182
3.7 Type Classes . . . 184
3.7.1 Deep Read/Write . . . 185
3.7.2 Deep Mutable/Const . . . 186
3.7.3 Purification . . . 186
3.7.4 Shape . . . 187
3.8 Error Reporting . . . 189
3.8.1 Constraint justifications . . . 190
3.8.2 Tracking purity errors . . . 193

4 Core Language **195**

4.1 Constraints and evidence . . . 196

4.1.1 Witness passing . . . 196

4.1.2 Dependent kinds . . . 197

4.1.3 Witnesses of mutability . . . 197

4.1.4 Witnesses of purity . . . 199

4.2 Simplified core language . . . 201

4.2.1 Symbol Classes . . . 202

4.2.2 Super-kinds . . . 203

4.2.3 Kinds . . . 203

4.2.4 Types . . . 203

4.2.5 Terms . . . 204

4.2.6 Weak values and lazy evaluation . . . 205

4.2.7 Stores, machine states and region handles . . . 206

4.2.8 Region allocation versus lazy evaluation . . . 207

4.2.9 Store typings . . . 208

4.2.10 Region similarity . . . 209

4.2.11 Duplication of region variables during evaluation . . . 210

4.2.12 Witness production . . . 211

4.2.13 Transitions . . . 212

4.2.14 Super-kinds of kinds . . . 214

4.2.15 Kinds of types . . . 215

4.2.16 Similarity . . . 217

4.2.17 Subsumption . . . 218

4.2.18 Types of terms . . . 219

4.2.19 Soundness of typing rules . . . 221

4.2.20 Goodness of typing rules . . . 222

4.3 Extensions to the simplified language . . . 224

4.3.1 Masking non-observable effects . . . 224

4.3.2 Masking effects on fresh regions . . . 225

4.3.3 Masking pure effects . . . 226

4.3.4 Bounded quantification . . . 228

4.3.5 Effect joining in value types . . . 231

4.4 Optimisations . . . 232

4.4.1 Local transforms . . . 232

4.4.2 Floating at the same level 233
4.4.3 Effects and region aliasing 235
4.4.4 Floating into alternatives 236
4.4.5 Floating outside lambda abstractions 237
4.5 Comparisons 240
4.5.1 Monadic intermediate languages. 1998
Tolmach, Benton, Kennedy, Russell. 240
4.5.2 System-Fc. 2007
Sulzmann, Chakravarty, Peyton Jones, Donnelly. 242

5 Conclusion **245**

5.1 Implementation 246
5.1.1 Implementing thunks and laziness 246
5.2 Limitations and possible improvements 247
5.2.1 Masking mutability constraints 247
5.2.2 Blocked regions and region sums 248
5.2.3 Bounded quantification and effect strengthening 248
5.2.4 Polymorphism, data sharing, and constraint masking . . . 249
5.2.5 Witnesses of no aliasing 251
5.2.6 Should we treat top-level effects as interfering? 252
5.2.7 Add witnesses to write effects 252
5.2.8 A better type for *force* 252
5.3 Summary of Contributions 254
5.4 The Hair Shirt 255

A Proofs of Language Properties **271**

Chapter 1

Introduction

1.1 Prelude

I am thinking of a data structure for managing collections of objects. It provides O(1) insert and update operations. It has native hardware support on all modern platforms. It has a long history of use. It's proven, and it's lightning fast.

Unfortunately, support for it in my favourite language, Haskell [PJ03a], appears to be somewhat lacking. There are people that would tell me that it's not needed [PJW92], that there are other options [Oka98b], that it's bad for parallelism [Can91] and bad for computer science in general [Bac78]. They say that without it, programs are easier to understand and reason about. Yet, it seems that every time I start to write a program I find myself wanting for it. It's called *the store*.

The *mutable* store, that is. I want for real destructive update in a real functional language (for my own, particular, subjective definition of 'real'). I wanted it for long enough that I decided I should take a PhD position and spend the next several years of my life trying to get it.

Soon after starting I came to realise two things:

1) That the problem was real, and that many people were aware of it.

2) That this was *not* a new problem.

1.2 The Problem

Functional programming is many things to many people, but at the heart of it is one central idea. Programs should be expressed in terms of higher order functions, referred to as *combining forms* in Backus's seminal paper [Bac78], instead of as imperative sequences of commands which update a global store.

The folklore promises that as functional programs admit more algebraic laws than their imperative counterparts, they will be easier to express and reason about. It also promises that functional programs have the potential to run faster, with the imagined speedup being partly due to freedom from the 'von Neumann bottleneck', that is sequential access to the global store, and partly due to optimising transforms which can be carried out due to the algebraic laws [dMS95].

After 30 years of intense research, several industry strength compilers [PJ94, NSvEP91, TBE+06, Mac91], and countless research papers, both of these promises have been delivered on — yet curiously, functional languages have not replaced imperative ones in mainstream software engineering.

There are a myriad of endlessly debated reasons for this apparent lack of use, and most will be quite familiar to the people likely to be reading this thesis. Often touted candidates include organisational inertia and marketing pressure exerted by large corporations seeking to cement their own particular language into the psyche of the industry programmer [GJSB05]. It is no doubt easy to form these opinions, especially if one is a researcher or academic in the field of programming languages. Almost by definition, we spend the majority of our time working with our own systems and attending conferences where the presentations are given by people in similar circumstances.

In recent years this situation has been recognised by the functional programming community itself, hence the creation of forums that seek to collect reports of industry experience with its languages [Wad04]. The conclusion of many of these presentations simply reiterates what we have known all along — that functional programming is wonderful and the application of higher order functions, pattern matching and strong typing (for some) leads to shorter development times and more reliable software.

By all accounts the community is thriving and much software is being written, yet the majority of it continues to be from graduate students and researchers in the field of programming language theory. Of this fact one can easily be convinced by visiting an arbitrary web-based job placement agency and comparing search results for "Haskell" or "O'Caml" versus any one of "Java", "Perl", "Ruby", "C++" or "Python".

Are Haskell and O'Caml destined to be The Velvet Underground of programming languages, where hardly anyone has heard them, but everyone who does forms a band?[1]

[1] After a quote attributed to Brian Eno. The Velvet Underground were a rock music group active in the late 60's, early 70's. They were highly influential, yet initially unsuccessful in a commercial sense.

Something's missing?

What if we were to take a step back from the glory of functional programming, and instead focus on what might be missing? After all, if functional languages could do everything that imperative languages could, *as well* as having strong typing, pattern matching, higher order functions and associated goodness, then at least there would be no *technically* based reason not to use them.

With this in mind, this thesis takes the position that an important missing feature from all current functional languages is real destructive update. A programmer should be free to update an arbitrary data structure in their program, with minimal runtime overhead, and with minimal interference from the type system or language design.

Note the use of the word "interference". In one sense, a language is a structure for formulating and communicating ideas, but in another it is a barrier to a true expression of intent. In an imperative language the programmer can update their data when and where they see fit, whereas in a typical functional language, they cannot. We seek to remove this barrier.

This work is embodied in the Disciplined Disciple Compiler (DDC)[2]. "Disciple" being the name of the language it compiles, and "Disciplined" invoking the type and effect discipline [TJ92b] of Talpin and Jouvelot which forms the basis of our type system. Wherever possible, we have avoided creating yet another functional language (YAFL) that no-one is likely to use. Disciple's syntax is based on Haskell, and DDC itself is written in Haskell. This allows us to leverage a huge body of existing people, ideas and code. Keeping the source and implementation languages similar will also make it easy to bootstrap DDC in future work.

As destructive update is the source of all our problems, we start with a discussion of *why* it should be included in a language in the first place. Having convinced ourselves that it is really needed, we will examine how it is supported in existing functional languages, and argue that this support is inadequate. We will discuss the notion of purity and how it benefits a language. We will also consider what it means for a language supporting destructive update and other side effects to be pure, and whether the formal notion of purity is useful in practice. Disciple allows arbitrary structures to be updated, and functions to have arbitrary side effects. Instead of relying on state monads, we use a type based analysis to recover mutability, effect and data sharing information from the program being compiled. We use an intermediate language similar to System-Fc [SCPJD07] and our analysis recovers enough information to do the same code transformation style optimisations as a highly optimising compiler such as GHC [PJ94]. We will discuss some of the practical problems with using lazy evaluation as the default method, and why space leaks are so common in lazy programs. Disciple uses strict evaluation by default, but allows the programmer to introduce laziness when desired. We use the type system to ensure that the combination of destructive update and laziness does not change the meaning of the program compared with the strict case.

This chapter outlines our approach and the reasons we have chosen it. Chapter 2 discusses the type system in detail, and Chapter 3 outlines the inference

[2]When dealing with a field that separates languages into "pure" and "impure", the religious connotations are already present. We make no apologies for the name.

algorithm. Chapter 4 describes our core language and gives a proof of soundness for its type system. Chapter 5 summarises what we have learned so far and suggests avenues for future work.

1.3 Why destructive update matters

Destructive update is the process of changing the value of an object in-place, by overwriting and hence destroying its old value. Without destructive update we cannot change the values of existing objects, only allocate new ones.

With deference to Turing completeness, destructive update is *simply not required* to write programs. Likewise, almost every feature of a particular language can be shown to be superfluous. Tiny systems such as the Lambda Calculus, Conway's game of life, and the Rule 30 cellular automata are Turing complete [Ren02, Coo04], and hence capable of universal computation. On the other hand, no one writes programs in them, at least not directly.

When considering language features we must always start from a practical, and therefore subjective viewpoint. When we say "destructive update matters", we mean that a large enough subset of programmers find it useful that it warrants consideration by all.

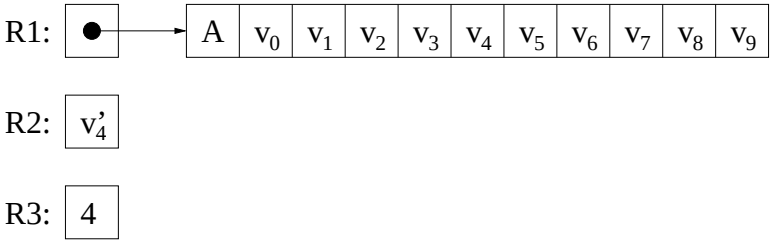
We suggest that destructive update furnishes the programmer with two important and powerful tools, and that these tools are either too cumbersome or too inefficient to create without it. The first tool is a set of efficient array-like data structures for managing collections of objects, and the second is the ability to broadcast a new value to all parts of a program with minimal burden on the programmer.

1.3.1 Efficient data structures require destructive update

For a mechanical device such as an internal combustion engine, efficiency is defined as the ratio of useful work output to the amount of energy input to the device [Gia00]. For a *computational* device such as a collection structure, we could reasonably define its efficiency as being the number of insert and update operations that can be completed per hardware clock cycle.

We pay no attention to the difficulty of designing the structure in the first place. Like internal combustion engines, the development of common data structures is best left to teams of experts, permitting the end user to focus on their own specific tasks.

When the number of objects to be managed is known beforehand, the simplest collection structure is the array. In a typical garbage collected runtime system, the allocation of an array requires just three machine instructions. We test the top-of-heap pointer to ensure enough space is available, write the object header word, and then advance the pointer. The update of a particular value is also straightforward. Suppose we have three registers: R1 holding a pointer to the array, R2 holding the new value, and R3 holding the index of the value to be updated. Many processors can perform this update with just one or two instructions [Sun02, Int06].



Of course, due to pipelining and cache effects, the number of machine instructions executed for an operation does not relate directly to the number of clock cycles used [HP96]. However, it is a usable approximation for this discussion, and we will consider the case of updating a flat array as approaching 100% efficiency for array-like structures.

Perfect efficiency would be achieved if every update operation completed in the minimum number of clock cycles possible on the available hardware. For most applications, perfect efficiency is unlikely to ever be achieved by a statically compiled program, as it is too difficult to accurately simulate pipeline states and data hazards in a multi-tasking system.

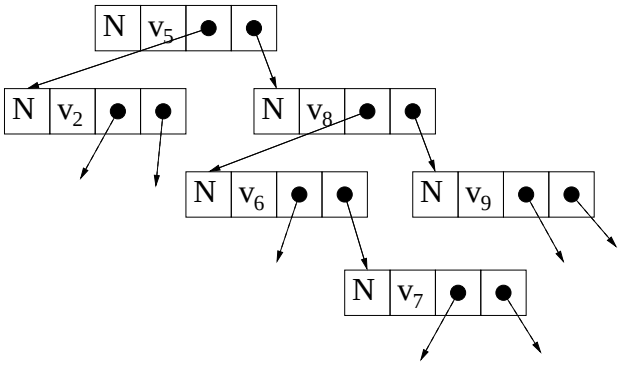
Ignoring cache effects, the efficiency of an array is independent of how many elements it containts. It costs no more to update a value in a 1000 element array than to update one in a 10 element array.

Functional arrays are unacceptably slow

Without destructive update we cannot change an array object once it is allocated, but we can side-step this problem by creating a new object instead of updating the old one. A simple method is to allocate a whole new array and copy the old values into it, with the new value in place of the one that is being updated. This works, but is a disaster for performance. Assuming we need one machine instruction for each value copied, performing an update on a 10 element array now requires 10 instructions, plus three to do the allocation. This represents a miserable 7.7% efficiency compared with a single instruction destructive update. For a 1000 element array we need at least 1003 instructions, representing approximately 0.1% efficiency. This is clearly unacceptable.

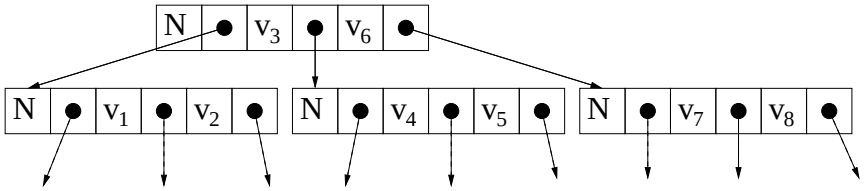
Tree structures are only a partial solution

We can recover some of this ground by using a tree structure instead of a flat array. If we store values in the internal nodes of a balanced binary tree then we need n nodes for n values, and the tree is ceil($\log_2(n)$) levels deep. Unfortunately, to change a value in the tree we must still allocate a new node. As this new node will be at a different address from the old one, we must then rebuild all of its parents so that the tree references the new node instead of the old one. For example, in the tree on the next page, to update v_7 we must reallocate the node that contains it as well as the nodes of v_6, v_8 and v_5.



For a small number of values, using a binary tree is *worse* than copying the whole array for each update, because each node contains an object header and two pointers instead of just a value. A balanced tree of 1000 elements is 10 levels deep, and as a rough approximation, half of the nodes lie at the bottom of the tree. If we were to update one of these nodes then we would need to reallocate all of its parents, which equates to 9 * 4 = 36 words of space. Not all nodes are at this level, but we haven't accounted for finding the node to be updated in the first place either. For a back-of-envelope calculation we could expect an average of at least 50 machine instructions to be required to update a node in this tree. This equates to 2% efficiency when compared with destructive array update of a similarly sized array.

Another option is to use trees of a higher order, perhaps a quadtree or a B-tree structure like the one shown in the following diagram. Adding more values per node reduces the depth of the tree. It also reduces the number of nodes we must rebuild when performing an update, but at the cost of making each node larger.



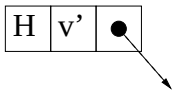
For a full (2,3)-tree with two keys and three branches per node, each node is 6 words long including the object header. Every extra level provides three times the number of nodes present in the previous level, and for 1000 values we need a little more than 6 levels. If we say that each node to be updated has an average of 5 parent nodes, this equates to 5 * 6 = 30 words of space to be reinitialised when updating a node. This isn't much better than the previous case.

Many algorithms naturally use an array as their primary collection structure. If, for the lack of destructive update, we are forced to a tree instead, then we automatically impose a $\log(n)$ slowdown on our algorithm's run time. To access an element in a tree we must traverse its structure, but array access can be performed in constant time. This slowdown is in addition to a substantial constant factor due to the extra book-keeping data that must be maintained, such as object headers and branch pointers that are not needed when using arrays. In [PMN88] Ponder gives a list of algorithms for which no equivalent, array-less algorithm of similar complexity is known.

The limit

There are an infinite variety of possible structures for simulating arrays, and trees represent just a few. By this stage, an astute reader may be thinking about all their own favourite structures and how much better they are than the ones outlined here [Oka98b, OG98]. As we are talking about machine instructions and constant factors, not just algorithmic complexity, there are also a huge variety of low level details to consider. Details include instruction selection, caching, data layout, pointer compression [LA05], and using "diff arrays" which rely on destructive update behind the scenes for efficiency, whilst presenting a functionally pure interface. An example of pointer compression is to replace each of the pointers in an object by offsets from a base pointer, instead of including the store address in full. This can result in substantial space savings for pointer heavy programs on 64 bit machines, where the size of a particular data structure is only a tiny fraction of the available address space. Diff arrays use a destructively updateable array for the current state of the structure, and updates to the structure are implemented by physically updating this array. Old states of the array are represented by a list of differences to apply to the current state, so old states become slower and slower to access as the program progresses.

There are many avenues for improvement, but without destructive update we are still hamstrung by the need to allocate objects to represent new program states. Consider a maximally efficient structure which requires only a single, tiny object to be allocated for each update. At the least, we could expect this new object to contain a header word, the new value, and a pointer to the rest of the structure:



However, the allocation and initialisation of this object will still require at least five machine instructions, three for allocation and two to write the new value and pointer. For some applications a constant five-fold slow down is of no consequence, but for others it is a deal breaker.

Tuples and records are also arrays

At the machine level, tuples and record objects are often very similar to arrays. We can implement records by representing each field as a pointer to the object containing its value, so at this level the record is simply an array of pointers. A record object with 5 fields would contain a header word and five pointers. Consider then the following record type from DDC:

```
data SquidS
    = SquidS
    { stateTrace           :: Maybe Handle
    , stateTraceIndent     :: Int
    , stateErrors          :: [Error]
    , stateStop            :: Bool
    , stateArgs            :: Set Arg
    , stateSigmaTable      :: Map Var Var
    , stateVsBoundTopLevel :: Set Var
    , stateVarGen          :: Map NameSpace VarBind
    , stateVarSub          :: Map Var  Var
    , stateGraph           :: Graph
    , stateDefs            :: Map Var Type
    , statePath            :: [CBind]
    , stateContains        :: Map CBind (Set CBind)
    , stateInstantiates    :: Map CBind (Set CBind)
    , stateGenSusp         :: Set Var
    , stateGenDone         :: Set Var
    , stateInst            :: Map Var (InstanceInfo Var Type)
    , stateQuantifiedVars  :: Map Var (Kind, Maybe Type)
    , stateDataFields      :: Map Var ([Var], [(Var, Type)])
    , stateProject         :: Map Var (Type, Map Var Var)
    , stateProjectResolve  :: Map Var Var
    , stateClassInst       :: Map Var [Fetter] }
```

This record represents the state of our type inferencer while it is reducing constraints. The meaning of the fields is not important for this discussion. We include this data type to make the point that real records can contain upwards of 22 separate fields. No matter how efficiently the internal sets, maps and graphs are implemented, without destructive update we cannot change the value of a field without rebuilding at least part of the record object. If we must rebuild it all, then this is at least 22 times slower than using destructive update.

In a language without destructive update we must allocate and initialize new objects to represent new program states. This a drastically less efficient alternative. In practice, even if a language does not support the destructive update of arbitrary structures, attempts are made to introduce it in a more restricted form. In [SCA93] Sastry presents an analysis to determine an order of evaluation for array access and update operations, that allows the updates to be implemented destructively. Besides being first order only, the trouble with many such analyses is that when they fail to introduce an update at an important point in the program, the programmer is left with little recourse to add it manually. There is also the problem of determining which updates *should* have been implemented destructively, but weren't.

As discussed in §1.6, Haskell includes a monadic sub-language that supports the destructive update of select structures such as arrays. However, this sub-language introduces its own problems, and algebraic data such as records and

lists cannot be similarly updated. In [San94] Sansom describes the time profile of an early version of GHC, and mentions that the use of a monadic mutable array over an association list improved the performance of type substitution by a factor of 10. When combined with other improvements, this resulted in a 20x speedup of the type checker as a whole.

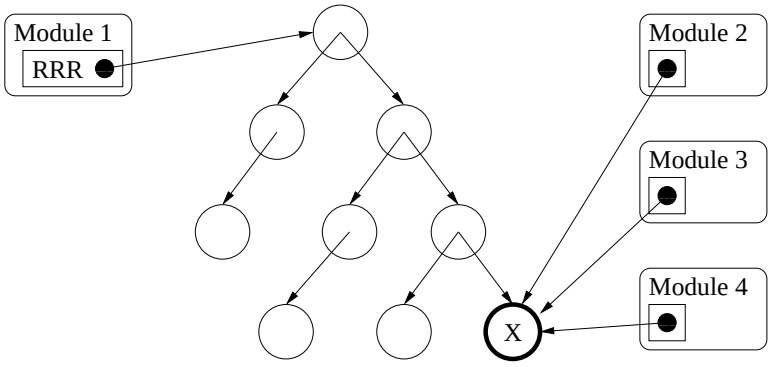
If a particular programmer does not use functional arrays or large records in their programs, then they may not be aware of the run-time cost of using them. However, those who do are looking down the barrel of a five fold slow-down, or worse, compared with other languages.

1.3.2 Destructive update helps to broadcast new values

There is an often rehearsed argument that the expressiveness of a language is more important than efficiency, because improvements to processor speed will quickly recover any lost ground. The standard counter is to say that the common man wants their new and faster computers to do *new and faster things*, not the original things less efficiently.

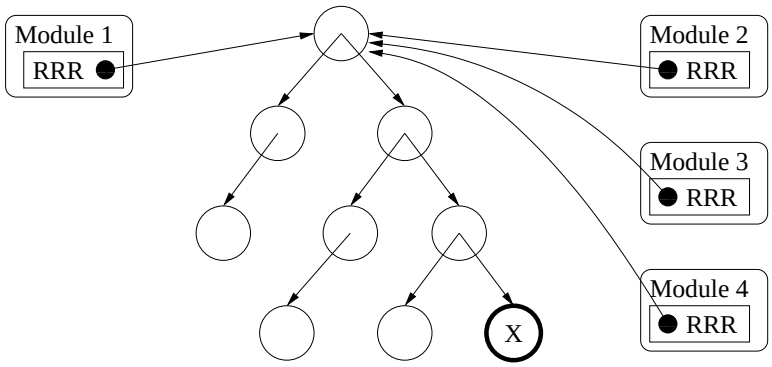
These arguments can also be applied to destructive update. "Surely", the antagonist offers, "a five fold slow-down is not *that* bad. Moore's law says we'll have that back in four years, and look at all the extra compiler optimisations we can do now that the language is pure!".

Computational efficiency may or may not matter to a particular programmer, but the level of abstraction offered by the language should matter to all. We will now move away from concerns over run time speed, and instead focus on the expressiveness of the language itself. Consider a set of program modules which all reference a single, shared value. This value could be a cursor position or timing value, something that changes often and is of interest to many separate modules. We will refer to this value as X. In a language with destructive update we can place X in a container structure and have each module access it via a pointer:



Using destructive update, one module can modify X and the new version is immediately visible to others. Notice that module 1 has a reference to the top of the container structure as well as a description of how to find the value of interest. In this example the container is a binary tree and the value is accessable by following three right branches from the root node. On the other side, modules 2, 3, and 4 do not need to know how to locate X within its container because they have a pointer directly to it. This second group of modules is not interested in the container or any other value contained within.

Without destructive update, a module cannot change X directly, nor can it change the pointers within client modules so that they reference any new objects it might create. The programmer is forced to rewrite all modules so that they reference the top level of the container structure, and include a description of how to find the value of interest.



By doing this we have reduced the level of abstraction in the program. Whereas modules 2, 3, and 4 are only interested in the value X, they must now also concern themselves with the container structure and how to find and update elements contained within.

This problem is compounded when a shared value is logically part of several containers. Perhaps X is also present in a graph structure, and the tree is used to represent a set of objects which must be written to disk when the program finishes. The modules that wish to update the shared value must have knowledge of all structures which contain it.

Shared values like the one described here are often needed to write interactive programs such as Frag [Che05]. In Haskell, variables holding timing values, mouse positions and other user interface states are typically bundled up into a record of IORefs. This in turn requires all code which accesses these values to be written in the IO monad, a point we will return to in §1.6.

Updating nested records in Haskell is painful

Haskell 98 has an conspicuously poor record system. In itself this is not a new observation, but we pause to discuss it because we feel the problem arises in part from the lack of destructive update in the ambient language. Standard complaints include the records not being light weight, not being extensible, and that all field names are in the top level scope [JPJ99]. In addition, we consider the syntax for updating nested records to be unusable.

The following code defines three record types with two fields each. Type `R1` contains type `R2`, and type `R2` contains type `R3`. Notice the prefixes `r1`, `r2` and `r3` on each field name. In Haskell, field names pollute the top level name space, so we can't have a field named `field1` in `R1` as well as in `R2` without creating a name clash.

```
data R1 = R1 { r1Field1  :: Int
             , r1Field2  :: R2 }

data R2 = R2 { r2Field1  :: Char
             , r2Field2  :: R3 }

data R3 = R3 { r3Field1  :: Bool
             , r3Count   :: Int }
```

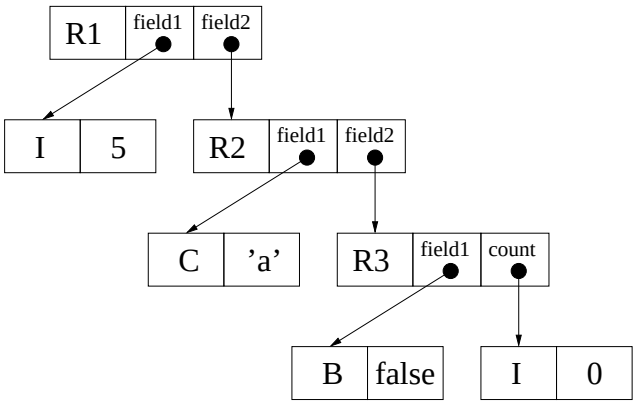
We will create a record value of type `R1` as an example. Similarly to the previous section, we treat the field `r3Count` as a shared value that many program modules will be interested in. When the record is created we will initialise this field to zero. The exact values used for the other fields are not important for this example.

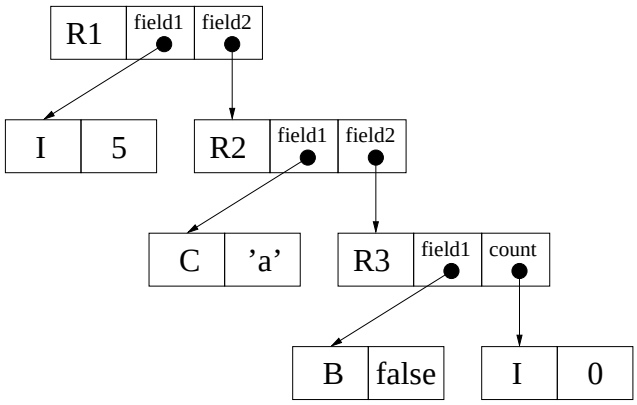
```
record1 = R1 { r1Field1 = 5
             , r1Field2 =
                  R2 { r2Field1 = 'a'
                     , r2Field2 =
                          R3 { r3Field1 = False
                             , r3Count  = 0 }}}
```

Extracting the counter field from the structure is straightforward. Each field name becomes a projection function which takes the record and produces the field value, for example `r1Field1 :: R1 -> Int`. We can make use of the function composition operator to extract the count field using a pleasing syntax:

```
count  = (r3Count . r2Field2 . r1Field2) record1
```

Updating the counter is another matter entirely. As we do not wish to modify the other fields in the structure, we must unpack and repack each level in turn. This process corresponds to reallocating parents when updating a node in a tree. Unfortunately, this time we cannot write a cute recursive function to do so because the records at each level have different types. The following diagram shows the structure of the nested records:





If we wish to change the *count* field in this structure to the value 1, we must allocate a new object containing this value. We must then rebuild the `R3`, `R2` and `R1` nodes so that the structure references this new object while retaining the pointers to the other nodes. Here is the gory Haskell expression:

```
record2
  = record1 { r1Field2 =
      (r1Field2 record1) { r2Field2 =
         ((r2Field2 . r1Field2) record1) { r3Count = 1 }}}
```

Clearly this is not something a typical programmer would enjoy writing. The field names and the variable `record1` are repeated twice each, the line must be broken into fragments to fit on the page, it does not indent well, and there is much visual noise.

It is worse when we need to update this field with a non-trivial expression. Consider the simple act of incrementing `r3Count`. We can use layout to reduce the noise, but it is still quite bad:

```
record3
  = record2 {
      r1Field2 = (r1Field2 record2) {
      r2Field2 = ((r2Field2 . r1Field2) record2) {
      r3Count  = (r3Count . r2Field2 . r1Field2) record2 + 1
    }}}
```

The need to write such tedious code to perform such a simple update would be a deal breaker for many programmers.[3]

Consider an equivalent statement in an imperative language, such as C++.

```
record2.field2.field2.count += 1
```

Destructive update allows the programmer to focus solely on the element of interest, while leaving the others untouched. Granted, the above statement does not have the same semantics as the Haskell version because it modifies the original object instead creating a new one. If this behaviour is required then many imperative, object oriented languages support a fragment such as:

```
record3 = record2.copy()
record3.field2.field2.count += 1
```

[3]It certainly is for the author.

To the surrounding program these two simple statements have the same effect as the Haskell expression shown above, with the added advantage that the concepts of *copy* and *update* are clearly separated.

In C++ we can make life even easier for the programmer. We can create a reference to the field of interest and increment it without any knowledge of the surrounding record structure:

```
int* countRef = &(record.field2.field2.count)

(*countRef) += 1;
```

Admittedly, references and destructive update can be used for evil as well as for good. Many confusing programs can be constructed in which different program modules communicate via shared mutable objects in a way which is entirely non-obvious to the casual observer, and almost impossible to debug. The counter to this argument is to say that confusing programs can be written in any language, and a good carpenter can hammer a nail into a wall without smashing their own fingers.

We should note that the problem of updating nested records in Haskell can be made easier by generic programming libraries such as 'Scrap Your Boilerplate' [LPJ03] (SYB) and DriFT [HJL06]. Although these systems can help, we feel they are not complete solutions as they lack the efficiency and ease of use of destructive update. SYB style systems tend to traverse uninteresting parts of a structure when performing an update, resulting in a significant performance penalty [Mit07]. DriFT is a preprocessor which can automatically generate update functions for each of the fields in a record, though it does not support nested update as per the previous example. When implementing DDC we defined tuples of get and set functions for each field, along with combinators to compose these functions and to update fields deep within nested records. We have found this to be a serviceable yet tedious approach, as we have not yet written a preprocessor to generate the tuples. This approach also does not solve the problem of field names being in top level scope.

1.4 What is purity?

Although the word *purity* has many varied meanings, most would agree that the following expression is pure:

$$(\lambda x.\ double\ x)\ (succ\ 5)$$

To reduce this expression call-by-value, we first evaluate the argument and then substitute into the body of the function.

$$\begin{aligned}&(\lambda x.\ double\ x)\ (succ\ 5)\\ &\longrightarrow (\lambda x.\ double\ x)\ 6\\ &\longrightarrow\ double\ \ 6\\ &\longrightarrow 12\end{aligned}$$

When we reduce the same expression call-by-name, we substitute the argument first, yielding the same result.

$$\begin{aligned}&(\lambda x.\ double\ x)\ (succ\ \ 5)\\ &\longrightarrow\ double\ (succ\ \ 5)\\ &\longrightarrow\ double\ \ 6\\ &\longrightarrow 12\end{aligned}$$

In the simply typed lambda calculus, the order in which function applications are evaluated does not affect the end result. This behavior is also known as the Church-Rosser property [Ros84], or *confluence*. Purity is of tremendous help to compiler writers because it gives us the freedom to reorder function applications during compilation, whilst preserving the meaning of the program. By reordering function applications we can expose many useful compiler optimisations [dMS95], and this well known identity of the *map* function is one such example:

$$map\ \ f\ (map\ \ g\ \ xs)\ \equiv\ \ map\ (f \circ g)\ \ xs$$

The first expression applies g to each element of the list xs yielding an intermediate list. It then applies f to each element of this list, yielding the result. In the second expression, the composition of f and g is applied to each element directly, without requiring the construction of an intermediate list. As long as we are free to reorder function applications, we can optimise a arbitrary program by rewriting expressions of the first form into the second.

The Glasgow Haskell Compiler includes a myriad of similar optimisations. Simple, frequently invoked rewrites are "baked-in" to the compiler proper, whereas more specific identities such the one above are part of the standard libraries, or are defined by the programmer directly. The recent work on stream fusion [CLS07] is a prime example of library defined rewrites which depend on purity.

In contrast, the following expression is decidedly not pure:

$$choose\ (inTrouble\ ())\ (launchMissiles\ \ 5)\ (eatCake\ \ 23)$$

The end result of this expression depends very much on the order of evaluation. The intention is for the first argument to execute first, before choosing *one* of the others. This ordering must be preserved in the compiled program, else the meaning of the program will be changed.

The concept of purity can also be applied to a language as a whole. With the Church-Rosser property in mind, Sabry defines a purely functional language to be one that satisfies the following criteria [Sab98]:

1. it is a conservative extension of the simply typed λ-calculus.
2. it has a well-defined call-by-value, call-by-need, and call-by-name evaluation functions (implementations), and
3. all three evaluation functions (implementations) are weakly equivalent.

We will consider the finer points of these criteria in a moment. Separate from Sabry's definition, the functional programming community generally recognises Haskell to be pure, while SML, Scheme, Fortran and C++ are said to be *impure* languages. However, note that Fortran and C++ are not extensions of the λ-calculus, so are not functional either.

Pure and *impure* are loaded terms

Sabry's definition contains several subtle points, not least of which are the words being defined. The Oxford English Dictionary (OED) gives several meanings for the word "impure". In regards to "a language or style", the word has the meaning of "containing foreign idioms or grammatical blemishes". In this context, the "foreign idioms" would include interactions with the outside world such as launching missiles or eating cake. These actions are not part of the formal definition of the lambda calculus, and it is not obvious what the result will be from the function names alone. The other meaning offered by the OED is along the lines of "not pure ceremonially; unhallowed", or "containing some defiling or offensive matter; dirty, unclean".

This is unfortunate terminology. In contrast, mathematical disciplines are sometimes separated into groups labeled pure and *applied*. The intention is that the more pure disciplines have no immediate practical applications, but are considered worthy of study because they increase our understanding of mathematics as a whole. On the other hand, the applied fields focus on using mathematics to achieve goals in other disciplines, such as physics. What was once considered a pure field may become more applied when a concrete application is found. For example, abstract algebra is now an integral part of the error control coding systems that we rely on for electronic communication [LDJC83].

Computational side effects

Granted, impure languages can be much harder to reason about, and the bulk of this thesis is about doing just that. Actions which affect the outside world must be implemented in the appropriate order, else the program may not yield the intended result. Other *internal* actions such as destructively updating data and then reading it back must also be sequenced appropriately.

When the end result of two expressions depends on the order in which they are evaluated, those expressions are said to have *computational effects*, or to *interfere* [Rey78].

Computational effects are also known as *side effects*, because an expression can return a value as well as "doing something else". This is another unfortunate

term. When a pharmaceutical product has a side effect, this is usually taken to be a *bad thing*. A pharmaceutical side effect is an undesirable result, but a computational side effect is often the entire reason for running a program in the first place.

Do we really need three implementations?

We return to Sabry's second and third criteria for a purely functional language:

2. it has a well-defined call-by-value, call-by-need, and call-by-name evaluation functions (implementations)

3. all three evaluation functions (implementations) are weakly equivalent.

At the time of writing, Haskell is call-by-need, and there is no formal definition of its implementation. The Haskell 98 report [PJ03a] contains a description of the syntax and English language notes on the intended meaning, but no formal operational or natural semantics. There are formal semantics for fragments such as the lazy lambda calculus [Abr90, Lau93b], but not the complete language. Whether this counts as having three "well defined" implementations is debatable.

SML has a formal semantics [MTHM97], yet it is call-by-value only. There is a lazy version of ML [Aug84], but not all features of SML are supported, notably exceptions and mutable references.

On the other hand, Haskell includes the *seq* combinator which can be used to turn a natively call-by-need application into a call-by-value one, within the same implementation. For example, the following application will evaluate call-by-need as default:

$$f\ exp...$$

However, we can force the argument to be evaluated in an approximately call-by-value manner by writing:

$$\begin{array}{ll}\textbf{let} & x = exp... \\ \textbf{in} & seq\ x\ (f\ x)\end{array}$$

By binding $exp...$ to the variable x and then passing this as the first argument to *seq*, we force it to be reduced to head normal form[4] [PJ87, PJ92] before substitution into f. This simulates call-by-value evaluation. However, in a lazy language the definition of "value" usually includes expressions that contain redexes, as long as there is no redex at top level. If we desire *hyper-strict* evaluation, where all possible redexes are reduced before application, then we need to make more judicious use of *seq*. We will return to this point in §1.8.

[4] In the STG machine on which GHC is based, function application is between variables and atoms. For this implementation an expression cannot be in *weak* head normal form without also being in head normal form.

Weak equivalence

Consider the following function application, where $\perp$ represents an expression that performs no IO actions, but does not terminate and thus does not yield a value.

$$(\lambda z.\ 5)\ \perp$$

When we evaluate this expression call-by-need, the variable z is not present in the body of the function, so we can simply discard the argument. However, if we use call-by-value, or the rewrite using *seq* from the previous section, then the evaluation of this expression will diverge.

Sabry's definition of *weak equivalence* accounts for this:

Let P be a set of programs, B be a set of observables and $eval_1$ and $eval_2$ be two partial functions (implementations) from programs to observables. We say $eval_1$ is weakly equivalent to $eval_2$ when the following conditions hold:

- If $eval_1(P) = B$ then either $eval_2(P) = B$ or $eval_2(P)$ is undefined
- If $eval_2(P) = B$ then either $eval_1(P) = B$ or $eval_1(P)$ is undefined

When discussing the observables of a program we will omit the time taken for it to evaluate. We will consider only the final value returned, and the IO actions performed, as being its "result". If one implementation evaluates more slowly than another, but otherwise returns the same value and performs the same actions, then by the above definition they are still equivalent. If this were *not* the case then most compiler optimisations would change the meaning of the program. In fact, if they *didn't* change its meaning, by making it faster, then they would be useless.

If evaluation time is not observable then, by rights, non-termination should not be observable either. In a practical sense, the only way we could observe that a program did not terminate is by waiting an infinite time for it to complete. For this reason we take non-termination as being part of the "undefined" in the definition of weak equivalence. Under this definition, if we evaluate a program with one implementation and it terminates, but in another it does not, the implementations are still weakly equivalent. As non-termination is not observable, by extension it is also not an action, nor does it correspond to a side effect.

We also take "undefined" to mean that the program would not compile due to a type error. Disciple is call-by-value by default, but also supports call-by-need evaluation. We shall see in §2.3.9 that if we attempt to suspend a function application that has an observable effect, something that would otherwise change the meaning of the program with respect to the call-by-value case, the compiler will detect this and report a type error. By Sabry's definition we argue that this makes Disciple a purely functional language — even though arbitrary structures can be destructively updated, and functions can have arbitrary side effects.

We consider Terauchi and Aiken's system of witnessing side-effects [TA05] to be purely functional for the same reason. Like ours, their system supports the update of mutable references at arbitrary points in the program. It also provides a type system to ensure *witness race freedom*, which means there are enough data dependencies in the program to prevent a parallel reduction of it from having an indeterminate result.

Expressive power and operational equivalences

Of course, Sabry's definition of what a purely functional language is may or may not correspond to the informal understanding of it by the functional programming community at large. There is also the question of whether "purely functional" should mean the same thing as "pure", as in our experience these terms are used interchangeably.

Whether or not purely functional languages are somehow intrinsically better than impure ones is a moot point. Sabry's discussion of why Haskell is purely functional hinges on the fact that monadic programs can be treated as producing a *description* of the IO actions to be performed, instead of executing them directly. The actual execution only happens when computing the observable result of the description, a process conceptually separate from evaluation. However, as GHC uses monads to support mutable references, and these references can contain lambda terms, the "evaluation" and "observation" functions would need to be defined as co-routines. As only the evaluation part adheres to Sabry's definition, we feel that the term "purely functional", when applied to a language as a whole, is a description of the formalisation of that language, and not of the feature set presented to the user. It is not about the "lack of side effects" or "lack of mutable state", because Haskell provides both of these.

On the other hand, the term "pure" when applied to a single expression has a more widely accepted meaning. A pure expression has no side effects and its evaluation can be reordered with any other expression without affecting its result. If the evaluation cannot be safely reordered with another then we will call the expression "impure". The ability to reorder expressions is an *operational equivalence*.

In [Fel91] Felleisen notes that: *"an increase in expressive power is related to a decrease in the set of "natural" (mathematically appealing) operational equivalences"*. In §1.3 we saw that the omission of destructive update from a language reduces its expressiveness because it means there is no easy way to update shared values contained within data structures. By "no easy way" we mean that if we had a program that destructively updated a shared value, and we had to rewrite that program without using update, then we would need to perform far reaching changes to the code.

Our language, Disciple, is based on Haskell and is extended to support the destructive update of arbitrary structures, and some other impure features. We add these features to increase the expressive power of the language, but by doing so we lose certain operational equivalences. The game is then to win a high degree of expressiveness while losing only a small number of operational equivalences. Expressiveness is not easy to quantify, but we touched on it in our discussion of why destructive update matters. In Chapter 4 we discuss how we have organised our core language so that only the parts of the program that use impure features lose operational equivalences. This allows the full gamut of program optimisations to be applied to the pure parts, which we feel is a fair compromise.

Non-termination is not an effect

In Disciple, the time taken for a program to evaluate is not formally observable, though we do not consider this approach to be the only valid option. For

example, in a hard real time system the time taken to evaluate a program is just as important as its final result (by definition). If a program written in such a system cannot produce its result in the required time, then the program is wrong. In Disciple we have no formal support for such requirements, other than good intentions. As we leave run time unobservable, then it follows that non-termination should not be observable either. This is a difference to Tolmach's system [Tol98] which has specific support for pure but potentially non-terminating computations. We will return to this in §4.5.1. Although non-termination is not observable, we certainly don't want to *introduce* it into an otherwise terminating program. By the definition of weak equivalence, introducing non-termination would not change a program's meaning, but it would certainly aggravate the programmer.

DDC is a general purpose compiler and its role is to produce a binary that runs "fast enough" to satisfy the programmer. This "fast enough" is not well defined, though the general rule is the faster the better. This has implications for our handling of potentially non-terminating expressions. Suppose $\perp$ represents some non-terminating, but otherwise effect free expression. If, when compiling a program, we see an expression of the form $((\lambda z.\ 5)\ \perp)$ then we will be free to perform a compile time β-reduction and rewrite this to 5, or not. Rewriting it reduces the run time of the program, from infinite to something finite, which we take as being a good thing. However, this treatment of non-termination is at odds with some styles of imperative programming that use functions like:

```
void loop ()
{
    while (true)
        ;
}
```

Although this function appears to do nothing, as part of a larger program it may be doing nothing for a particular purpose. For example, programs in embedded systems are often based around interrupt service routines (ISRs). Such programs typically use the main routine to set up a number of ISRs, and then enter an endless loop. When a hardware device requires service, or a timer expires, the processor calls the ISR, does some computation and returns to the main loop. In this case the program is not supposed to terminate, and it would be wrong to "improve" it by eliminating the loop. In DDC we handle this by requiring endless loops to contain a side effecting function. For example:

$$\mathit{loop}\ () = \textbf{let}\ \_\ =\ \mathit{sleep}\ 1\ \textbf{in}\ \mathit{loop}\ ()$$

The type of *sleep* will include an effect that signals to the compiler that the function is evaluated for some reason other than to gain its return value. We discuss effect types in §2.3.

Referential Transparency

Purity is related to the notion of *referential transparency*. A language is said to be referentially transparent if any subexpression can be replaced by any other that is equal in value, without affecting the end result of the program [SS90].

Reusing our previous example:

$$(\lambda x.\ double\ \ x)\ (succ\ \ 5)$$

If we take $(succ\ \ 5)$ to have the value 6 then we are free to replace any instance of $(succ\ \ 5)$ with this value, without changing the result of the program:

$$(\lambda x.\ double\ \ x)\ 6$$

This is clearly valid, because it is the result we would have obtained when reducing the expression call-by-value anyway, and we know that the lambda calculus is pure.

For contrast, consider the following expression which reads a character from the console:

$$getChar\ ()$$

Is this expression referentially transparent? If we were to say that the "value" of $getChar\ ()$ is a character, then the answer would be no. The character returned will depend on which key the user presses, and could be different every time. We cannot take the first character that the user enters and use this to replace all other instances of $getChar\ ()$ in the program without changing its meaning.

We could equally say that this expression does not in fact *have* a value separate from the context in which it is evaluated. Knowledge of the return value is inextricably linked to knowledge of what key the user will press. Saying that the "value" of $getChar\ ()$ is a just character is a gross oversimplification.

Another way of looking at this is to say that the expression $getChar\ ()$, and the character value, are not *observationally equivalent* [Gun92]. The *observed result* of evaluating a character is just the character, but the evaluation of $getChar\ ()$ also changes (or examines) the state of the outside world.

If desired, we could deal with this problem by simply embedding the notion of "the outside world" directly into the space of values. Once this is done we might then pretend that the language was referentially transparent all along. In Haskell syntax, we could give *getChar* the following type:

$$getChar :: World \rightarrow (Char, World)$$

This function takes the previous state of the world and returns a character along with the new world. The technique of threading the world through IO functions goes back to at least FL [AWW91], though the state parameter was named the *history* instead of the world.

To construct a useful example, we would also like to have a function which prints characters back to the console:

$$putChar :: Char \rightarrow World \rightarrow World$$

The problem of manufacturing the initial world can be solved by introducing a primitive function which executes the program, similarly to the *main* function in C or Haskell.

$$runProg :: (World \rightarrow World) \rightarrow ()$$

Now we can can write a program which reads a character and prints it back to the user:

```
let prog world
 =   (let   (c, world2)   = getChar world
            world3        = putChar c world2
      in    world3)
in runProg prog
```

Whether we chose to swallow these definitions would likely depend on whether we had constructivist or intuitionistic tendencies. As it is impossible to actually construct a value of *World* type, we cannot replace an expression such as (*getChar world*) with its resulting value, like we did with (*succ* 5). We could replace it with an expression such as (*id* (*getChar world*)), but that seems pointless.

Nevertheless, programming in this style has an important advantage. We can write our compiler as though the language were *indeed pure and referentially transparent*, because the act of passing around the world explicitly introduces the data dependencies needed to enforce the desired sequence of effects. In addition, we do not actually need to construct the world value at all. At runtime can we can simply pass around a dummy value, or eliminate the world passing entirely during compilation, once it has served its purpose. This allows the programmer to manage the problem, but the burden of correctness is theirs. For some programs, failing to supply adequate data dependencies can cause unexpected results at runtime.

Data Dependencies

Consider the following expression:

f (*putStr* "*hello*") (*putStr* "*world*")

In what order should these two strings be printed? If we read this as a curried, call-by-value application, then the expression is equivalent to:

(*f* (*putStr* "*hello*")) (*putStr* "*world*")

In this case the output would be: "*worldhello*". On the other hand, if the compiler did a left to right conversion to administrative normal form during desugaring, then we could also read the expression as:

```
let   x = putStr "hello"
      y = putStr "world"
in    f x y
```

If the runtime system then evaluated these bindings top to bottom, in a call-by-value manner, then the output would instead be: "*helloworld*". If evaluation was call-by-name then it would depend on which order *f* used its arguments, if at all.

If we instead converted the expression to C99, its result would be undefined by the language standard [C05]. In C99, side effects are only guaranteed to be completed at each *sequence point*, before moving onto the next one. There is a sequence point just before the call to a function, but only *after* all arguments

are evaluated. Each argument does not have its own sequence point, so the compiler is free to call the *putStr* functions in any order.

If we do not wish to rely on the order specified (or not) by the language standard, we could instead use our world passing mechanism to enforce a particular sequence:

```
let prog world
 =   (let   (x, world2) = putStr "hello" world
            (y, world3) = putStr "world" world2
      in    fun x y world3)
in runProg prog
```

Now there there is no ambiguity.[5] Assuming that *putStr* is an atomic, primitive operation which is strict in both arguments, the first occurrence must be evaluated before the second because we need to pass the token bound to *world2* to the next occurrence.

Unfortunately, this mechanism falls apart if we mix up the variables binding the world token, such as *world* and *world2*. Our program can also give unexpected results if we accidentally re-use them:

```
let prog world
 =   (let   (x, _) = putStr "hello" world
            (y, _) = putStr "world" world
      in    fun x y world)
in runProg prog
```

In this case we have passed the same token to each instance of *putStr*. In a call-by-need language such as Haskell, and in the absence of data dependencies, the order in which let bindings are evaluated depends on the order their values are demanded by the surrounding program.

1.5 Linear and uniqueness typing

Linear typing is a way to enforce that particular values in a program, like our *world* token, are used in a single threaded manner. Linear values can be used once, and once only, and cannot be discarded [Wad90b]. Uniqueness typing combines conventional typing with linear typing so that non-linear values, capable of being shared and discarded, can exist in the same program as linear values [BS94]. This can be done by adding sub-typing and coercion constraints between uniqueness variables as in Clean [BS93], or more recently, by using boolean algebra and unification as in Morrow [dVPA07].

With uniqueness typing we can give *putStr* the following type:

$$putStr :: String^{\times} \xrightarrow{\times} World^{\bullet} \xrightarrow{\times} World^{\bullet}$$

The first $\bullet$ annotation indicates that when we apply this function, the world token passed to it must not be shared with any other expression. On the right of the arrow, the $\bullet$ indicates that when the function returns, there will be no other references to the token bar this one. This forces the world token to be used in

[5] Or at least less ambiguity, we're still glossing over the actual operational semantics of the language.

a single threaded manner, and not duplicated. In contrast, the $\times$ annotation indicates that the *String* and function values may be shared with other parts of the program.

Using this new type explicitly disallows sharing the world as per the example from the previous section:

```
let prog world
 =   (let   (x, _) = putStr "hello" world
            (y, _) = putStr "world" world
      in    fun x y world)
in runProg prog
```

Here, the world token passed to *putStr* is non-unique because there are three separate occurrences of this variable. On an operational level, we can imagine that in a call-by-need implementation, a thunk is created for each of the let bindings, and each of those thunks will hold a pointer to it until they are forced.

Uniqueness typing can also be used to introduce destructive update into a language whilst maintaining the illusion of purity. From this point on we will elide $\times$ annotations on function constructors to make the types clearer.

Using the Morrow [dVPA07] system we could define:

$$\begin{aligned} &\mathit{newArray} && :: \mathit{Int}^{\times} \longrightarrow (\mathit{Int}^{\times} \rightarrow a^{u}) \longrightarrow \mathit{Array}^{\bullet}\ a^{u} \\ &\mathit{update} && :: \mathit{Array}^{\bullet}\ a^{u} \longrightarrow \mathit{Int}^{\times} \xrightarrow{\bullet} a^{u} \xrightarrow{\bullet} \mathit{Array}^{\bullet}\ a^{u} \end{aligned}$$

newArray takes the size of the array, a function to create each initial element, and produces a unique array. The u annotation on the type variable a is a uniqueness variable, and indicates that the array elements are polymorphic in uniqueness. The *update* function takes an array, the index of the element to be updated, the new element value, and returns the updated array. Notice the uniqueness annotations on the two right most function arrows of *update*. As we allow partial application we must prevent the possibility of just the array argument being supplied and the resulting function additionally shared. When applying a primitive function like *update* to a single argument, many implementations will build a thunk containing a pointer to the argument, along with a pointer to the code for *update*. If we were to share the thunk then we would also share the argument pointer, violating uniqueness.

Making the array unique forces it to be used in a single threaded manner. This in turn allows the runtime system to use destructive update instead of copy when modifying it. We can do this whilst maintaining purity, as uniqueness ensures that only a single function application will be able to observe the array's state before it is updated.

To read back an element from the array we can use the select function:

$$\mathit{select} :: \mathit{Array}^{\bullet}\ a^{\times} \longrightarrow \mathit{Int}^{\times} \xrightarrow{\bullet} (a^{\times},\ \mathit{Array}^{\bullet}\ a^{\times})^{\bullet}$$

select takes an array, the index of the element of interest and returns the element and the original array. As the tuple returned by the function contains a unique array it must also be unique. This is known as *uniqueness propagation*[BS93]. Similarly to the partial application case, if the tuple could be shared by many expressions then each would also have a reference to the array, ruining uniqueness.

Notice that it is only possible to select *non-unique* elements with this function. After *select* returns there will always be two references to the element, the one returned directly in the tuple and the one *still in the array*.

One way to work around this problem is to replace the element of interest with a dummy at the same moment we do the selection. Of course, once we have finished with the element we must remember to swap it back into the array. By doing this we can preserve uniqueness, but at the cost of requiring a different style of programming for unique and non-unique elements.

$$replace :: Array^{\bullet}\ a^{\bullet} \longrightarrow Int^{\times} \xrightarrow{\bullet} a^{\bullet} \xrightarrow{\bullet} (a^{\bullet},\ Array^{\bullet}\ a^{\bullet})^{\bullet}$$

Uniqueness typing goes a long way towards introducing destructive update into a language, while maintining the benefits of purity. Unfortunately, besides the added complexity to the type system, programs using it can become quite verbose. Having the required data dependencies in one's code is all well and good, but manually plumbing every unique object around the program can become tedious.

We will take a moment to meditate on the following type signature, from the `analtypes` module of the Clean 2.2 compiler source code:

```
checkKindsOfCommonDefsAndFunctions
    :: !Index !Index !NumberSet ![IndexRange]
       !{#CommonDefs} !u:{# FunDef} !v:{#DclModule}
       !*TypeDefInfos !*ClassDefInfos !*TypeVarHeap
       !*ExpressionHeap !*GenericHeap !*ErrorAdmin
    -> ( !u:{# FunDef}, !v:{#DclModule},  !*TypeDefInfos
       , !*TypeVarHeap , !*ExpressionHeap , !*GenericHeap
       , !*ErrorAdmin)
```

This function has thirteen arguments, and the returned tuple contains 7 components. The `!`, `#` and `*` are strictness, unboxedness and uniqueness annotations respectively, and {`a`} denotes an array of elements of type `a`.

Admittedly, we did spend a few minutes looking for a good example, but the verbosity of this signature is not unique among its brethren. We are certainly not implying that the Clean implementer's coding abilities are anything less than first rate. However, we do suggest that the requirement to manually plumb state information around a program must be alleviated before such a language is likely to be adopted by the community at large. With this point in mind, we press on to the next section.

1.6 State monads

In the context of functional programming, a monad is an abstract data type for representing objects which include a notion of sequence. Introduced by Moggi [Mog89] and elaborated by Wadler and others [Wad90a, PJW92, Lau93a, LHJ95], they are a highly general structure and have been used for diverse applications such as IO, exceptions, strictness, continuations and parsers [LM01, HM98].

In Haskell, the primary use of the general monad structure is to hide the plumbing of state information such as world tokens, and the destructively updateable

arrays from the previous section. For example, in thread-the-world style, a function to read an *Int* from the console would have type:

$$getInt :: World \rightarrow (Int, World)$$

This signature has two separate aspects. The *Int* term in the tuple gives the type of the value of interest, while the two occurrences of *World* show that this function also alters the outside world. We can separate these two aspects by defining a new type synonym:

$$\textbf{type}\ IO\ \ a = World \rightarrow (a,\ \ World)$$

We can then rewrite the type of *getInt* as:

$$getInt :: IO\ \ Int$$

This new type is read: "*getInt* has the type of an IO action which returns an *Int*". Note that we have not altered the underlying type of *getInt*, only written it in a more pleasing form. We can also define a function *printInt*, which prints an *Int* back to the console:

$$printInt :: Int \rightarrow IO\ ()$$

By applying the *IO* type synonym we can recover its thread-the-world version:

$$printInt :: Int \rightarrow\ World \rightarrow ((),\ \ World)$$

The magic begins when we introduce the *bind* combinator, which is used to sequence two actions:

$$\begin{array}{l} bindIO :: IO\ \ a \rightarrow (a \rightarrow IO\ \ b) \rightarrow IO\ \ b \\ bindIO\ \ m\ \ f \\ \quad = \lambda\ world. \\ \qquad \textbf{case}\ m\ \ world\ \ \textbf{of} \\ \qquad\quad (a, world') \rightarrow f\ \ a\ \ world' \end{array}$$

bindIO takes an IO action m, a function f which produces the next action in the sequence, and combines them into a new action which does both. In a lazy language such as Haskell we use a case-expression to force the first action to complete before moving onto the second. In a default-strict language like Disciple we could write *bind* using a let-expression, which would have the same meaning:

$$\begin{array}{l} bindIO :: IO\ \ a \rightarrow (a \rightarrow IO\ \ b) \rightarrow IO\ \ b \\ bindIO\ \ m\ \ f \\ \quad = \lambda\ world. \\ \qquad \textbf{let}\ (a,\ \ world') =\ \ m\ \ world \\ \qquad \textbf{in}\ \ \ f\ \ a\ \ world' \end{array}$$

We also need a top-level function to run the whole program, similar to *runProg* from before:

$$\begin{array}{l} runIO :: IO\ \ a \rightarrow a \\ runIO\ \ m = m\ \ TheWorld \end{array}$$

In this definition, *TheWorld* is the actual world token value and is the sole member of type *World*. In a real implementation, *World* could be made an

abstract data type so that client modules are unable to manufacture their own worlds and spoil the intended single-threadedness of the program. We would also need to ensure that only a single instance of *runIO* was used.

Here is a combinator that creates an action that does nothing except return a value:

$$
\begin{aligned}
&\mathit{returnIO} :: a \to \mathit{IO}\ a \\
&\mathit{returnIO}\ x = \lambda\, \mathit{world}\ .\ (x, \mathit{world})
\end{aligned}
$$

Now we can write a program to read two integers and return their sum, without needing to mention the world token explicitly:

$$
\begin{aligned}
&\mathit{runIO} \\
&\quad (\mathit{bindIO}\ \ \mathit{getInt}\ (\lambda x. \\
&\quad\ \mathit{bindIO}\ \ \mathit{getInt}\ (\lambda y. \\
&\quad\ \mathit{returnIO}\ (x + y))))
\end{aligned}
$$

In Haskell we can use any function of two arguments infix by surrounding it with back-quotes. We can also use the function composition operator \$ to eliminate the outer parenthesis:

$$
\begin{aligned}
&\mathit{runIO}\ \ \$ \\
&\quad \mathit{getInt}\ \ `\mathit{bindIO}`\ \lambda x. \\
&\quad \mathit{getInt}\ \ `\mathit{bindIO}`\ \lambda y. \\
&\quad \mathit{returnIO}\ (x + y)
\end{aligned}
$$

Finally, by using *do-notation* and the monad constructor class [Jon93] we can hide the uses of *bindIO* and write this program in a more familiar style:

$$
\begin{aligned}
&\mathit{runIO}\ \ \$ \\
&\ \mathbf{do}\ \ x \quad \leftarrow\ \mathit{getInt} \\
&\qquad\ y \quad \leftarrow\ \mathit{getInt} \\
&\qquad\ \mathit{returnIO}\ (x + y)
\end{aligned}
$$

Representing effects in value types is a double edged sword

The use of state monads in this way has several benefits. First and foremost, by using *bindIO* we have eliminated the need to manually plumb the world token around our programs. We can also use state monads to manage *internal* state by replacing the world token with references to these structures. Additionally, because monads are a general data type whose application is not restricted to just IO and state, we can define combinators which work on *all* monads including lists, exceptions, continuations and parsers.

Including effect information in types also aids program documentation. Programmers often write code comments to record whether certain functions perform IO actions or use internal state. By including this information directly in type signatures we leverage the compiler to *check* that this documentation remains valid while the program is developed.

However, the fact that effect information is represented in the space of *values* and *value types* is a double edged sword. On one hand, we did not need any specific support from the language to define our IO monad. On the other hand, functions which perform IO actions (still) have different structural types compared to ones that do not.

For example, a function which doubles an integer and returns the result would have type:

$$double :: Int \to Int$$

A function which doubles an integer as well as printing its result to the console would have type:

$$doubleIO :: Int \to IO\ Int$$

Imagine that during the development of a program we wrote a function that uses the first version, *double*:

$$\begin{array}{l} fun :: Int \to Int \\ fun\ x \\ =\ \mathbf{let}\ \ldots\ = \ldots \\ \qquad\quad x' = double\ x \\ \qquad\quad y = \ldots \\ \quad \mathbf{in}\ x' + y \end{array}$$

Suppose that after writing half our program we then decide that *fun* should be using *doubleIO* instead. The definition of *fun* we already have uses a let-expression for intermediate bindings, but now we must refactor this definition to use the do-notation, or use an explicit *bind* combinator to plumb the world through. For the do-notation, we must change the binding operator for our monadic expression to $\leftarrow$, as well as adding a **let** keyword to each of the non-monadic bindings:

$$\begin{array}{l} fun :: Int \to IO\ Int \\ fun\ x \\ =\ \mathbf{do}\ \ \mathbf{let}\ \ldots = \ldots \\ \qquad\quad x' \leftarrow doubleIO\ x \\ \qquad\quad \mathbf{let}\ \ldots = \ldots \\ \qquad\quad x' + y \end{array}$$

The type of *fun* has also changed because now *it* performs an IO action as well. We must now go back and refactor all other parts of our program that reference *fun*. In this way the *IO* monad begins to infect our entire program, a condition colloquially known as *monad creep* [Lou08] or *monaditis* [Kar08]. Although we have hidden the world token behind a few layers of syntactic sugar, it is still there, and it still perturbs the style of our programs. The space of values and the space of effects are conceptually orthogonal, but by representing effects as values we have muddled the two together.

One could argue that in a well written program, code which performs IO should be clearly separated from code which does the "real" processing. If this were possible then the refactoring problem outlined above should not arise too often. However, as monads are also used for managing *internal* state, and such state is used in so may non-trivial programs, all serious Haskell programmers will have suffered from this problem at some point. In practice, the refactoring of programs between monadic and non-monadic styles can require a substantial amount of work [LNSW01].

Haskell has fractured into monadic and non-monadic sub-languages

Being a functional language, programs written in Haskell tend to make heavy use of higher-order functions. Higher-order functions serve as control structures similar to the **for** and **switch** statements in C, with the advantage that new ones can be defined directly in the source language.

This heavy use of higher-order functions aggravates the disconnect between the monadic and non-monadic styles of programming. Every general purpose higher-order function needs both versions because monads are so often used to manage internal state. Consider the *map* function which applies a worker to all elements of a list, yielding a new list:

$$
\begin{aligned}
&map :: (a \to b) \to [a] \to [b] \\
&map\ f\ [\,] \qquad\quad = [\,] \\
&map\ f\ (x : xs) \quad = f\ x : map\ f\ xs
\end{aligned}
$$

This definition is fine for non-monadic workers, but if the worker also performs an IO action or uses monadic state then we must use the monadic version of *map* instead:

$$
\begin{aligned}
&mapM :: Monad\, m \Rightarrow (a \to m\ b) \to [a] \to m\ [b] \\
&mapM\ f\ [\,] \qquad\quad = return\ [\,] \\
&mapM\ f\ (x : xs) \\
&=\ \mathbf{do} \quad x' \qquad\quad \leftarrow f\ x \\
&\qquad\quad xs' \qquad\quad \leftarrow mapM\ f\ xs \\
&\qquad\quad return\ (x' : xs')
\end{aligned}
$$

Interestingly, we can make the non-monadic definition of *map* redundant by deriving it from this monadic one. We will use the identity monad, which contains no state and does not allow access to the outside world. This monad is just an empty shell which satisfies the definition:

$$
\begin{aligned}
&map :: (a \to b) \to [a] \to [b] \\
&map\ \ f\, xx = runIdentity\ (mapM\ (\lambda x.\, return\ (f\ x))\ \ xx
\end{aligned}
$$

Although we have no proof, we believe that it is possible to transform at least all second order monadic functions to similar non-monadic versions in this way. It is a pity then that the standard Haskell libraries are missing so many monadic versions. For example, the `Data.Map` package of GHC 6.10.1, released in November 2008, defines a finite map collection type that includes the functions *map*, *mapWithKey* and *mapAccum* among others. The types of these functions are:

$$
\begin{aligned}
&map \qquad\quad :: (a \to b) \qquad\quad \to Map\ \ k\ a \to Map\ \ k\ b \\
&mapWithKey \quad :: (k \to a \to b) \qquad \to Map\ \ k\ a \to Map\ \ k\ b \\
&mapAccum \qquad :: (a \to b \to (a, c)) \quad \to a \to Map\ \ k\ b \to (a, Map\ \ k\ c)
\end{aligned}
$$

There are no equivalent *mapM*, *mapWithKeyM* and *mapAccumM* functions in this library. In fact, there are no monadic versions for *any* of the `Data.Map` functions. The *Map* data type is also abstract, so if the programmer wants to apply a monadic worker function to all of its elements then life becomes troublesome. One solution is to convert the entire structure to a list and use *mapM* discussed earlier. Of course, doing this will incur a performance penalty if the compiler is unable to optimise away the intermediate lists.

The lack of monadic versions of functions is not confined to the `Data.Map` library. GHC 6.10.1 also lacks monadic versions of the list functions *find*, *any*

and *span*. If monads were mostly used for domain specific applications, then the lack of library functions may not hurt in practice. For example, we have used the Parsec monadic parser combinator library [LM01] in the implementation of DDC. During development we mainly used the parser specific combinators provided by the library, and doubt that we could even think of a sensible use for *mapAccumM* in this context.

On the other hand, the management of IO and internal state is not a domain specific problem. State monads permeate the source code for many well known Haskell applications such as darcs and the aptly named XMonad window manager [SJ07].

We do not feel that the lack of monadic library functions is due to poor performance on the part of library developers. Similarly, the lack of a standard linked list library in C99 can easily be blamed on the absence of a polymorphic type system in the language. In C99 there is no way to express a type such as $length :: [a] \rightarrow Int$, so programmers tend to roll their own list structures every time. A list of integers could be defined as:

```
struct ListInt { int x; struct ListInt* xs; };
```

In C99, functions over lists can be succinctly written as for-loops:

```
int lengthListInt (struct ListInt* list)
{
    int len = 0;
    for (struct ListInt* node = list;
         node != 0;
         node = node->xs)
        len ++;
    return len;
}
```

This works, but the programmer must then define a separate version of each list function for every element type in their program. Either that or abuse the `void*` type. More commonly, the definitions of simple list functions are typed out again and again, and more the complex ones are defined with macros. A library writer cannot hope to create functions for every possible element type, so we are left with no standard list library at all.

Similarly, Haskell does not provide a convenient way to generate both monadic and non-monadic versions of a function, nor does it provide an easy way to abstract over the difference. Programmers are taught not to cut and paste code, so we are left with one version of each function but not the other.

Monad transformers produce a layered structure

Monad transformers [LHJ95] offer a convenient way of constructing a monad from several smaller components, each providing a different facet of its computational behavior. The resulting data type is known as a *monad stack*, due to the layered way of constructing it.

For example, version 0.8 of the XMonad window manager uses a stack providing configuration information, internal state and IO:

$$\textbf{newtype}\ X\ a\ =\ X\ (ReaderT\ \ XConf\ (StateT\ \ XState\ \ IO)\ a)$$

This type is constructed by applying two monad transformers, *ReaderT* and *StateT* to the *inner monad*, *IO*. *StateT* extends *IO* with the ability to access the *XState* record type, while *ReaderT* extends it with the ability to access configuration information stored in the *XConf* record type.

The implementation of DDC's type inferencer also uses a monad stack built with *StateT* and *IO*. In this case, *StateT* supplies access to the current state of the algorithm while *IO* provides a destructively updatable array used to represent the type graph. Monad transformers save the programmer from the need to manually define their own monads. Without such a mechanism they would be forced to redefine primitive functions like *bind* and *return* each time a new monad was needed.

As mentioned by Filinski [Fil94], the structure created by monad transformers is distinctly hierarchical. In the *X* type above, *IO* is on the bottom, followed by *StateT*, followed by *ReaderT*. This fact is reflected in programs using it, as explicit lifting functions must be used to embed computations expressed in lower monads into the higher structure. For example, the *liftIO* function takes an *IO* action and converts it into an equivalent action in a monad which supports IO:

$$liftIO :: MonadIO\ m \Rightarrow IO\ a \rightarrow m\ a$$

For both XMonad and the DDC type inferencer, the fact that monad transformers produce a layered structure is of no benefit. Actions which supply configuration information, alter the internal state of the program, and interface with the outside world are all commutable with each other. On the other hand, monads which express computational behaviors such as back-tracking and exceptions are not similarly commutable [Fil94].

The XMonad source code of November 2008 includes a binding which renames *liftIO* into the shorter *io*. A hand count by the author yielded 57 separate uses of this lifting function, versus 65 occurrences of the keyword **do**. If it were possible to collapse the monad stack into a single layer then we could avoid this explicit lifting of IO actions. Of course, we would want to achieve this without losing the behavioral information present in their types. The effect typing system we shall discuss in the next chapter does just this.

Interestingly, from the high occurrence of IO lifting functions and the pervasiveness of the *X* type, we see that XMonad is in fact an imperative program. It is imperative in the sense that its processing is well mixed with IO, though not in the sense that it is based around the destructive update of a global store. Although it is written in a "purely functional language", this does not change the fact that the construction of a window manager is an inherently stateful and IO driven problem, with a stateful and IO driven solution.

1.7 Ref types invite large refactorisation exercises

SML obstinately supports destructive update, though its use is restricted to arrays and to data structures that incorporate the special *Ref* type. The following functions are used to program with *Ref*. We use Haskell syntax for consistency.

$$
\begin{array}{ll}
newRef & :: a \rightarrow Ref\ a \\
readRef & :: Ref\ a \rightarrow a \\
writeRef & :: Ref\ a \rightarrow a \rightarrow ()
\end{array}
$$

newRef takes an object and returns a fresh mutable reference to that object. *readRef* takes a reference to an object and returns the object. *writeRef* takes a mutable reference, a new object, and updates the reference to point to the new object.

Although serviceable, tying update to a particular type constructor forces the programmer to decide which parts of their structures should be updatable when their types are defined. On the surface this may seem reasonable, but consider the design of a simple library for a cons-list. We start with the data type:

$$
\begin{array}{l}
\textbf{data}\ List\ a \\
\quad = Nil \\
\quad \mid Cons\ a\ (List\ a)
\end{array}
$$

We would now like to define a set of functions which operate on values of this type. One such function is *index*, which returns the element at a particular position in the list:

$$
\begin{array}{lll}
index :: Int \rightarrow List\ a \rightarrow a & & \\
index\ 0 & (Cons\ x\ xs) & = x \\
index\ n & (Cons\ x\ xs) & = index\ (n-1)\ xs \\
index\ \_ & Nil & = error\ \ldots
\end{array}
$$

Suppose that once we have finished this definition we then want a function *replace* that destructively replaces the element at a certain position in the list. This requires the head of the *Cons* cell to be updatable, so we insert a *Ref* constructor into the data type:

$$
\begin{array}{l}
\textbf{data}\ List\ a \\
\quad = Nil \\
\quad \mid Cons\ (Ref\ a)\ (List\ a)
\end{array}
$$

The definition of *replace* is then:

$$
\begin{array}{ll}
replace\ 0\ e\ (Cons\ rx\ xs) & = writeRef\ rx\ e \\
replace\ n\ e\ (Cons\ rx\ xs) & = replace\ (n-1)\ e\ xs
\end{array}
$$

This is all well and good, but as the *List* type has changed we need to go back and change the definition of *index* to read the element out of the reference before returning it. We must also inspect every other function we've defined that uses the *List* type. If a function accesses the head of a *Cons* cell then it needs a call to *readRef* as well.

$$
\begin{array}{ll}
index :: Int \rightarrow List\,a \rightarrow a & \\
index\ 0\ (Cons\ x\ xs) & = readRef\ x \\
index\ n\ (Cons\ x\ xs) & = index\ (n-1)\ xs \\
index\ \_\ Nil & = error\ \ldots
\end{array}
$$

Conceptually, the operation of *index* hasn't changed at all. *index* still recursively steps through the list until it finds the desired element, then returns it. However, we had to modify its definition because we added a function to the library which requires a certain property (mutability) of the data structure, even though *index* itself doesn't make use of that property. Notice that the modifications required are purely mechanical in nature, and that this problem is very similar to monad creep discussed in the previous section.

Suppose that after defining a few more functions, we desire a new one called *insertAt*. This function will make use of destructive update to insert a new element at a particular position in the list. This requires the *tail* of each *Cons* cell to be mutable as well, so we have to change the data type once again:

$$
\begin{aligned}
&\textbf{data}\ \mathit{List}\ a \\
&\quad = \mathit{Nil} \\
&\quad \mid \mathit{Cons}\ (\mathit{Ref}\ a)\ (\mathit{Ref}\ (\mathit{List}\ a))
\end{aligned}
$$

The definition for *insertAt* is:

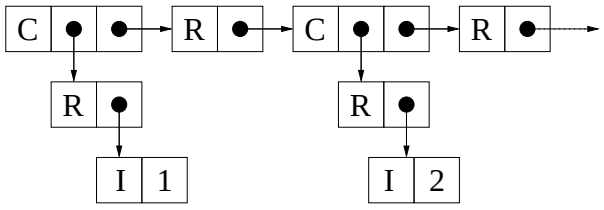
$$
\begin{aligned}
&\mathit{insertAt} :: \mathit{Int} \to a \to \mathit{List}\ a \to () \\
&\mathit{insertAt}\ \_\ e\ \mathit{Nil} \quad = \mathit{error} \ldots \\
&\mathit{insertAt}\ 0\ e\ (\mathit{Cons}\ r\ rxs) \\
&\quad = \quad \textbf{let}\ xs = \mathit{readRef}\ rxs \\
&\qquad\ \ \textbf{in}\ \ \mathit{writeRef}\ rxs\ (\mathit{Cons}\ (\mathit{Ref}\ e)\ (\mathit{Ref}\ xs)) \\
&\mathit{insertAt}\ n\ e\ (\mathit{Cons}\ r\ rxs) \\
&\quad = \quad \textbf{let}\ xs = \mathit{readRef}\ rxs \\
&\qquad\ \ \textbf{in}\ \ \mathit{insertAt}\ (n-1)\ e\ xs
\end{aligned}
$$

Once again, we must go back and inspect every function we have defined so far to make sure that all accesses to the tail of a *Cons* cell first read the reference. Our *index* function is now:

$$
\begin{aligned}
&\mathit{index} :: \mathit{Int} \to \mathit{List}\ a \to a \\
&\mathit{index}\ 0\ (\mathit{Cons}\ x\ xs) \quad = \mathit{readRef}\ x \\
&\mathit{index}\ n\ (\mathit{Cons}\ x\ xs) \quad = \mathit{index}\ (n-1)\ (\mathit{readRef}\ xs) \\
&\mathit{index}\ \_\ \mathit{Nil} \qquad\qquad = \mathit{error}\ \ldots
\end{aligned}
$$

More mechanical modifications have wasted more programming time. What can be done to alleviate this problem? The central activity of programming is defining data structures and writing functions which operate on them. Unless a programmer is simply replicating a program they have written before then they are unlikely to know exactly which parts of their structure should be wrapped in *Ref* and which can be left bare.

If we define all structures to be mutable from the start then we can avoid having to re-inspect existing functions as the data type evolves, though this would require many superfluous calls to *readRef*. In addition, a naive implementation of *Ref* would simply insert reference objects into the run-time data structure, so we would pay a performance penalty as well:



On the other hand, if we define our data types *without* using *Ref*, then structures of that type can not be updated — ever. If those structures are provided by a library and a client programmer decides they actually *do* want to perform an update, then it is likely that the only practical solution will be to define their own types and write code that duplicates functionality already present in the original library.

This is exactly the case with SML lists and tuples, which are all immutable. Although some code duplication can be alleviated by using similar module signatures for mutable and immutable objects, the fact that the two have fundamentally different types only serves to encourage it. If only the immutable versions are provided in base libraries, then users are encouraged to use these structures in cases where a mutable one would be more appropriate. This in turn relegates mutable structures to be second class citizens of the language.

1.8 Practical limitations of lazy evaluation

The following example from [GS01] demonstrates the subtlety of space usage in lazy languages:

$$\begin{array}{lll} \textbf{let} & xs & = [1..n] \\ & x & = head\ \ xs \\ & y & = last\ \ xs \\ \textbf{in} & x + y & \end{array}$$

We will use GHC as a reference point for the behavior of a real implementation. When compiled with no optimisations, the execution of this program will create a thunk for each let-binding before evaluating the addition [PJ92]. If we assume that addition demands its arguments left to right, the thunk for x will be forced first, resulting in the value 1. This thunk will then be overwritten with its value, which eliminates the contained reference to xs. The evaluation of y entails the construction and traversal of the list $[1..n]$ until we reach its last element. In a garbage collected implementation this can be done in constant space because *last* does not hold a reference to a list cell once it has traversed it.

However, if we change the order of arguments to addition the program consumes space proportional to the length of the list:

$$\begin{array}{lll} \textbf{let} & xs & = [1..n] \\ & x & = head\ \ xs \\ & y & = last\ \ xs \\ \textbf{in} & y + x & \end{array}$$

In this case, the evaluation of y entails the construction of the entire list. The list cannot be garbage collected until the thunk for $head\ \ xs$ has been forced, because it contains a reference to its first element. This example shows that only slight modifications to a program can result in dramatic differences in space usage.

All strictness analyses are incomplete

The run time performance of many lazy programs can be improved by exploiting the *strictness* properties of functions. A function f is *strict* if and only if

$f \perp \equiv \perp$ [PJ87]. This can arise for three reasons. If f inspects the value of its argument when it evaluates, then it will diverge if its argument does. If f always returns its argument uninspected, then the result will be $\perp$ if the argument is. Finally, f may always diverge, independent of the argument value. If none of these cases apply then function is *non-strict*.

For example, the *choose* function is strict in its first argument but not the others:

$$choose\ \ b\ x\ y = \textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y$$

When this function is applied to its three arguments, it will always require the value of b. On the other hand, either x or y may be returned, but not both. A similar example is the *first* function which returns just its first argument while discarding the second:

$$first\ \ x\ y = x$$

As x is passed through to the result, it is strict in this argument. As the function body makes no reference to y, it is non-strict in that one. Strictness analysis [BHA85, WH87, SR95] is used to recover the strictness properties of functions. A compiler can use this information to convert a call-by-need program into a more call-by-value version without changing its meaning. For an implementation such as GHC, this amounts to identifying the let-bound variables which are passed to strict functions, and evaluating those bindings as soon as they are encountered, instead of building thunks.

For many lazy programs, especially those performing lots of numeric computation, evaluating strict bindings early can result in substantial performance improvements. Early evaluation saves the allocation and initialisation of thunks, as well the need to update and garbage collect them once their values are demanded.

In practice, a compiler should reduce strict bindings to weak head normal form (whnf) [PJ87] only. Reduction to whnf eliminates outer redexes while allowing thunks to be present deep within data structures, such as at the tail position of lists. To see why, consider our first example again:

$$\begin{array}{lll} \textbf{let} & xs & = [1..n] \\ & x & = head\ \ xs \\ & y & = last\ \ xs \\ \textbf{in} & x + y & \end{array}$$

The fact that addition and *head* are strict in their arguments implies that the xs, x, and y bindings can be evaluated as soon as they are encountered. If we evaluate $[1..n]$ to whnf we construct just the outer *Cons* cell and the program runs in constant space. However, if we were to fully evaluate $[1..n]$ before taking its head, then the program will consume space proportional to this list.

Like all compile time analyses, strictness analysis is necessarily incomplete. This is plainly obvious from our *choose* example:

$$choose\ \ x\ y\ z = \textbf{if}\ x\ \textbf{then}\ y\ \textbf{else}\ z$$

Suppose we write an expression which uses *choose* to print one of two results:

$$putStr\ (choose\ \ b\ \ exp1\ \ \ exp2)$$

putStr is strict in its argument, yet the question of whether it prints *exp1* or *exp2* can only be answered by knowing the value of b. In general, the value of b cannot be determined at compile time. Apart from bumping up against the halting problem, this fact is obvious if we consider a situation where b is derived from user input.

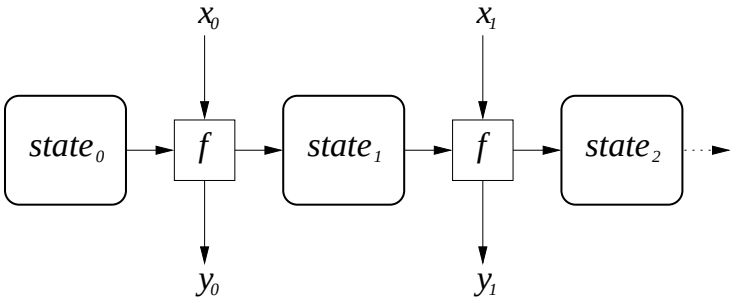
In a typical Haskell program, many functions are concerned with processing algebraic data. Such functions are usually written with pattern matching, or with a case-expression that examines the outer constructor of the input value. case-expressions are a generalisation of if-expressions, so we have the same problem as with the *choose* example above. In general, for a particular function call we can not know what the outer constructor of its argument will be, which defeats strictness analysis in a similar manner.

Space leaks can be elegantly created with *mapAccumL*

In a lazy functional program, a *space leak* is created when unevaluated thunks reference a large amount of data, preventing it from being reclaimed by the garbage collector. This can be counter intuitive at first glance. How can an *unevaluated* expression use more space than its actual result? Consider the expression (*range* 1 100) which builds the list [1..100]. We would expect a thunk representing the application of *range* to its arguments 1 and 100 to use much less space than a fully evaluated list of 100 elements.

However, consider the case where one of the arguments is a variable instead of an integer value. A thunk which represents the application (*range* 1 n) contains a reference to the object n, and as long as the thunk is live this object cannot be garbage collected. Suppose n is also a thunk, and that it references a large amount of data. While our original list remains unevaluated, this data remains live. It may be that the program's performance would be improved by forcing the list to be fully evaluated as soon as possible. This would allow the garbage collector to reclaim space used by thunks and objects that are no longer referenced. Of course, whether this would work in practice is very application specific. Factors to consider include the size of the resulting list versus the size of the retained data, whether the entire list value will actually be used by the program, whether the live data is also held live by other expressions, cache and main memory sizes, and so on.

Space leaks are especially common in lazy programs which are based around state and state transformers. For these programs, execution is divided into a sequence of steps, with a well-defined state before and after each step. The function f takes the old state, some input x and produces the next state and some output y:



In Haskell, this pattern of computation is expressed by the function *mapAccumL* which has type:

$$mapAccumL :: (state \to x \to (state, y)) \to state \to [x] \to (state, [y])$$

mapAccumL takes a transition function, an initial state, a list of inputs and produces the final state and the list of outputs. Many programs use a similar pattern of computation, though not all express it with *mapAccumL*. Consider an interactive program such as a computer game. We could imagine that *state* is a description of the game world, x is the user input, y is a description of the user display, and f is the game logic which computes a new state and display based on the input.

For a computer game, the state could consist of the player's position, surrounding terrain, current enemy positions, remaining ammunition, and so on. A space leak is created when the program fails to demand the entire y value after each step. Suppose that y includes the player's score at each step of the game, but this information is not displayed in real time. Although the score at each step might be expressed by a single integer, as it depends on the current game state at least past of this structure must be kept live until the integer is fully evaluated. If a user plays the game for an hour, with a new state generated 30 times a second, then this can equate to a substantial amount of retained data. Additionally, when the score is a non-trivial function of the current state, reasoning about the *amount* of space wasted becomes intractable.

In Haskell, the only practical way to deal with a complex leak is to write so called *deepseq* functions that manually traverse over an entire structure to ensure it is fully evaluated. Other techniques can help, such as having the garbage collector perform leak avoiding projections [Wad87], but to fully cure leaks the programmer must ensure that all structures which *should* be evaluated actually are. Most *deepseq* functions are written to eliminate all redexes in a structure, and are therefore equivalent to the *reduce to normal form* strategy from [THLPJ98]. A built-in *deepseq* function was proposed for Haskell', the successor standard to Haskell 98 [Has08], but as of November 2008 it has not been implemented in GHC.

It is also possible to add strictness annotations to user defined data types. These annotations prevent thunks being created at certain positions in the structure, but cannot be easily be added to library defined types such as *Map* and *List*.

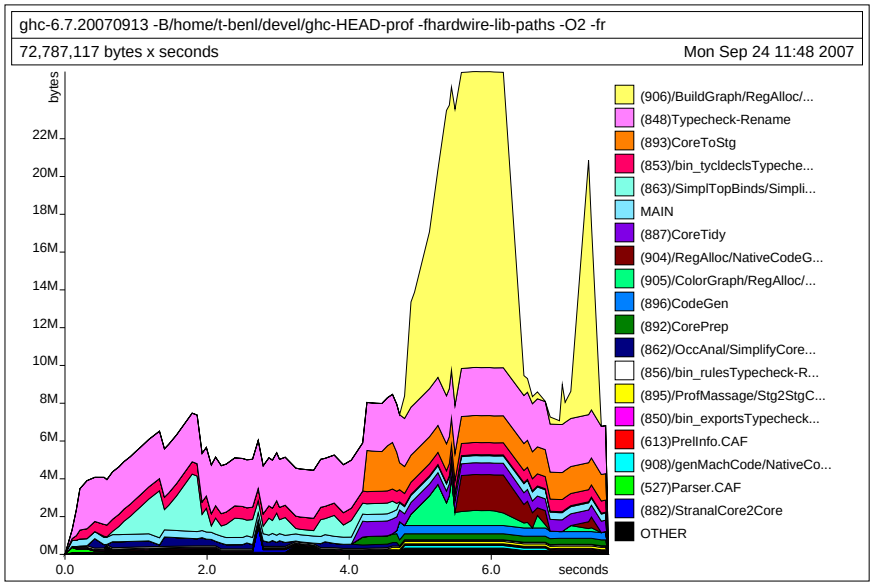
Case study of a space leak

A state based space leak was encountered while the author was developing a graph coloring register allocator [Cha04, SRH04] for GHC 6.7. The algorithm is based around a graph where each node represents a program variable. An edge between two nodes represents a constraint that those two variables can not be assigned to the same register. The goal is to assign registers, visualised as colors, to each of the nodes in a way that satisfies the constraints, whilst using only the available set of registers. The algorithm proceeds by extracting a constraint graph from the code undergoing register allocation, and then attempting to color it. If this is not possible with the available colors (registers) then the algorithm modifies the code to store some variables on the stack instead of registers, and tries again. For non-pathological programs this process should converge within three or four iterations.

Graph coloring register allocation is a state based algorithm. The state consists of the current version of the code undergoing allocation, along with the constraint graph. As opposed to the *mapAccumL* function, there is no extra per-step input to the state transition function corresponding to the x values in the previous diagram. The output y values correspond to graph profiling information, such as the number of colored versus uncolored nodes remaining after each step.

When the allocator was being developed we were well aware of space leaks and their causes. The intended operation of the algorithm was to build a complete constraint graph, attempt to color it, and if that failed to build a new graph and leave the old one for the garbage collector. We knew that if the program retained any references to the profiling information for old graphs, then this would prevent those graphs from being garbage collected. If this happened we would have a space leak, so we made considerable effort *not* to retain profiling information unless it was explicitly requested. We reasoned that if the user requested profiling information then they would not mind if the allocator ran a little slower due to retained data, as this was not a common operation.

However, once it was written, an examination of heap space [SPJ95] used by the allocator revealed the following:

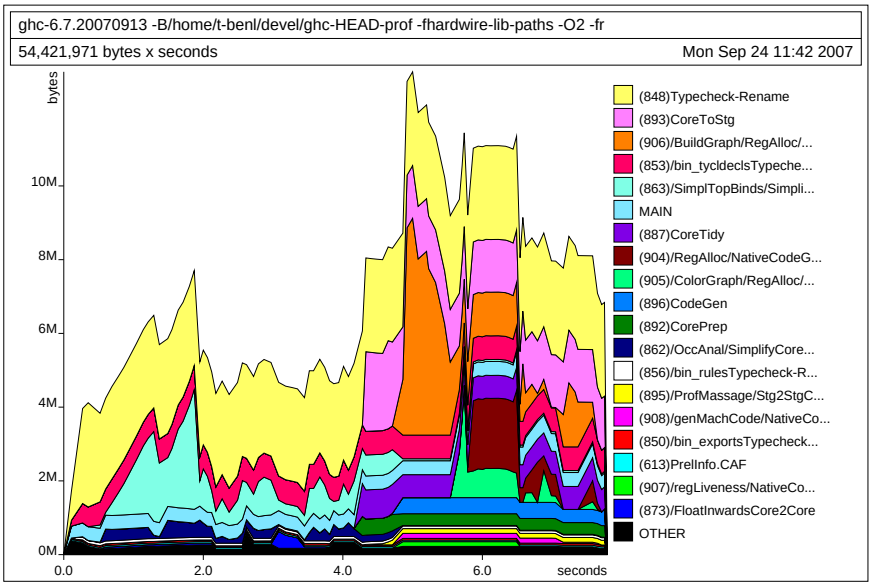


The two large spikes in space usage that appear around 6 and 7 seven seconds are directly attributable to the register allocator. This is when performing allocation for the `SHA1.hs` module from the darcs 1.0.8 source code. Object type profiling revealed that most of the space was taken up by thunks representing function applications.

As to the exact cause of the leak, we are not sure. We could imagine that when the compiler emits a particular compiled machine instruction, this action demands the result of register allocation for that instruction. The registers allocated to a particular instruction depend on what other registers are assigned to surrounding instructions. We could then imagine a section of graph in the final state of the allocator being demanded. This in turn might demand larger sections of graph from previous states, along with parts of the various intermediate versions of assembly code that we tried to find allocation solutions for.

Good research has been done on formally analysing the space usage of call-by-need programs [GS01, BR00]. However, trying to reason about the exact space behavior of a three thousand line program, compiled with a production compiler that incorporates tens, if not hundreds of individual optimisations is another matter entirely. We plainly admit that our reasoning is little but inspired guess work.

What we do know is that using a *deepseq* function to force the graph to be fully constructed before coloring cured the worst of the problem. This result was obtained through experimentation and frequent consultation with the heap usage profile. The following profile is for the final version:



In this version the large spikes in space usage have been reduced, resulting in a peak usage around half of the unforced version. We conjecture that the remaining cost attributed to `(906)/BuildGraph/RegAlloc` is mostly due to the legitimate construction of the graph during allocation, though once again we can not be sure. We deemed the profile acceptable, and moved on to other things.

We glean several points from this experience. Firstly, although a programmer may write what they feel is a state based program, if it is expressed in a lazy language then it may not behave that way at runtime. Secondly, the exact cause of space leaks in large lazy programs can be very hard to reason about. That being said, although the *problem* may be hard to characterise, the solution is well understood. Forcing thunks to values eliminates their contained references and frees up objects for the garbage collector.

On a philosophical note, we feel there is an immense practical difference between optimisation and control. Having a large number of optimisations in a compiler is all well and good, but if the compiled code *still* doesn't run fast enough then the language (and/or compiler) must provide enough additional control for the programmer to step in and fix the problem. If this is not possible then the programmer will be forced to use a different language, and if that happens more than once then they will be unlikely to choose the same system for their next project.

In this case we were able to fix the problem. However, we do note the irony of writing extra code to manually force the evaluation of expressions that we did not intend to be suspended in the first place. We cannot, off hand, think of a single place in the register allocator code where lazy evaluation was used for a useful purpose.

We are not suggesting that laziness is never useful, more that it depends on the application. For a selection of programs from the nofib benchmark suite [Par92], Harris [HS07] gives the percentage of allocated thunks that were actually evaluated at run-time. In the 20 programs considered, 9 ended up evaluating at least 99% of their thunks, 14 evaluated at least 90%, while only one evaluated less than 80%.

The fact that a program evaluates almost all of its thunks does not imply it does not make use of laziness. For example, if we use laziness to evaluate the expression *sum* [0..100] in constant space, then all the thunks in the list will be forced. The application of *sum* to a *Cons* cell demands the element value as well as the tail of the cell. However, the fact that a program evaluates 99% of its thunks would suggest that it is not creating lazy lists where the spine is evaluated but the majority of the elements are not. It would also suggest that the program is not using the "sexier" lazy structures, such as infinite game trees [Hug89].

1.9 A way forward

Disciple allows destructive update and lazy evaluation to be present in the same language. We do this while preserving the overall structure of types, and while keeping most of the nice algebraic properties associated with purely functional languages. By preserving the structure of types we hope to avoid the refactoring exercises discussed earlier. We do not rule out support for state monads or *Ref* types, rather we desire a system which does not *require* them to write most programs.

We use a type based mutability and effect analysis. The analysis determines which objects in the program might be destructively updated, and which are guaranteed to remain constant. Using this information, the compiler can perform optimisations on the pure parts of the program while leaving expressions with interfering computational effects in their original order. This strategy is similar to that taken by Tolmach [Tol98] though we use a System-F [Rey74] style core language instead of a monadic one. The core language uses a witness passing mechanism to manage mutability and effect constraints, similar to that used to manage type equality constraints in System-Fc [SCPJD07]. Although the extra mutability and effect information is visible in source types, it can usually be elided by the programmer, and is therefore not an undue burden.

The default evaluation method is call-by-value. This makes it easier to construct an efficient implementation on current hardware, as well as eliminating an important source of space leaks. The programmer may manually suspend function applications when desired, and the runtime system will automatically force them as needed. This is unlike library based implementations of laziness in languages such as O'Caml. These implementations require the use of explicit forcing functions, as well as changing the types of lazy values.

We also use our analysis to detect when an object is guaranteed *not* to be a thunk. Our implementation of lazy evaluation is likely to be slower than a natively lazy system such as GHC. However, non-lazy code should not suffer a substantial performance penalty compared with other default strict languages such as ML and C.

Our type system uses a type class based mechanism to attach purity, mutability, constancy, laziness and other constraints to data types. This allows functions to be polymorphic in those attributes and not require the overall structure of types to be changed. We also use this mechanism to detect when the combination of laziness and destructive update in the same program might give a result different to the call-by-value case. We flag these as type errors and assert that our language is still purely functional by Sabry's definition [Sab98]. Our language includes support for record types, and we use type directed field projections which permit field names to be in record-local scope.

We would like for it to be possible *and* practical to write efficient programs in our language. Finally, we would like it to be attractive to people who don't actually care that much about the philosophy of functional programming. We follow Steele and Sussman [SS76], and Knuth [Knu74] in that a language designer's emphasis should be on helping the programmer solve their particular problems. Our aim is not to separate language features into "good" and "bad", only to offer sharp tools in a hope they will be useful.

Chapter 2

Type System

We have seen that ML style references are clumsy to work with because their use changes the structure of types. This causes mutable values to be type-incompatible with constant values, and invites large re-factorisation efforts when writing code. Could there be a better way? In a traditional imperative language such as C, Pascal or Java, the programmer is free to update data as they see fit, and the type of mutable data is not required to be structurally different from that which remains constant. However, if we were to allow the programmer to update any object in the system, without tracking it carefully, then we would have to assume that *every* object was mutable. This would dramatically limit our ability to perform optimisations on the intermediate code. With this in mind, we first consider the form of update we wish to support, and then seek a way of tracking which objects it is applied to.

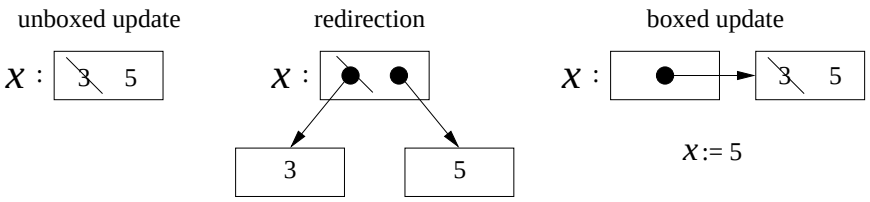
We intend this chapter to serve as a gentle introduction to our type system, and to the concepts involved. For this reason we have refrained from starting with a formal description of our language or its typing rules. This information is given in §3.2.

2.1 Update without imperative variables

In a typical imperative language the syntax to bind a variable is the same as that used to update it. This conflates the two issues. Consider the semantics of the following program fragment:

$$x = 5$$

Readers with a more functional background would likely read this as "x has value 5". Readers with a more imperative background could equally reply "x is being updated to value 5." With the difference between boxed and unboxed integers in mind, the fragment could also mean "x is a pointer to an object, and it is the pointer which is being updated" or perhaps "x points to an object, and it is the *object* which is being updated". These three readings for update are shown in the following diagram, where the value is being updated from 3 to 5.



In the first two cases it is the local value of x that is being changed. In these cases we call x an *imperative variable*, and we do not support this form of update in our language. However, in the last case only the object pointed to by x changes. We support this option and write $x := 5$ to distinguish it from the syntax for *binding* which is simply $x = 5$.

Why do it this way? Firstly, we intend to support update by the inclusion of functions such as:

$$\begin{array}{lll} \mathit{updateInt} & :: & \mathit{Int} \rightarrow \mathit{Int} \rightarrow () \\ \mathit{updateChar} & :: & \mathit{Char} \rightarrow \mathit{Char} \rightarrow () \end{array}$$

The first parameter of these functions is the object to be updated, and the second is the source of the new value. We use (:=) as an overloaded update operator, so $x := 5$ can be rewritten as $\mathit{updateInt}\ x\ 5$.

In light of this, we restrict update to objects for two main reasons. The first is that in our implementation we use local unboxing [Ler97] to support efficient numeric computation, and we desire intermediate results to be held in the register set wherever possible. If we permit local values to be updated, then we would also want to pass them by reference, so that called functions could update them via this reference.

Consider then an unboxed version of *updateInt*, and some C code that uses it:

```
void trouble(void)
{
    int x, y;

    ...
    x = 3;
    y = 5;
    updateIntUnboxed (&x, y);
    ...
}
```

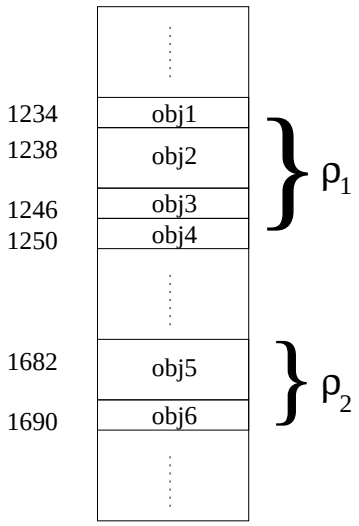
This code is valid, though deeply troubling to a C compiler. As x is passed by reference, its value *must* be held on the stack instead of in the register set, otherwise we couldn't construct a pointer to it. More seriously, in regards to separate compilation, a C compiler would be unable to guarantee that this pointer becomes unreachable before `trouble` returns, losing the stack frame and the storage for `x`. As in [Hen02] we have observed GCC to disable a number of low-level optimisations when compiling code which uses pointers to automatic variables. We could perhaps implement local update as a primitive of the language, but we avoid this option due to the extra complexity and conceptual mismatch relative to object update.

Another reason for not supporting imperative variables, and perhaps a more convincing one for readers who don't spend all their time writing compilers, is that it simplifies the type system. If we only support update of the objects *pointed to* by our variables then we only need to reason about the mutability of these objects, and not of the variables as well. This can be contrasted with [Ode91] and [SSD08b] which reason about both.

2.2 Region Typing

2.2.1 Regions and aliasing

A region is the set of store locations where a particular object may be present at runtime. We use regions to reason about the mutability and aliasing of objects. The following diagram shows a store containing a number of objects, divided into two regions. This diagram is intended to be suggestive only. Many systems besides our own make use of regions, and a particular system may allow them to be disjoint areas of the store, include free space, grow with time, be hierarchical, include only sub-components of an object, and so on.



We use ρ_n to denote *region handles*. A region handle can be thought of as a runtime identifier for a particular region, or perhaps an index into a table that describes the extent of a region. For the simple system in the diagram, we could treat a region as a set of aligned, 4-byte words. In this case our region handles could be defined as:

$$\begin{aligned} \rho_1 &= \{1234, 1238, 1246, 1250\} \\ \rho_2 &= \{1682, 1690\} \end{aligned}$$

At compile time we will not necessarily know how the objects in the store will be arranged, or how to define the region handles. We would like to write functions that operate on objects from any region, independently of how they are arranged. For this purpose we introduce *region variables*, which we use to bind region handles. Region variables are identified as r_n in this text.

There are conceptual similarities between region and value information. Consider the following statements of value:

$$\begin{aligned} 23 &\mapsto < ...10010... > \\ a &= 23 \end{aligned}$$

In the first statement, the numeral 23 represents an object in the store that includes a particular bit string. In the second statement we have used a *value variable* to bind the numeral 23. We can think of regions as being akin to the objects in the store, region handles being akin to numerals, and region variables being like value variables. In this sense, regions are physical things,

region handles are descriptions of them, and region variables are place holders for the handles.

Clearly though, regions and values are different kinds of things. As per tradition we use a star * to denote the kind of value types. Region kinds are denoted by a percent sign %[1]. In the concrete syntax we also use % as a namespace qualifier, writing `%rn` in place of r_n. This helps the parser, as well as being convenient for syntax highlighting text editors.

Unlike the system of Tofte and Birkedal [TB98], ours deals only with region variables and not with the definition of region handles, or the layout of the store. Their system uses regions to manage the allocation of free space at run time, where ours uses regions as part of an analysis to guide compile time code optimisations.

Our analysis is type based. We add region variables to all type constructors which correspond to data objects that can be usefully updated. This includes constructors like *Int*, *Bool* and *List*, but not the function constructor ($\rightarrow$) or the unit constructor (). The function constructor does not need one because the value of a function cannot be updated at runtime. The unit constructor does not need one because there is only one possible unit value.

For example, a list of character pairs could have type:

$$\begin{aligned} &\mathit{pairs} \quad :: \mathit{List}\ r_1\ (\mathit{Pair}\ r_2\ (\mathit{Char}\ r_3)\ (\mathit{Char}\ r_4)) \\ &\mathit{pairs} \quad = [\mathit{MkPair}\ \texttt{'g'}\ \texttt{'o'},\ \mathit{MkPair}\ \texttt{'b'}\ \texttt{'y'},\ \ldots] \end{aligned}$$

In a top level signature such as this, if two type constructors have different region variables, such as $\mathit{Char}\ r_3$ and $\mathit{Char}\ r_4$, then the corresponding values are guaranteed to be represented by different run-time objects. However, in general these objects may *alias*. For example, here is a function which creates a pair of integers:

$$\begin{aligned} &\mathit{intPair} :: \forall r_1\ r_2\ r_3.\ \mathit{Int}\ r_1 \rightarrow \mathit{Int}\ r_2 \rightarrow \mathit{Pair}\ r_3\ (\mathit{Int}\ r_1)\ (\mathit{Int}\ r_2) \\ &\mathit{intPair}\ x\ y\ =\ \mathit{MkPair}\ x\ y \end{aligned}$$

As the region variables r_1, r_2 and r_3 are quantified in the type signature, we may pass the same integer object for both arguments of the function:

$$\begin{aligned} &\mathit{five} \quad :: \mathit{Int}\ r_5 \\ &\mathit{five} \quad = 5 \end{aligned}$$

$$\begin{aligned} &\mathit{pairOfFives} \quad :: \mathit{Pair}\ r_4\ (\mathit{Int}\ r_5)\ (\mathit{Int}\ r_5) \\ &\mathit{pairOfFives} \quad = \mathit{intPair}\ \mathit{five}\ \mathit{five} \end{aligned}$$

Here, the region variables r_1 and r_2 of *intPair* have been instantiated to r_5. This tells us that both elements of the pair may refer to the same heap object, and they will in this case. Note that in the body of *intPair*, the value variables x and y may also refer to the same object because we can pass the same one for both arguments.

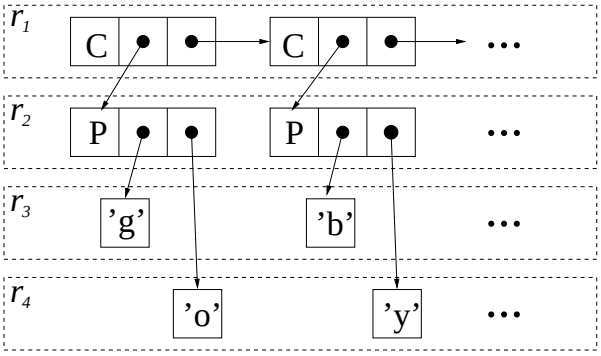
In the type of *pairOfFives*, the region variable r_4 is fresh because the evaluation of *intPair* will allocate a new object to represent the pair. Freshly allocated objects do not alias any existing objects.

Aliasing information is of fundamental importance when reasoning about destructive update, as any read or write actions performed on objects in one region

[1] Pictorially, % is two circles separated by a line, a mnemonic for "this, or that"

will not be visible to the parts of a program that only deal with another. To use the language of [Rey78], actions performed on disjoint regions do not *interfere*.

In many cases the region variables attached to differently named type constructors will be distinct, but this is not required in general. In our *pairs* example, all the list constructor cells are in one region, and all the pair cells are in another:



Setting $r_1 = r_2$ would be equivalent to placing the list cells in the same region as the pair cells.

Like Talpin and Jouvelot's original work [TJ92b], our concept of a region is simply a name for a set of locations. We sometimes find it useful to visualise regions as colours of paint, which we apply to data objects stored in the heap. Setting $r_1 = r_2$ corresponds to painting all the list and pair cells the same colour. They will be harder to distinguish afterwards, corresponding to a weakening of our analysis, but it will cause them no harm.

As region variables are provided as parameters to type constructors, the kinds of the constructors reflect this. *Char* takes a region and produces a type. *List* takes a region, a type, and produces a new type. *Pair* takes a region, two types, and produces a new type:

$$\begin{array}{ll} \mathit{Char} & :: \% \rightarrow * \\ \mathit{List} & :: \% \rightarrow * \rightarrow * \\ \mathit{Pair} & :: \% \rightarrow * \rightarrow * \rightarrow * \end{array}$$

2.2.2 Region classes

When a value is mutable we add mutability constraints to the region variables in its type. For example, if we wanted to update the characters in a string we would give it type:

$$\mathit{str} :: \mathit{Mutable}\ \ r_2 \Rightarrow \mathit{List}\ \ r_1\ (\mathit{Char}\ \ r_2)$$

The constraint $\mathit{Mutable}\ \ r_2$ is a *region class*. Region classes are similar to the value type classes in Haskell [HHPJW96], such as *Show* and *Eq*. With value type classes, the type constraint $\mathit{Eq}\ a$ requires a to be a type that supports equality. Similarly, the region constraint $\mathit{Mutable}\ \ r_2$ requires r_2 to correspond with a region that supports update.

When discussing our system we use the word "type" to refer to all the information in a signature, including value type information such as *List* and *Char*,

any constraints on variables, region information, as well as the effect and closure information we will discuss later. For this reason we also refer to both region classes and value type classes as simply "type classes". Note that the programmer usually doesn't have to provide this additional information in type signatures. Most can be reconstructed by the type inferencer. This is discussed further in §2.7.2 and §3.4.7.

Returning to the signature of *str*, we call term on the right of the ⇒, the *body* of the type. We call the value portion of the body is its *shape*, because this information describes the overall structure of the object in the store.

As our types often contain a large number of constraints, we usually write them after the body, instead of before it as in Haskell:

$$\begin{array}{lll} str & :: & List\ r_1\ (Char\ r_2) \\ & \triangleright & Mutable\ r_2 \end{array}$$

The ▷ is pronounced "with", and is written as `:-` in the concrete syntax. The difference between the above type and the original prefix form is purely syntactic, and our compiler accepts both.

In the above type, no constraint has been placed on r_1. If we wish to update the spine of the list as well as its characters, then this region must also be mutable. Multiple constraints are separated by commas:

$$\begin{array}{lll} str & :: & List\ r_1\ (Char\ r_2) \\ & \triangleright & Mutable\ r_1 \\ & , & Mutable\ r_2 \end{array}$$

Being able to update the spine of a list is useful for operations such as inserting a new element into the middle of the list, as it allows us to change the tail pointers of existing cons cells.

On the other hand, if we wish to *prevent* updates to the spine we could use the constraint $Const\ r_1$ to enforce this:

$$\begin{array}{lll} str & :: & List\ r_1\ (Char\ r_2) \\ & \triangleright & Const\ r_1 \\ & , & Mutable\ r_2 \end{array}$$

As there are two region variables in this type, both the spine and elements can have differing mutabilities. Attempting to constrain a region variable to be both *Mutable* and *Const* results in a compile time type error.

2.2.3 Functions, allocation and non-material regions

In our system the successor function has the following signature:

$$succ :: \forall(r_1 :: \%)\ (r_2 :: \%).\ Int\ r_1 \rightarrow Int\ r_2$$

In this type we have included the kind of each region variable, but as in Haskell we can omit this information if it can be easily inferred. The variables r_1 and r_2 must have region kind because they are used as parameters to the *Int* constructor, so we instead write:

$$succ :: \forall r_1\ r_2.\ Int\ r_1 \rightarrow Int\ r_2$$

Starting with the *Int* r_1 term on the left of the arrow, the fact that r_1 is quantified indicates that *succ* can operate on values from any region. On the right of the arrow, the fact that r_2 is quantified indicates that *succ* can produce its output *into* any region. This is possible because the function allocates a new *Int* object each time it is called, and freshly allocated objects do not alias existing objects. Alternatively, if a function does not allocate its return value, then the region variables in its return type will not be quantified:

$$
\begin{aligned}
&x :: \mathit{Int}\ \ r_3 \\
&x = 5 \\
&\\
&\mathit{sameX} :: () \to \mathit{Int}\ \ r_3 \\
&\mathit{sameX}\ () = x
\end{aligned}
$$

In this example, *sameX* returns the same object every time it is called. This object comes from its environment, and is shared between all calls to it, hence r_3 must remain unquantified. Unquantified region variables can also appear on the left of an arrow. This happens when a function conflates its arguments with values from the environment:

$$
\begin{aligned}
&y :: \mathit{Int}\ \ r_4 \\
&y = 23 \\
&\\
&\mathit{chooseY} :: \mathit{Int}\ \ r_4 \to \mathit{Int}\ \ r_4 \\
&\mathit{chooseY}\ z = \textbf{if}\ \ldots\ \textbf{then}\ y\ \textbf{else}\ z
\end{aligned}
$$

The object returned by *chooseY* could be either its argument z, or the shared object y. Our system cannot represent the fact that the returned object might be in one region *or* another, so we use the same variable for both. This limitation is discussed in §5.2.2. In this example, r_4 is also present in the environment, so it cannot be quantified in the type of *chooseY*.

Although r_4 appears in the types of both y and *chooseY*, each occurrence has a slightly different meaning. In the type of y, it represents a particular set of locations in the heap, and one of those locations contains the integer object of value 23. On the other hand, the use of r_4 in the type of *chooseY* does not mean that *chooseY* also contains an integer object. Instead, these occurrences represent locations in the store where the function's argument and return values lie. We distinguish between these two cases by saying that r_4 in the type of y is in a *material* position, whereas in the type of *chooseY* is not. The difference between the material and immaterial positions of type constructors is discussed more fully in §2.5.4.

2.2.4 Updating data requires it to be mutable

When a function can update its argument, we add a constraint to its type that requires the argument to be mutable. For example, the *inc* function destructively increments its argument and has type:

$$
\begin{aligned}
\mathit{inc}\ \ &:: \quad \forall r_1.\ \mathit{Int}\ \ r_1 \to () \\
&\rhd \quad \mathit{Mutable}\ \ r_1
\end{aligned}
$$

This type indicates that *inc* can operate on integers in any region, as long as that region is mutable. We treat mutability as a capability provided by the objects in our system, and the requirement for this capability flows from the functions that make use of it. An alternative setup would be to explicitly *permit*

update by requiring the programmer to supply type signatures and mutability constraints for every object that is to be updated, or to use a special keyword when allocating them. We feel that the use of a special keyword would create clutter in the program, though we will sometimes require mutability constraints to be given in a type signature. We will return to this in §3.4.7.

During type inference, the compiler compares all the constraints placed on the region variables in the program. In the absence of an explicit type signature, if a particular region is not constrained to be mutable, then at runtime the objects contained within that region will never be passed to a function that can update them. For this reason, material region variables that are not constrained to be mutable are considered to be constant.

This does not apply to quantified, immaterial regions in the types of functions. In this case the three options: mutable, constant, and unconstrained, have distinct meanings. For example, in the following type signature:

$$foo :: \forall r_1.\ Int\ \ r_1 \rightarrow ()$$

As r_1 is unconstrained we may apply this function to integers which are either mutable or constant, whereas with:

$$\begin{array}{lll} foo' & :: & \forall r_1.\ Int\ \ r_1 \rightarrow () \\ & \triangleright & Const\ \ r_1 \end{array}$$

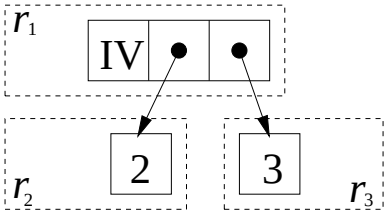
We can only apply this function to integers which are constant.

2.2.5 Primary regions and algebraic data

In all examples so far, the type constructors used have had only one region parameter. This is typical for simple types with a small amount of internal structure, but we need more when defining algebraic data. Consider a vector of two integers:

$$\begin{array}{l} \textbf{data}\ \ IntVector\ \ r_1\ r_2\ r_3 \\ \quad = IV\ \ (Int\ \ r_2)\ (Int\ \ r_3) \end{array}$$

The first region variable r_1 corresponds to the region containing the outer IV constructor. This is called the *primary region variable*. The variables r_2 and r_3 appear in the body of the definition and represent the regions containing the integer components. For example, the value $(IV\ \ 2\ 3)$ would be represented as:



These three separate region variables provide three degrees of freedom when deciding which parts of the object should be mutable and which should be constant. Allowing r_2 and/or r_3 to be mutable permits the components to be updated, and when r_1 is mutable we can update the pointers in the outer constructor. The *tag* of the outer constructor is also contained in the primary

region. Updates to the tag permit the value of enumerations such as *Bool* to be changed.

Note that with this system it is not possible to give the *pointers* to the two components separate region variables. We omit this capability to reduce the complexity of the system, though we see no fundamental barrier to supporting it if required in the future.

2.2.6 Thinking about regions

There are several ways to conceptualise what a region actually "is", and we have mentioned two so far. Firstly, a region is an area of the heap where objects can be stored at runtime. For systems that use regions to manage allocation and deallocation [TBE+06], this is the most natural. Fresh objects are allocated into a particular region, and the whole region is reclaimed by the storage manager when the objects contained are no longer needed by the running program. Such systems use region allocation to augment or replace the standard garbage collection mechanism. At an operational level, the regions in such systems are usually contiguous, or are constructed with a linked list of contiguous memory pages. However, DDC does not use regions to manage allocation, it relies on a traditional garbage collector. We can still imagine a region to be a specific area of the heap, but the parts of the heap that make up the region are scattered throughout, and do not form a contiguous block.

Secondly, a region is a label for a collection of objects in the store. Earlier we suggested imagining these labels to be like colours of paint. When a program is compiled, the compiler decides on a fixed set of colours (regions). It then pretends that at runtime, every object allocated by the program will be painted with one of these colours. The colours help it to reason about what the program is permitted to do with a certain object. For example, we could paint all the mutable objects with shades of pink, and all the constant objects with shades of blue. Importantly though, the colours are just pretend. Our analysis is static, so we do not record what region an object belongs to in the object itself, or maintain any region information at runtime.

Finally, a region is a label for a set of program points which perform allocation [Pie05]. If we know that a particular object is in region r_1, then it must have been allocated by one of the points corresponding to r_1. Every object is allocated by one program point, and an allocation point can allocate zero or more objects. Allocation points exist within functions, so whether or not an allocation point ever allocates depends on whether the function is ever called. However, during evaluation the objects tend to get mixed up, such as when choosing between two objects in an if-expression. This means that the compiler will usually lose track of the exact point where a particular object was allocated. It can only hope to reduce it to a small set of possibilities. Using this idea we can imagine that if a region variable is constrained to be *Const*, some part of the program requires an allocation point to produce a constant object. Likewise, if a region variable is constrained to be *Mutable*, some part of the program requires a mutable object. A mutability conflict arises when a particular allocation point must produce an object that is both mutable and constant. This is not possible, so we report an error. We will exploit this line of reasoning further when we come to prove the soundness of our core language in §4.2.19.

2.3 Effect typing

2.3.1 Effects and interference

When the evaluation of an expression performs read or write actions on mutable data, the compiler must ensure that these actions occur in the correct order, else the meaning of the program will change. We have seen how region variables are used to reason about the *mutability* of data, and we now discuss how to reason about the actions. Following Talpin and Jouvelot [TJ92b] we use *effect typing* to annotate function types with a description of the actions each function performs.

For example, the *inc* function reads its argument, computes the successor, and writes the new value back to its argument. Adding this information to the type gives us:

$$\begin{array}{lll} inc & :: & \forall r_1.\ Int\ r_1 \xrightarrow{Read\ r_1 \vee Write\ r_1} () \\ & \rhd & Mutable\ r_1 \end{array}$$

The effect annotation on the function arrow tells us which regions in the store will be accessed when it evaluates. When the effect term becomes large this syntax is hard to read. For this reason we usually introduce an *effect variable*, and add a constraint to the type that contains the original effect term:

$$\begin{array}{lll} inc & :: & \forall r_1.\ Int\ r_1 \xrightarrow{e_1} () \\ & \rhd & e_1 = Read\ r_1 \vee Write\ r_1 \\ & , & Mutable\ r_1 \end{array}$$

Effect variables are identified as e_n in this text, and as variables preceded by an exclamation mark[2] `!en` in the concrete syntax. The exclamation mark is used as both a namespace qualifier and as the symbol for effect kinds. In the concrete syntax, effect constructors such as `!Read` and `!Write` are also preceded by this namespace qualifier. Akin to value type constructors, the effect constructors have specific kinds. Both *Read* and *Write* take a region and produce an effect, so we have:

$$\begin{array}{ll} Read & :: \% \rightarrow ! \\ Write & :: \% \rightarrow ! \end{array}$$

Treating the function constructor as a general type constructor, we can read the infix application $a \xrightarrow{e} b$ as shorthand for the prefix application $(\rightarrow)\ a\ b\ e$. This will help when presenting the typing rules of the core language, as we can use general type application to build function types instead of requiring a rule specific to functions.

Single, *atomic effects* such as $Read\ r_1$ and $Write\ r_1$ are gathered together with the join operator $\vee$. Effects form a lattice ordered by set inclusion on atomic effects. We use $\sigma_1 \sqsubseteq \sigma_2$ to mean effect σ_1 is no greater than effect σ_2, for example:

$$Read\ r_1 \sqsubseteq Read\ r_1 \vee Write\ r_1$$

The $\vee$ operator corresponds to set union. We use $\bot$ (bottom) to represent the effect of an expression which performs no visible actions, and a function arrow

[2] a mnemonic for: "something's happening!"

with no annotation is taken to have this effect. Conversely, we use $\top$ (top) to represent the effect of an expression which could perform all possible actions. This top element is useful because we can erase any effect term in our program by replacing it with $\top$, without loss of safety. We can also use $\top$ when the true effect of an expression is unknown. As we desire a top element in our effect structure, we use a lattice to gather effects instead of using sets directly. We also find the lattice notation more convenient, as we can write $\sigma \vee \mathit{Read}\ \ r_1$ instead of $\sigma \cup \{\mathit{Read}\ \ r_1\}$, where σ is an arbitrary effect. The original effect system of Gifford and Lucassen [GL86] is also presented as a lattice, though they do not use an explicit top element.

The notion of effect is intimately related to the notion of *interference* [Rey78], which relates to how the evaluation of one expression may affect the outcome of another. For example, if one expression has the effect $\mathit{Read}\ \ r_1$ and another has $\mathit{Write}\ \ r_1$, then they *may* be accessing the same heap object. In this case our compiler must worry about the order in which these two expressions are evaluated, and in particular, it must preserve this order when performing optimisations. Importantly, the notion of interference is separate from the usual method of propagating information between expressions via data dependencies. For example:

$$\begin{aligned} &y = \mathit{double}\ \ x \\ &z = \mathit{succ}\ \ y \end{aligned}$$

The evaluation of the first statement is most certainly going to affect the outcome of the second, but we don't count this as interference, because changing their order would violate the scoping rules of the language and prevent the program from being compiled.

On the other hand, if we had:

$$\begin{aligned} &y = \mathit{succ}\ \ x \\ &\mathit{inc}\ \ z \end{aligned}$$

These two statements may or may not interfere, depending on whether x, y or z are aliases for the same object.

When speaking of effects, we pronounce $\bot$ as "pure", because the evaluation of an expression with this effect can be safely reordered with respect to any other. We pronounce $\top$ as "sync" because an expression with this effect may interfere with all other impure expressions, so it must be synchronised with respect to them all.

2.3.2 Effect information in types

Here is the type of *updateInt*, which overwrites the value of its first argument with the second:

$$\begin{aligned} \mathit{updateInt}\ \ :: &\quad \forall r_1\ r_2.\ \mathit{Int}\ \ r_1 \rightarrow \mathit{Int}\ \ r_2 \xrightarrow{e_1} () \\ \rhd &\quad e_1 = \mathit{Write}\ \ r_1\ \vee\ \mathit{Read}\ \ r_2 \\ , &\quad \mathit{Mutable}\ \ r_1 \end{aligned}$$

When typeset, effect variables are written above the function arrow. However, in the concrete syntax we combine them with the arrow:

```
updateInt :: forall %r1 %r2
          .  Int %r1 -> Int %r2 -(!e1)> ()
          :- !e1 = !{ !Write %r1; !Read %r2 }
          ,  Mutable %r1
```

The syntax `!{ !e1; !e2; ... }` is equivalent to $e1 \vee e2 \vee ...$

All functions that write to a particular region also require that region to be mutable. When we express type signatures, we can leave out mutability constraints so long as we include the appropriate write effect.

On the other hand, the inclusion of a mutability constraint does not imply that a function is necessarily capable of writing to the associated region. The effect information in a type gives an *upper bound* on the particular actions a function may perform at runtime. For example, the following type signature is valid, but some of the information contained does not correspond to an actual property of the function:

$$\begin{array}{lll} \mathit{returnFive} & :: & \forall r_1\ r_2.\ \mathit{Int}\ r_1 \xrightarrow{e_1} \mathit{Int}\ r_2 \\ & , & e_1 = \mathit{Write}\ r_1 \\ & \triangleright & \mathit{Mutable}\ r_1 \\ \mathit{returnFive}\ x & = & 5 \end{array}$$

This is an example of *effect weakening*. It is always safe to treat a particular function (or expression) as having a larger effect than it necessarily does. With regard to interference, weakening the effect of an expression corresponds to synchronising its evaluation with other parts of the program, more than we would strictly need to.

Returning to the type of *updateInt*, the effect term we use for e_1 could really be anything we like, as long as it includes $\mathit{Write}\ r_1 \vee \mathit{Read}\ r_2$. Indeed, we could weaken its type by quantifying e_1 and making this fact explicit:

$$\begin{array}{lll} \mathit{updateInt} & :: & \forall r_1\ r_2\ e_1.\ \mathit{Int}\ r_1 \rightarrow \mathit{Int}\ r_2 \xrightarrow{e_1} () \\ & \triangleright & e_1 \sqsupseteq \mathit{Write}\ r_1 \vee \mathit{Read}\ r_2 \\ & , & \mathit{Mutable}\ r_1 \end{array}$$

Writing this another way, we could place the $e_1 \sqsupseteq \mathit{Write}\ r_1 \vee \mathit{Read}\ r_2$ constraint directly on the quantifier:

$$\begin{array}{lll} \mathit{updateInt} & :: & \forall r_1\ r_2\ \ (e_1 \sqsupseteq \mathit{Write}\ r_1 \vee \mathit{Read}\ r_2) \\ & . & \mathit{Int}\ r_1 \rightarrow \mathit{Int}\ r_2 \xrightarrow{e_1} () \\ & \triangleright & \mathit{Mutable}\ r_1 \end{array}$$

This new constraint gives a lower bound on the effect with which e_1 can be instantiated as. We will return to the practical differences between the strong and weak forms of *updateInt* in §2.3.6

Note that although atomic effects have a textual ordering when collected together with $\vee$, there is no corresponding information in the analysis. In the type of *updateInt*, the effect term $\mathit{Write}\ r_1$ appears before $\mathit{Read}\ r_1$ on the page, yet clearly the function must read the source argument before it writes to the destination. The $\vee$ operator is commutative so $\sigma_1 \vee \sigma_2$ is equivalent to $\sigma_2 \vee \sigma_1$. For comparison, in the behavior types of Nielson and Nielson [NN93, NN99], the order of actions is preserved.

2.3.3 Effects and currying

In our examples, usually only the right-most function arrow will have an effect annotation, though this is not required in general. Our primitive *updateInt* function needs both arguments before it can proceed, hence both $Read\ r_2$ and $Write\ r_1$ appear on the same arrow.

If we partially apply *updateInt* by supplying just the first argument, then the runtime system will build a thunk. This thunk holds a pointer to the object code for the "real" primitive update function, along with a pointer to the supplied argument. Building a thunk has no visible effect on the rest of the program, so this partial application is pure. Only when we apply the second and final argument will the runtime system be in a position to call the primitive function to carry out the update action.

In contrast, we could define a slightly different function that reads the source argument as soon as it is applied:

$$
\begin{array}{l}
readThenUpdateInt \\
\quad :: \quad \forall r_1\ r_2.\ Int\ r_1 \xrightarrow{e_1} Int\ r_2 \xrightarrow{e_2} () \\
\quad \rhd \quad e_1 = Read\ r_1 \\
\quad , \quad e_2 = Write\ r_2 \\
\quad , \quad Mutable\ r_2
\end{array}
$$

$$
\begin{array}{l}
readThenUpdateInt\ src \\
\quad = \quad \textbf{do} \quad src' = copyInt\ src \\
\qquad\qquad\ \ (\lambda\, dest \rightarrow updateInt\ dest\ src')
\end{array}
$$

where

$$
\begin{array}{l}
copyInt \\
\quad :: \quad \forall r_1\ r_2.\ Int\ r_1 \xrightarrow{e_1} Int\ r_2 \\
\quad \rhd \quad e_1 = Read\ r_1
\end{array}
$$

Note that unlike in Haskell, the Disciple do-expression is not monadic. A do-expression consists of a sequence of statements or bindings, terminated with a statement. The value of the whole expression is the value of the last statement. We treat **do** *binds*; *expr* as being sugar for **let** *binds* **in** *expr*, where the **let** is non-recursive.

In *readThenUpdateInt* we make a copy of the source argument as soon as it is available. The variable src' binds this copy and is free in the inner function. If we partially apply *readThenUpdateInt* to just its first argument, then the runtime system will build a thunk which references the copy. At this point we are free to update the original source object, without affecting the result of the inner function.

We can see this behavior in the type signature for *readThenUpdateInt*. Once the first argument is applied the function does not cause any more visible read effects.

2.3.4 Top level effects

So far we have only considered actions that modify the *internal* state of the program, that is, reads and writes to mutable data. For a general purpose language we must also be able to perform IO. The order of these actions must be maintained during compilation, and we can use the effect mechanism to do so. We refer to effects which represent actions on external state as *top-level* effects. These effects exist in the top level scope and cannot be safely masked.

Although the *Read* and *Write* effect constructors are "baked-in" to the language, we allow the programmer to define their own constructors to represent top level effects. For instance, for a typical interactive application we could define the following:

effect *Console*
effect *FileSystem*
effect *Network*

The primitive functions that access the outside world include these constructors in their effect terms. For example:

$$\begin{array}{lll} \mathit{putStr} & :: & \forall r_1.\ \mathit{String}\ r_1 \xrightarrow{e_1} () \\ & \rhd & e_1 = \mathit{Read}\ r_1 \vee \mathit{Console} \end{array}$$

The type of *putStr* tells us that it will read its argument and perform an action on the console. DDC ensures that the orderings of calls to *putStr* are maintained with respect to all functions that have top level effects.

In particular, if we define a function with a different top-level effect:

$$\begin{array}{lll} \mathit{readFile} & :: & \forall r_1\ r_2.\ \mathit{FilePath}\ r_1 \xrightarrow{e_1} \mathit{String}\ r_2 \\ & \rhd & e_1 = \mathit{Read}\ r_1 \vee \mathit{FileSystem} \end{array}$$

We must still synchronise uses of *readFile* with *putStr*, because in general, console and file actions may interfere. This point is discussed further in §5.2.6.

2.3.5 Effects in higher order functions

When we move to higher order functions, we begin to see effect variables in the types of their parameters. For example, the type of *map* is:

$$\begin{array}{lll} map & :: & \forall a\ b\ r_1\ r_2\ e_1 \\ & . & (a \xrightarrow{e_1} b) \rightarrow List\ \ r_1\ a \xrightarrow{e_2} List\ \ r_2\ b \\ & \rhd & e_2 = e_1 \vee Read\ \ r_1 \end{array}$$

$$\begin{array}{ll} map\ \ f\ [\,] & = [\,] \\ map\ \ f\ (x : xs) & = f\ x\ :\ map\ \ f\ \ xs \end{array}$$

The map function applies its first parameter to every element of a list, yielding a new list. It must inspect the list to determine whether it is empty or a cons cell, hence the $Read\ \ r_1$ effect. When it applies its parameter, that function invokes its actions, hence the variable e_1 also appears in the effect term for e_2.

The actual effect bound to e_1 depends on how *map* is applied. For example, we could use partial application to define a new function which will take the successor of a list of integers:

$$\begin{array}{lll} succ & :: & \forall r_3\ r_4 \\ & . & Int\ \ r_3 \xrightarrow{e_3} Int\ \ r_4 \\ & \rhd & e_3 = Read\ \ r_3 \end{array}$$

$$\begin{array}{lll} mapSucc & :: & \forall r_5\ r_6\ r_7\ r_8 \\ & . & List\ \ r_5\ (Int\ \ r_6) \xrightarrow{e_4} List\ \ r_7\ (Int\ \ r_8) \\ & \rhd & e_4 = Read\ \ r_6 \vee Read\ \ r_5 \end{array}$$

$$mapSucc\ \ =\ \ map\ \ succ$$

Due to the application $map\ \ succ$, the read effect of *succ* is bound to e_1 in the type of *map*. This effect term is then substituted into the constraint for e_2. Accounting for type generalisation, this read effect becomes the $Read\ \ r_6$ term in the type of *mapSucc*.

From the type of *mapSucc* we see that it will read the list cells from the region named r_5, as well as reading the element cells (via *succ*) from the region named r_6.

2.3.6 Constraint strengthening and higher order functions

The core of our type inference algorithm is modeled after the Type and Effect Discipline [TJ92b]. It returns a type term and a set of effect constraints for every expression in the program. This combination of type term and constraints corresponds to the "weak" version from §2.3.2. For example, the inferred type of *succ* would be:

$$\begin{array}{lll} succ & :: & \forall r_1\ r_2\ e_1.\ Int\ \ r_1 \xrightarrow{e_1} Int\ \ r_2 \\ & \rhd & e_1 \sqsupseteq Read\ \ r_1 \end{array}$$

We read this type as: a function which takes an *Int* in a region named r_1, returns an *Int* in a region named r_2, and whose evaluation causes an effect that includes $Read\ \ r_1$. We use $\sqsupseteq$ in the constraint because we can treat *succ* as having any effect, as long as it includes $Read\ \ r_1$. However, as the function

itself only has the $Read\ r_1$ effect, we will not lose any information if we replace $\sqsupseteq$ by $=$ and strengthen this type to:

$$\begin{array}{lll} succ & :: & \forall r_1\ r_2.\ Int\ r_1 \xrightarrow{e_1} Int\ r_2 \\ & \rhd & e_1 = Read\ r_1 \end{array}$$

We could also substitute the constraint into the body of the type, yielding the *flat* version:

$$succ \quad :: \quad \forall r_1\ r_2.\ Int\ r_1 \xrightarrow{Read\ r_1} Int\ r_2$$

We gain two immediate benefits when strengthening types in this way. Firstly, the types of most common library functions can be expressed without using the unfamiliar $\sqsupseteq$ operator, which reduces the number of symbols that beginners need to worry about, and is a benefit not to be underrated. The second is that it reduces the need for a large number of effect applications in programs which have been translated to our core language.

Our core language discussed in §4 is an extension of System-F, similar in spirit to the core language used in GHC. As usual, the instantiation of type schemes corresponds to type application in the core language. An application of *succ* using the weak version of its type would require an expression such as:

$$succ\ r_a\ r_b\ (Read\ r_1)\ x$$

Here, r_a, r_b and $Read\ r_1$ satisfy the $\forall r_1\ r_2\ e_1.$ portion of the type scheme. Both r_a and r_b are true parameters. They supply information regarding the location of the argument and return value, and are likely to be different for each use of *succ*. On the other hand, the fact that *succ* has the effect $(Read\ r_1)$ is obvious from its type, and supplying this information every time it is called needlessly increases the verbosity of the core program. This becomes problematic when we apply functions that have a more interesting behaviour. It is not uncommon for types in typical programs to have upwards of 20 atomic effect terms.

By strengthening the type of *succ* we can elide the effect application and apply the function with the smaller expression:

$$succ\ r_a\ r_b\ x$$

This is possible unless the application of *succ* genuinely needs to be treated as having a larger effect. This can occur for two reasons. Firstly, when choosing between two functions on the right of an **if** or **case**-expression, we must weaken their effect terms so that their types match. We discuss this further in §4.3.

Secondly, it is not obvious how to strengthen the types of higher order functions, or if this is even possible in general.[3] These types can include $\sqsupseteq$ constraints on effect variables that appear in parameter types. Such constraints require function parameters to have *at least* a certain effect, but as we can treat any function as having more effects than it is actually able to cause, they don't provide any useful information to the compiler. The fact that we have constraints of this form is an artefact of the bi-directional nature of the typing rules, and the Hindley-Milner style unification algorithm used to perform inference. The effect of a function can include the effect of its parameter, as per the *map* example, but also the other way around. We will see an example of this in a moment.

[3] I do not know how to do this, but do not have a proof that it is impossible.

First Order

We start with a simple first order function, *id*:

$$\begin{array}{lcl} id & :: & \forall a\ e_1.\ a \xrightarrow{e_1} a \\ & \rhd & e_1 \sqsupseteq \bot \\ id & = & \lambda x.\ x \end{array}$$

If an effect term corresponds to an action that could be carried out if the function were evaluated, then we call it a *manifest* effect of the function. In this example, e_1 is a manifest effect, albeit it is $\bot$. Clearly, *id* is pure so there is nothing preventing us from dropping the quantifier for e_1 and substituting $\bot$ for e_1 in the body of the type:

$$id :: \forall a.\ a \xrightarrow{\bot} a$$

Notice that in the original type, e_1 is manifest, and does not appear in the parameter portion of the type, that is, on the left of a function arrow.

Second Order

Here is an example second order function:

$$\begin{array}{lcl} appFive & :: & \forall a\ r_1\ e_1.\ (Int\ \ r_1 \xrightarrow{e_1} a) \xrightarrow{e_1} a \\ & \rhd & e_1 \sqsupseteq \bot \\ appFive & = & \lambda g.\ g\ 5 \end{array}$$

appFive accepts a function parameter and applies it to the integer 5. The effect caused by the use of *appFive* will be the same as the effect caused by the parameter function. This information is represented by the fact that e_1 appears in both the parameter type and as a manifest effect on right most function arrow. Although we have the constraint $e_1 \sqsupseteq \bot$, unlike the case for *id*, we cannot safely strengthen this type and substitute $\bot$ for e_1:

$$appFive_{bad} :: \forall a\ r_1.\ (Int\ \ r_1 \xrightarrow{\bot} a) \xrightarrow{\bot} a$$

This new type is strictly less general than the original because we can only apply it to parameter functions that are pure. However, $e_1 \sqsupseteq \bot$ is a statement that is always true, so we can drop it from the signature and write:

$$appFive :: \forall a\ r_1\ e_1.\ (Int\ \ r_1 \xrightarrow{e_1} a) \xrightarrow{e_1} a$$

In future we will always elide trivial constraints such as $e_1 \sqsupseteq \bot$. To make things slightly more interesting, we will add another effect to *appFive*:

$$\begin{array}{lcl} succFive & :: & \forall r_1\ r_2\ r_3\ e_1\ e_2 \\ & . & (Int\ \ r_1 \xrightarrow{e_1} Int\ \ r_2) \xrightarrow{e_2} Int\ \ r_3 \\ & \rhd & e_2 \sqsupseteq e_1 \vee Read\ \ r_2 \\ \\ succFive\ \ g & = & succ\ (g\ 5) \end{array}$$

succFive is similar to *appFive*, except that it passes the result of its parameter function to *succ*. This introduces the new effect $Read\ \ r_2$. Note that the effect of the parameter, e_1, and the manifest effect of the overall function are now linked

via the constraint on e_2. This is in contrast to *appFive*, where they were linked via a single variable. When we strengthen the effect constraint and substitute it into the body of the type we get:

$$\begin{array}{lll} succFive_{strong} & :: & \forall r_1\ r_2\ r_3\ e_1 \\ & . & (Int\ \ r_1 \xrightarrow{e_1} Int\ \ r_2) \xrightarrow{e_1 \vee Read\ \ r_2} Int\ \ r_3 \end{array}$$

Performing this substitution has not lost any information. We can see that the effect of evaluating *succFive* is to apply the parameter function and read its result. If desired, we could introduce a new effect variable for the $e_1 \vee Read\ \ r_2$ term, and convert the strong form back to the original weak version. In this case the two are equivalent.

For comparison, here is a second order function where strengthening does not work:

$$\begin{array}{lll} chooseFive & :: & \forall r_1\ r_2\ e_1 \\ & . & (Int\ \ r_1 \xrightarrow{e_1} Int\ \ r_2) \xrightarrow{e_1} Int\ \ r_2 \\ & \rhd & e_1 \sqsupseteq Read\ \ r_1 \\ chooseFive\ \ g & = & \textbf{let}\ f\ =\ \textbf{if}\ \ldots\ \textbf{then}\ g\ \textbf{else}\ succ \\ & & \textbf{in}\ \ f\ 5 \end{array}$$

Note that the if-expression is choosing between the parameter function g and and *succ*. The type inference algorithm uses unification to ensure that both these expressions have the same type. *succ* reads its argument, so g is treated as though it reads its argument also. This is the reason for the $Read\ \ r_1$ constraint on the variable e_1, which names the effect of the parameter function. It is important to note that the function parameter passed to *chooseFive* is now *required* to have the $Read\ \ r_1$ effect. If we wanted to apply *chooseFive* to the pure function *id*, then we would need to instantiate *id* with a weaker effect, so that it also contains $Read\ \ r_1$.

This "leaking" of a function's real, manifest effect into the type of its parameter is the other half of the bi-directional information flow discussed earlier. Interested parties are referred to the literature on intersection and union types as a possible way around this problem [CF04, DP03]. Such type systems can express more detailed properties of programs, but full type inference is often undecidable. Perhaps a union typing system guided by type annotations could give a more pleasing type to *chooseFive*. However, we have been primarily interested in compile time optimisation and are unconvinced of the benefits of a more complex system, so have not looked into this further.

Also, such constraints only seem to arise in programs that choose between functions, or use collection structures that contain functions. We haven't written many Disciple programs which do this, and are not sure if having constraints on effect variables in parameter types represents a real problem in the language.

We cannot strengthen the type of *chooseFive* and remove the $\sqsupseteq$ constraint as we did previously. Substituting $Read\ \ r_1$ for e_1 in the body would break the link between the effect of the parameter and the manifest effect of the overall function:

$$\begin{array}{lll} chooseFive_{bad} & :: & \forall r_1\ r_2 \\ & . & (Int\ \ r_1 \xrightarrow{Read\ \ r_1} Int\ \ r_2) \xrightarrow{Read\ \ r_1} Int\ \ r_2 \end{array}$$

For this reason we must include bounded quantification in both our source and core languages. We strengthen $\sqsupseteq$ constraints to $=$ constraints only when the variable does not appear in a parameter type (to the left of a function arrow). This simple rule allows us to elide the majority of effect applications that would otherwise appear once the program has been translated to the core language. As we shall see, there are cases where we could strengthen but don't, but they are rare in practice.

One more second order function follows. This time we have applied *succ* to the result of f to yield an additional read effect:

$$\begin{array}{lll} \mathit{chooseSuccFive} & :: & \forall r_1\ r_2\ r_3\ e_1\ e_2 \\ & . & (\mathit{Int}\ r_1 \xrightarrow{e_1} \mathit{Int}\ r_2) \xrightarrow{e_2} \mathit{Int}\ r_3 \\ & \rhd & e_1 \sqsupseteq \mathit{Read}\ r_1 \\ & , & e_2 \sqsupseteq e_1 \vee \mathit{Read}\ r_2 \end{array}$$

$$\begin{array}{lll} \mathit{chooseSuccFive}\ g & = & \textbf{let}\ f\ =\ \textbf{if}\ \ldots\ \textbf{then}\ g\ \textbf{else}\ \mathit{succ} \\ & & \textbf{in}\ \mathit{succ}\ (f\ 5) \end{array}$$

The point to notice here is that the constrained effect variable e_1 also appears in the constraint for e_2. This means that when we convert the type to use bounded quantifiers we must be careful about their order. For example, writing each quantifier separately gives:

$$\begin{array}{l} \mathit{chooseSuccFive} \\ \quad :: \quad \forall r_1.\ \forall r_2.\ \forall r_3.\ \forall(e_1 \sqsupseteq \mathit{Read}\ r_1).\ \forall(e_2 \sqsupseteq e_1 \vee \mathit{Read}\ r_2) \\ \quad . \quad (\mathit{Int}\ r_1 \xrightarrow{e_1} \mathit{Int}\ r_2) \xrightarrow{e_2} \mathit{Int}\ r_3 \end{array}$$

Unlike the first three region quantifiers, we cannot change the order of the two effect quantifiers, else e_1 would be out of scope in the second constraint. This has two important implications for our implementation.

The first is that although our type inference algorithm returns a type which includes a constraint *set* using $\rhd$, the core language uses individual bounded quantifiers as above. This means that when converting types to the core representation we must do a dependency walk over the constraint set to ensure the quantifiers are introduced in the correct order.

The second is that we have no way of representing graphical or recursive effect constraints in the core language, so we must break these loops during translation. This process is covered in §2.3.8 and §3.4.

Third Order

Moving up the chain, we now consider a third order function *foo*. We will reuse *appFive* in this example, so repeat its definition. We admit that *foo* is a constructed example, but make the point that a type system must handle such examples anyway. The reader is invited to analyse their own favourite third order function.[4]

$$\begin{array}{ll} \mathit{foo} & = \lambda f.\ \mathit{succ}\ (f\ \ \mathit{succ}) \\ \mathit{appFive} & = \lambda g.\ g\ 5 \end{array}$$

[4] We had enough trouble coming up with this one.

As the operation of *foo* is perhaps non-obvious to the casual observer, we offer an example call-by-value reduction of the term (*foo appFive*):

$$\begin{array}{lll} \mathit{foo}\ \ \mathit{appFive} & \longrightarrow & (\lambda f.\ \mathit{succ}\ (f\ \ \mathit{succ}))\ \ \mathit{appFive} \\ & \longrightarrow & (\lambda f.\ \mathit{succ}\ (f\ \ \mathit{succ}))\ (\lambda g.\ g\ 5) \\ & \longrightarrow & (\mathit{succ}\ ((\lambda g.\ g\ 5)\ \ \mathit{succ})) \\ & \longrightarrow & (\mathit{succ}\ (\mathit{succ}\ \ 5)) \\ & \longrightarrow & 7 \end{array}$$

The type of *foo* inferred by our system is:

$$\begin{array}{lll} \mathit{foo} & :: & \forall r_1\ r_2\ r_3\ r_4\ e_1\ e_2\ e_3 \\ & . & ((\mathit{Int}\ \ r_1 \xrightarrow{e_1} \mathit{Int}\ \ r_2) \xrightarrow{e_2} \mathit{Int}\ \ r_3) \xrightarrow{e_3} \mathit{Int}\ \ r_4 \\ & \triangleright & e_1 \sqsupseteq \mathit{Read}\ \ r_1 \\ & , & e_3 \sqsupseteq e_2 \vee \mathit{Read}\, r_3 \end{array}$$

foo takes a second order function as its parameter. In the source, *foo*'s parameter is applied to *succ*, hence the $(\mathit{Int}\ \ r_1 \xrightarrow{e_1} \mathit{Int}\ \ r_2)$ component of its type. As the result of this application is passed again to *succ*, the result has type $\mathit{Int}\ \ r_3$. The function *foo* itself returns the result of this final application, giving the return type $\mathit{Int}\ \ r_4$.

Note the semantic difference between the two effect constraints. The constraint on e_3 gives the manifest effect of evaluating the function, whereas the constraint on e_1 says that *foo*'s parameter will be passed a function which has a read effect.

In this type, as e_1 does not express a link between the parameter and the manifest effect of the function, we *could* strengthen it to:

$$\begin{array}{lll} \mathit{foo} & :: & \forall r_1\ r_2\ r_3\ r_4\ e_2 \\ & . & ((\mathit{Int}\ \ r_1 \overset{\mathit{Read}\ r_1}{\longrightarrow} \mathit{Int}\ \ r_2) \xrightarrow{e_2} \mathit{Int}\ \ r_3) \overset{e_2 \vee \mathit{Read}\, r_3}{\longrightarrow} \mathit{Int}\ \ r_4 \end{array}$$

However, functions of order three and higher are rare, so in our current implementation we stick with the simpler strengthening rule.

Higher order functions in practice

When researching the material in this section we had difficulty finding examples of useful functions of order three or greater. In [Oka98a] Okasaki suggests that in the domain of parser combinators, functions up to sixth order can be useful in practice. However, the signatures he presents use type synonyms, and the *principle* types of the combinators are of lower order. For example, using the ML syntax of the paper the *bind* combinator is:

$$\textbf{fun}\ \ \mathit{bind}\ \ (p,\ \ f)\ \ \mathit{sc} = p\ \ (\textbf{fn}\ x \Rightarrow\ \ f\ \ x\ \ \mathit{sc})$$

If we limit our self to simple types then this is a third order function:

$$\mathit{bind}\ \ : ((* \to *) \to *,\ * \to * \to *) \to * \to *$$

Yet its intended type signature, given as a *comment* in the ML code is:

$$(*\quad \mathit{bind} : \ `a\ \ \mathit{Parser}\ \ *\ \ (`a \to `b\ \ \mathit{Parser})\ \ \to\ \ `b\ \ \mathit{Parser}\quad *)$$

Although *Parser* is a type synonym for a third order function, it could be argued that this does not make *bind* fifth order.

2.3.7 Observable effects and masking

Consider the following function:

$$
\begin{array}{ll}
\mathit{slowSucc}\ x & \\
= \mathbf{do} & y\ \ = 0 \\
& y\ \ := y + 1 \\
& x + y
\end{array}
$$

We have used the operator (:=) as sugar for the *updateInt* function from §2.3.2. This function has six atomic effects. The two addition expressions read both their arguments, and the update function reads the result of $(y + 1)$ then overwrites the old value of y.

If we included all of these effects in the type for *slowSucc* then we would have:

$$
\begin{array}{lll}
\mathit{slowSucc} & :: & \forall r_1\ r_2\ r_3\ r_4\ r_5 \\
& . & \mathit{Int}\ r_1 \xrightarrow{e_1} \mathit{Int}\ r_5 \\
& \rhd & e_1\ \ = \mathit{Read}\ r_1 \vee \mathit{Read}\ r_2 \vee \mathit{Read}\ r_3 \vee \mathit{Read}\ r_4 \\
& & \quad \vee \mathit{Write}\ r_2 \\
& , & \mathit{Mutable}\ r_2
\end{array}
$$

Here is a version of *slowSucc* where the variables and constants have been annotated with the regions they are in, relative to the above type signature.

$$
\begin{array}{ll}
\mathit{slowSucc}\ x^{r_1} & \\
= \mathbf{do} & y^{r_2}\ \ = 0^{r_2} \\
& y^{r_2}\ \ := (y^{r_2} + 1^{r_3})^{r_4} \\
& (x^{r_1} + y^{r_2})^{r_5}
\end{array}
$$

The point to note is that much of the information in the type of *slowSucc* won't be of interest to a function that calls it. The constants 0 and 1, the value of y, and the result of the addition $(y + 1)$ are entirely local to the definition of *slowSucc*. If we so desired, space to hold these values could be allocated on the stack when calling the function, and then freed when returning from it. The fact that *slowSucc* makes use of these values is not *observable* from any calling context.

The only way a caller can communicate with a particular function is via its argument and return values, as well as via its free variables. A caller can pass an argument, receive a result, and in a language with destructive update the called function could modify values accessable via its free variables.

From the type signature for *slowSucc* we see that its argument is passed in a region named r_1, and its return value is produced into a region named r_5. Other than the addition and update operators, this particular function has no free variables. As regions r_2, r_3 and r_4 are not free in the body of the type, that is the $\mathit{Int}\ r_1 \xrightarrow{e_1} \mathit{Int}\ r_5$ term, the effects and constraints on these regions can be erased. We call this process *masking* those effects and constraints. This gives:

$$
\begin{array}{lll}
\mathit{slowSucc} & :: & \forall r_1\ r_5 \\
& . & \mathit{Int}\ r_1 \xrightarrow{e_1} \mathit{Int}\ r_5 \\
& \rhd & e_1\ \ = \mathit{Read}\ r_1
\end{array}
$$

Note that *slowSucc* has a pure interface. Although it uses destructive update internally, a calling function cannot observe this. This form of effect masking

achieves a similar result to monadic encapsulation of effects in the ST monad [LPJ94], with the advantage of being performed automatically by the compiler.

Here is another example:

$$
\begin{array}{lll}
\mathit{length}\ \mathit{xs} & & \\
= \mathbf{do} & n & = 0 \\
& \mathit{map\_}\ (\lambda\_.\ n := n + 1) & \mathit{xs} \\
& n &
\end{array}
$$

This imperative version of the list length function initialises a counter to zero, uses *map_* to increment the counter for every element of the list, then returns the counter. *map_* is similar to the standard *map* function, except that it discards its return value. When using *map_* the parameter function is only executed for its effect. In this way *map_* is similar to `mapM_` from Haskell. If we used just the masking rule from the previous example then we would have the following type for *length*:

$$
\begin{array}{lll}
\mathit{length} & :: & \forall a\ r_1\ r_2.\ \mathit{List}\ r_1\ a \xrightarrow{e_1} \mathit{Int}\ r_2 \\
& \rhd & e_1 = \mathit{Read}\ r_1 \vee \mathit{Write}\ r_2 \\
& , & \mathit{Mutable}\ r_2
\end{array}
$$

The *map_* function reads its argument list, so we have $\mathit{Read}\ r_1$ in the type of *length*. The expression $n := n + 1$ updates the value of n, which is finally returned. This gives $\mathit{Int}\ r_2$ as the return type, along with $\mathit{Write}\ r_2$ and $\mathit{Mutable}\ r_2$ as effects and constraints of the function.

Note that the return value of *length* is freshly allocated, so the calling function cannot have a reference to it beforehand. Because of this, the fact that the return value was created via destructive update is unimportant. We can use an additional masking rule: if a region variable is quantified, not present in a parameter type, and not present in the closure of the function, then effects and constraints on that region can be masked. We will discuss closures in §2.5. Masking the type of *length* above gives:

$$
\begin{array}{lll}
\mathit{length} & :: & \forall a\ r_1\ r_2.\ \mathit{List}\ r_1\ a \xrightarrow{e_1} \mathit{Int}\ r_2 \\
& \rhd & \mathit{Read}\ r_1
\end{array}
$$

Once again, we see that although *length* uses destructive update internally, it has a pure interface.

We will now sadly admit that although our current implementation of DDC masks the $\mathit{Write}\ r_2$ effect in the type of *length* it does not also mask the $\mathit{Mutable}\ r_2$ constraint. Although we can plainly see that this is a valid operation in the source language, we do not yet have a system in place to mask the corresponding constraint in the core language. In future work we plan to use the system outlined by Gupta [Gup95] to do so. This is discussed further in §2.8.7 and §5.2.1.

2.3.8 Recursive effects

Consider the following function:

$$
\begin{array}{l}
fac\ n \\
\quad = \textbf{case}\ n\ \textbf{of} \\
\qquad 0 \quad \rightarrow 1 \\
\qquad \_ \quad \rightarrow n * fac\ (n-1)
\end{array}
$$

This function also contains six separate sources of effects. Firstly, when the case-expression evaluates it must read the value of n to determine which alternative to take. The multiplication and subtraction expressions must read their operands. Finally, evaluation of the recursive call to *fac* causes all of these effects again. Just as the recursive function *fac* is defined in terms of itself, the effect of *fac* includes itself.

With this in mind we could give *fac* the following type:

$$
\begin{array}{lll}
fac & :: & \forall r_1\ r_2\ e_1.\ Int\ r_1 \xrightarrow{e_1} Int\ r_2 \\
 & \rhd & e_1 \sqsupseteq Read\ r_1 \vee e_1
\end{array}
$$

The effect term $Read\ r_1$ is due to the **case**, multiply and subtraction expressions, and e_1 is due to the recursive call. As per the previous section, we have masked the effect of reading the two '1' constants.

Now, although the effect e_1 is constrained to include itself, the fact that e_1 is recursive is not used by our subsequent analysis. Due to this, we will simplify this type by breaking the recursive loop. We do this by first decomposing the constraint $e_1 \sqsupseteq Read\ r_1 \vee e_1$ into two parts:

$$
\begin{array}{ll}
e_1 & \sqsupseteq Read\ r_1 \\
e_1 & \sqsupseteq e_1
\end{array}
$$

The second part, $e_1 \sqsupseteq e_1$ is trivially satisfied, so we can write the type of *fac* in a simpler form:

$$
\begin{array}{lll}
fac & :: & \forall\ r_1\ r_2\ e_1.\ Int\ r_1 \xrightarrow{e_1} Int\ r_2 \\
 & \rhd & e_1 \sqsupseteq Read\ r_1
\end{array}
$$

We can also apply the effect strengthening rule to eliminate the quantifier for e_1 and change the constraint operator from $\sqsupseteq$ to $=$. This gives us our final type:

$$
\begin{array}{lll}
fac & :: & \forall\ r_1\ r_2.\ Int\ r_1 \xrightarrow{e_1} Int\ r_2 \\
 & \rhd & e_1 = Read\ r_1
\end{array}
$$

Note that as our core language cannot represent recursive effect types, we must always perform this loop breaking simplification. Other systems based on behaviors and trace effects [NN93, SSh08] express these loops using a fix point operator, but we are not aware of any way to use this information to optimise the program.

2.3.9 Constant regions and effect purification

Recall from §2.2.4 that the constraint *Mutable* r_1 indicates that region r_1 *may* be updated, while *Const* r_1 indicates that it will *never* be updated. During type inference, once all the region constraints from a source program have been processed, any regions that have not been constrained to be mutable are assumed to be constant. This is the first source of *Const* constraints in our system.

The second source is the use of lazy evaluation. In Disciple, lazy evaluation is introduced by suspending particular function applications. We do this with the suspension operator @. For example:

$$six = succ\ @\ 5$$

This syntax is desugared into an application of the primitive arity-1 suspend function:

$$six = suspend1\ \ succ\ 5$$

Where *suspend1* has type:

$$\begin{array}{lll} suspend1 & :: & \forall a\ b\ e_1.\ (a \xrightarrow{e_1} b) \rightarrow a \rightarrow b \\ & \rhd & Pure\ \ e_1 \end{array}$$

Note that as the two right most function arrows have no effect annotations, they are taken to be $\bot$ (pure). *suspend1* takes a parameter of type $a \xrightarrow{e_1} b$, an argument of type a and defers the application by building a thunk at runtime. When the value of this thunk is demanded, the function parameter will be applied to its argument, yielding the result of type b. Clearly, the function parameter must not cause visible side effects. If it did then the value of its result would depend on *when* the thunk is forced, which usually won't be what the programmer had intended. For this reason, the effect constraint *Pure* e_1 requires the visible effect of the function parameter to be $\bot$.

We now consider the type of *succ* including region and effect information:

$$\begin{array}{lll} succ & :: & \forall r_1\ r_2.\ Int\ \ r_1 \xrightarrow{e_1} Int\ \ r_2 \\ & \rhd & e_1 = Read\ \ r_1 \end{array}$$

The type of *succ* includes an effect *Read* r_1, and when *suspend1* is applied to *succ* we get the constraint *Pure* (*Read* r_1). Now, *Read* r_1 is not the $\bot$ which this constraint requires. However, suppose r_1 was constant. Read effects on constant regions can be safely ignored because it does not matter when a particular read takes place, the same value will be returned every time. During type inference, purity constraints on read effects are discharged by forcing the regions read to be constant. We call this *effect purification*.

If the region happens to already be mutable then it cannot additionally be made constant. In this case the system reports a *purity conflict* and gives an error message that includes the term in the program that caused the region to be marked as mutable, along with the suspension that requires it to be constant.

For example:

$$
\begin{array}{l}
succDelay\ () \\
= \textbf{do} \quad x \quad = 5 \\
\qquad\quad y \quad = succ\ @\ x \\
\qquad\quad \ldots \\
\qquad\quad x \quad := 23 \\
\qquad\quad \ldots
\end{array}
$$

In this program we have suspended the application of *succ*, which will read the integer bound to x. Later in the program, this integer will be updated to have a new value, 23. The trouble is that the eventual value of y will depend on *when* this result is demanded by the surrounding program. If it is demanded before the update then it will evaluate to 6, but if it is demanded after it will evaluate to 24.

The usual sense of an erroneous program is one that cannot be reduced to a value because the reduction reaches a point where no further rule applies, such as with $True + 42$. Although *succDelay* does not have this problem, we argue that its behaviour is non-obvious enough to justify rejection by the type system. This is akin to compiler warnings about uninitialised variables in C programs. Uninitialised variables *per se* will not crash a program, but the behavior of a program which uses them can be so confusing that it is best to reject it outright.

Of course, in a particular implementation we can always add a trapdoor. Our *suspend1* function is primitive, but is not baked into the type system. In our runtime system we have implemented *suspend1* in C. We import it with the foreign function interface, like any other primitive function. To allow *succDelay* we would simply import the C implementation of *suspend1* again with a different name and leave the $Pure\ e_1$ constraint out of the new type signature. This would be akin to using the *unsafePerformIO* function with GHC. *unsafePerformIO* allows a side-effecting function to be used in a context that demands a pure one, leaving the burden of correctness on the programmer instead of the compiler and type system.

2.3.10 Purification in higher order functions

Purity constraints can also be applied to the effects of function parameters. This is common for higher order functions that work on lazy data structures. For example, here is a definition of the lazy map function, which reads elements of the input list only when the corresponding element of the output is demanded.

$$
\begin{array}{ll}
mapL\ f\ [\,] & = [\,] \\
mapL\ f\ (x : xs) & = f\ x\ :\ mapL\ f\ @\ xs
\end{array}
$$

We will desugar the pattern match into a case-expression, use *Nil* and *Cons* in place of [ ] and :, as well as using the equivalent *suspend* function in place of @.

$$
\begin{array}{l}
mapL\ f\ xx \\
= \textbf{case}\ xx\ \textbf{of} \\
\qquad \begin{array}{ll}
Nil & \rightarrow Nil \\
Cons\ x\ xs & \rightarrow Cons\ (f\ x)\ (suspend1\ (mapL\ f)\ xs)
\end{array}
\end{array}
$$

The effect of *mapL* includes the effect of inspecting the value of xx in the case-expression, as well as the effect of evaluating the application $f\ x$. On the other

hand, the use of *suspend1* requires $(mapL\ \ f)$ to be a pure function. The fact that *mapL* suspends its recursive call forces it to be pure.

We can purify the effect of the case-expression by requiring the cons cells of the list to be in a constant region. We cannot purify the effect of $f\ x$ locally, because f is an unknown function, but we can require that callers of *mapL* provide a guarantee of purity themselves. We do this by placing a purity constraint on the effect of f, which gives *mapL* the following type:

$$\begin{array}{lll} mapL & :: & \forall a\ b\ r_1\ r_2\ e_1 \\ & . & (a \xrightarrow{e_1} b) \rightarrow List\ \ r_1\ a \xrightarrow{e_2} List\ \ r_2\ b \\ & \rhd & e_2 \quad = e_1 \vee Read\ \ r_1 \\ & , & Pure\ \ e_1 \\ & , & Const\ \ r_1 \end{array}$$

This says that we can only use *mapL* with pure parameter functions, and with constant lists. These constraints are sufficient to guarantee that the value returned will not depend on when it is demanded.

The above type is the one produced by our current implementation. Note that even though $Read\, r_1$ and e_1 are pure, we have retained these effects in the constraint for e_2. It would be "nicer" to erase them, but we have not yet implemented a mechanism to perform the corresponding effect masking in the core language, which is discussed in §4.3.1.

Alternatively, erasing these effects would produce the following type:

$$\begin{array}{lll} mapL & :: & \forall a\ b\ r_1\ r_2\ e_1 \\ & . & (a \xrightarrow{e_1} b) \rightarrow List\ \ r_1\ a \rightarrow List\ \ r_2\ b \\ & , & Pure\ \ e_1 \\ & , & Const\ \ r_1 \end{array}$$

The two constraints $Pure\ \ e_1$ and $Const\ \ r_1$ express the *implicit* constraints on functions and data present in lazy languages such as Haskell. In Haskell, all functions are pure[5] and all data is constant.[6] By adding a single @ operator to our strict version of *map* we have created the lazy version. This new version is type compatible with the strict version, except for the added constraints that ensure referential transparency.

[5] Bar some hacks when implementing IO.

[6] Though, not as far as the runtime system is concerned.

2.3.11 Strict, spine lazy and element lazy lists

Returning to the sugared version of *mapL*, note that this function is *spine lazy*.

$$\begin{array}{ll} mapL\ f\ [\,] & = [\,] \\ mapL\ f\ (x : xs) & = f\ x\ :\ mapL\ f\ @\ xs \end{array}$$

A spine lazy map is one that only allocates cons cells for the output list when they are demanded. Alternatively, we could move the @ operator and create a version that allocated all of the cons cells as soon as it was called, but deferred the evaluation of the actual list elements:

$$\begin{array}{ll} mapLE\ f\ [\,] & = [\,] \\ mapLE\ f\ (x : xs) & = f\ @\ x\ :\ mapLE\ f\ xs \end{array}$$

We mention this because in our introduction we discussed the fact that in Haskell, the functions *map* and *mapM* are conceptually similar, but require different definitions and have different types. We argued that this created a need to refactor lots of existing code when developing programs. Although we have now introduced *three* different Disciple versions, *map*, *mapL*, *mapLE* which are strict, spine lazy, and element lazy respectively, this is a different situation.

In the types of these three functions, the value type portion remains the same. If we cover up the region, effect and constraint information, we are left with an identical type in each case:

$$map :: (a \to b) \to List\ a \to List\ b$$

The three versions *map*, *mapL*, *mapLE* are all interchangeable as far as their value types are concerned. This is comparable to the difference between *foldl* and *foldl′* in the standard Haskell libraries. *foldl′* is a stricter version of *foldl*, but it has the same type.

Of course, in Disciple we still want *mapM* when using monads such as parsers. The fact that we can express side effecting programs without needing state monads does not imply the monad abstraction is not useful for other purposes.

2.3.12 Lazy and Direct regions

Region classes are a general mechanism that allows us to express specific properties of data. We have already discussed the *Mutable* and *Const* classes that are used to express whether an object may be updated or must remain constant. We use the additional classes *Lazy*, *LazyH* and *Direct* to track the creation of thunks due to the use of *suspend* functions. A *Lazy* constraint applied to the primary region of a data type indicates that values of that type may be represented as thunks. *LazyH* applied to a type variable indicates that the top level (head) region of that type may be a thunk. On the other hand *Direct* applied to a primary region variable indicates that the object is guaranteed *not* to be a thunk. This allows us to optimise the handling of boxed values in the core language, as well as improve code generation for case expressions in the back end.

Note that the concepts of directness and *strictness* are quite different. When a function is strict in its parameter, if the evaluation of a particular argument

diverges then the application of the function to this argument also diverges. On the other hand, when a function is direct in its parameter, it will not accept values represented by thunks, and when it is direct in its result, it will not produce thunks.

Here is a version of *suspend1* that uses a *LazyH* constraint to indicate that this function produces thunks:

$$\begin{array}{lll} \mathit{suspend1} & :: & \forall a\ b\ e_1.\ (a \xrightarrow{e_1} b) \to a \to b \\ & \rhd & \mathit{Pure}\ e_1 \\ & , & \mathit{LazyH}\ b \end{array}$$

We will suspend an application of *succ* as an example:

$$x = \mathit{suspend1}\ \ \mathit{succ}\ 5$$

To work out the type of x, we first instantiate the types of *suspend1* and *succ*. We have used primed variables for the instantiated names:

$$\begin{array}{lll} \mathit{suspend1}_{\,\mathit{inst}} & :: & (a' \xrightarrow{e_1'} b') \to a' \to b' \\ & \rhd & \mathit{Pure}\ e_1',\ \mathit{LazyH}\ b' \\ \mathit{succ}_{\mathit{inst}} & :: & \mathit{Int}\ r_1' \xrightarrow{e_2} \mathit{Int}\, r_2' \\ & \rhd & e_2 = \mathit{Read}\ r_1' \end{array}$$

Applying $\mathit{suspend1}_{\,\mathit{inst}}$ to $\mathit{succ}_{\mathit{inst}}$ gives:

$$\begin{array}{lll} (\mathit{suspend1}\ \ \mathit{succ}) & :: & \mathit{Int}\ r_1' \to \mathit{Int}\ r_2' \\ & \rhd & \mathit{Pure}\ (\mathit{Read}\ r_1'),\ \ \mathit{LazyH}\ (\mathit{Int}\ r_2') \end{array}$$

By assigning the constant 5 the type $\mathit{Int}\ r_1'$ we get:

$$\begin{array}{lll} (\mathit{suspend1}\ \ \mathit{succ}\ 5) & :: & \mathit{Int}\ r_2' \\ & \rhd & \mathit{Pure}\ (\mathit{Read}\ r_1'),\ \ \mathit{LazyH}\ (\mathit{Int}\ r_2') \end{array}$$

We reduce the $\mathit{Pure}\ (\mathit{Read}\ r_1')$ constraint by requiring that r_1' is constant. The constraint $\mathit{LazyH}\ (\mathit{Int}\ r_2')$ tells us that r_2' may contain a thunk, so we reduce it to $\mathit{Lazy}\ r_2'$:

$$(\mathit{suspend1}\ \ \mathit{succ}\ 5) \ :: \ \mathit{Int}\ r_2' \ \rhd \ \mathit{Const}\ r_1',\ \mathit{Lazy}\ r_2'$$

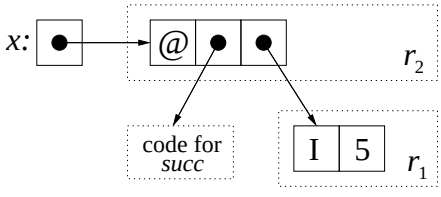
Although this type includes the constraint $\mathit{Const}\ r_1'$, the region variable r_1' is not present in its body. The region r_1' relates to the constant value 5, not to the resulting value x, so we can drop it and get:

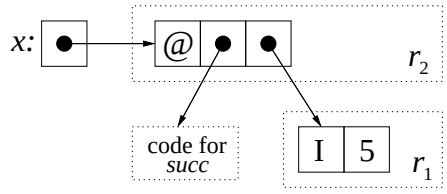
$$(\mathit{suspend1}\ \ \mathit{succ}\ 5) \ :: \ \mathit{Int}\ r_2' \ \rhd \ \mathit{Lazy}\ r_2'$$

The region variable r_2 cannot be quantified because it is material in this type. The final type of x is:

$$x \ :: \ \mathit{Int}\ r_2' \ \rhd \ \mathit{Lazy}\ r_2'$$

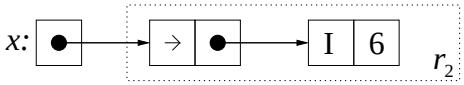
This says that the outer-most constructor of this object may be a thunk, and it certainly will be after the call to *suspend1*:





As the application thunk represents a value of type $Int\ r_2'$ we draw it as belonging to the region r_2. This is opposed to thunks that represent partial applications. These thunks have no regions because they always represent objects of function type, and function types are not annotated with region variables.

When the value of x is forced, the application $succ\ 5$ will be evaluated. Following lazy evaluation, the thunk will then be overwritten by an indirection node pointing to the result:



During back end code generation, we must account for the fact that x may point to a thunk or indirection. To extract the unboxed integer from x we must first load the tag of the object pointed to. This allows us to identify the sort of object it is, and decide whether to force the thunk, follow the indirection, or load the value as required. On the other hand, if we knew that x was direct, as with:

$$x \quad :: \quad Int\ r_2' \ \rhd\ Direct\ r_2'$$

Then we would be sure that x only pointed to a boxed integer. This would save us from having to load the tag and do the test. Similarly to the way non-mutable regions default to being constant, non-lazy regions default to being direct.

2.3.13 Liftedness is not a capability

We should note that the constraint names *Lazy* and *Direct* have an operational flavour because DDC uses this information to guide optimisations. We could perhaps rename them to *Lifted* and *Unlifted*, which would reflect the fact that a *Lifted* value represents a computation that may diverge.

A similar approach is taken in [LP96] and [PJSLT98], though they distinguish between pointed and lifted types. In [LP96], the type of unlifted integers is written $Int^{\#}$. The type of lifted integers is defined to be $Int^{\#}_{\perp}$, with the $\perp$ in the subscript acting as a type operator that allows the bottom element to be one of the "values" represented by the type. Note that with this formulation, monotypes such as integers must be either lifted or unlifted.

Our method of attaching constraints to region variables allows us to reuse the type class machinery to encode a similar property. However, type class constraints express a "supports" relationship, which doesn't quite match up with the concept of liftedness. For example, the constraint $Eq\ a$ means that a is a type whose values support being tested for equality. The constraint $Mutable\ r$ means that the objects in region r support being updated. Likewise, $Const\ r$

means that the objects in r can be safely read by a suspended computation, that is, they support laziness. If an object is constrained to be neither *Mutable* nor *Const* then we cannot assume it is safe to do either of these things.

Extensionally, if a type is completely unconstrained then we know nothing about the values that inhabit that type. Each new constraint provides a new piece of information, and that information gives us the capability to do something new with the corresponding values.

If a region is *Direct* then we can generate faster code to read objects in that region, because they are guaranteed not to be represented by thunks. However, the fact that a region is *Lazy* doesn't provide us with an additional capability. *Lazy* constraints are used only to ensure that a region is not also treated as *Direct*, as once we add thunks to a region we must test for them when reading every object from that region. In this sense, *Lazy* is a sort of "anti-capability" that indicates that a region has definitely been polluted by thunks and can no longer be used "directly".

For example, consider the following type:

$$fun :: \forall r_1\ r_2.\ Int\ r_1 \rightarrow Int\ r_2$$

As r_1 is unconstrained, objects passed to this function *may* be represented by thunks. If instead we had:

$$fun :: \forall r_1\ r_2.\ Int\ r_1 \rightarrow Int\ r_2\ \rhd\ Direct\ r_1$$

Then objects passed to the function are guaranteed *not* to be represented by thunks, and we can optimise the function using this information. On the other hand, if we had:

$$fun :: \forall r_1\ r_2.\ Int\ r_1 \rightarrow Int\ r_2\ \rhd\ Lazy\ r_1$$

The *Lazy* constraint says that objects passed to this function may be represented by thunks, but this isn't new information compared with the first version. However, during type inference, if we discover that a term has type:

$$Int\ r_1 \rhd Lazy\ r_1,\ Direct\ r_1$$

Then this could mean that a lazy object, which might be a thunk, was passed to a function that can only accept a direct object, which cannot be a thunk. This is invalid, and will be marked as a type error.

2.4 The problem with polymorphic update

A well known problem can arise when destructive update is added to a language with a Hindley-Milner style polymorphic type system. The classic example is as follows:

$$
\begin{array}{ll}
id & :: \forall a.\ a \to a \\
succ & :: Int \to Int
\end{array}
$$

$$
\begin{array}{l}
broken\ () \\
= \mathbf{do} \quad ref = newRef\ \ id \\
\qquad\quad writeRef\ \ ref\ \ succ \\
\qquad\quad (readRef\ \ ref)\ \ True
\end{array}
$$

If we treated this function as though it were written in Standard ML, we could argue that it is not type safe and would likely crash at runtime. The first line creates a reference to a polymorphic function *id*, while the second updates it to hold a less general function *succ*. This invalidates the original type of *ref*. The problem appears to center on the type inferred for *ref*:

$$ref :: \forall a.\ Ref\ (a \to a)$$

The ∀-quantifier allows us to instantiate this type differently for each use of *ref*. However, our static type system is unable to track the fact that once we update the reference we can no longer treat it has having this general type.

2.4.1 Fighting the value restriction

After winning out over several other systems [Gar02] the standard way of addressing the problem with polymorphic update is to apply the *value restriction* [Wri96]. The value restriction states that the type of a let-bound variable should only be generalised if the right of the binding is a syntactic value, such as a variable, literal, lambda abstraction, or application of a data constructor to another value. These expressions are called *non-expansive* because their evaluation will neither generate an exception or extend the domain of the store [MTHM97, Tof90].

The value restriction has the advantages that it is simple, easy to implement, and does not require extra information to be attached to the structure of types. This last point is especially important for ML-style languages in which the programmer must write full type signatures when defining module interfaces.

The down side is that a class of expressions that were previously assigned polymorphic types lose their polymorphism. For example:

$$f = map\ \ id$$

The type of f is not generalised because the right of the binding is not a syntactic value. To regain polymorphism we must η-expand it to give:

$$f = \lambda x.\ \ map\ \ id\ \ x$$

or equivalently, write it as a function binding:

$$f\ x = map\ \ id\ \ x$$

In [Wri96] it was argued that as the number of modifications needing to be performed to existing ML programs was small compared to the overall size of the code, the value restriction does not place an undue burden on the programmer in practice. However, in light of more recent languages such as Haskell [PJ03a], the value restriction would interfere with applications such as parser combinator libraries, which make heavy use of polymorphic values [LM01].

More recently, a variant named the *relaxed value restriction* [Gar02] uses a subtyping based approach to recover some of the polymorphism lost by the simpler restriction. Unfortunately, straight-forward examples like $(map\ \ id)$ remain monomorphic.

2.4.2 Don't generalise variables free in the store typing

In [Tof90] Tofte uncovers the crux of the problem with polymorphic update by attempting to prove the soundness of an ML-style type system with mutable references, and showing where the proof breaks down.

Unsurprisingly, the offending case is the one for let-bindings. The dynamic rule is as follows:

$$\frac{s\ ;\ E \vdash t_1 \longrightarrow v_1\ ;\ s_1 \qquad s_1\ ;\ E + \{x \mapsto v_1\}\ \vdash t_2 \longrightarrow v\ ;\ s'}{s\ ;\ E \vdash \textbf{let}\ x = t_1\ \textbf{in}\ t_2 \longrightarrow v\ ;\ s'} \qquad \text{(MLEvLet)}$$

The judgement form $s\ ;\ E \vdash t \longrightarrow v\ ;\ s'$ is read: starting with store s and environment E, the expression t evaluates to value v and a (perhaps changed) store s'. The store s maps locations to values while the environment E maps variables to values. Store locations are created when we allocate a new reference cell, and modifying the contents of a reference cell corresponds to changing the value bound to a particular location. The corresponding type rule is:

$$\frac{\Gamma \vdash t_1 :: \tau_1 \qquad \Gamma,\ x : \text{Gen}(\Gamma,\ \tau_1) \vdash t_2 :: \tau}{\Gamma \vdash \textbf{let}\ x = t_1\ \textbf{in}\ t_2 :: \tau} \qquad \text{(MLTyLet)}$$

Here, $\text{Gen}(\Gamma,\ \tau_1)$ performs generalisation and is short for $\forall a_1..a_n.\ \tau_1$, where $a_1...a_n$ are the type variables in τ_1 that are not free in Γ.

In general, t_1 may contain location variables, so we need to know the types of the values bound to these locations before we can check the type of the whole expression. This information is held in the *store typing* which maps locations to types.

If we have an expression t_1 of type τ_1, then reducing it relative to a particular store s_1 should yield a value v_1. We desire this value to have the same type as the original expression, and express this fact with the statement:

$$s_1 :: ST_1 \models v_1 :: \tau_1$$

This statement reads: in store s_1 with typing ST_1, v_1 has type τ_1. Now the trouble starts. Although we know that v_1 has type τ_1, when evaluating a let-expression we must satisfy the the second premise of (MLTyLet).

This requires that we strengthen the previous statement to:

$$s_1 :: ST_1 \models v_1 :: \text{Gen}(\Gamma,\ \tau_1)$$

This says that we're now considering the value to have a more general type than it used to. An example of this would be to first treat the term $(\lambda x.\ x)$ as having the monomorphic type $b \to b$, and then later deciding that it has the more general, polymorphic type $\forall b.\ b \to b$. In a language without references, as long as b is not free in the type environment then this generalisation is justified.

If b is not free in the type environment, then there is nothing stopping us from α-converting any local uses of it, and thus eliminating all mention of this particular variable from our typing statements. By doing this we could be sure that no other parts of the program are treating b as being any specific, concrete type, because they have no information about it.

However, when we introduce mutable references we must also introduce the concept of a store and its associated store typing. This store typing includes type variables, and when we try to strengthen the original statement the proof falls apart. Consider again our *broken* example that creates a reference to the polymorphic function *id*. Expanding out the definition of *id* gives:

$$\begin{array}{l}\textbf{let}\ \ ref = newRef\ (\lambda x.\ x)\\ \textbf{in}\ \ \ldots\end{array}$$

Once $newRef\ (\lambda x.\ x)$ has been reduced to a value, the statement we need to strengthen is:

$$\{loc_1 \mapsto \lambda x.\ x\} :: \{loc_1 \mapsto (a \to a)\}\ \models\ loc_1 :: a \to a$$

Notice how the reduction of $newRef\ (\lambda x.\ x)$ has created a new location in the store and bound the identity function to it. In the *store typing* this function has the type $(a \to a)$ which includes the free variable a.

However, during generalisation this fact is ignored and we end up with:

$$\{loc_1 \mapsto \lambda x.\ x\} :: \{loc_1 \mapsto (a \to a)\}\ \models\ loc_1 :: \boldsymbol{\forall a.\ a \to a}$$

This statement is clearly suspect because the type assigned to loc_1 no longer models its type in the store. When we update the reference to hold *succ*, the type of the binding in the store changes. Unfortunately, the static typing rules still treat loc_1 as having the more general type:

$$\{loc_1 \mapsto succ\} :: \{loc_1 \mapsto (Int \to Int)\}\ \models\ loc_1 :: \boldsymbol{\forall a.\ a \to a}$$

If we were to then read the *succ* function back from the store and apply it to a non-*Int* value like *True*, the runtime result would be undefined.

Tofte sums up the problem with the following observation:

> *The naive extension of the polymorphic type discipline* [with mutable references] *fails because it admits generalisation on type variables that occur free in the store typing.*

2.4.3 Generalisation reduces data sharing in System-F

The value restriction does not solve the fundamental problem of a static analysis being unable to track runtime changes in the type of data. What it does is to limit polymorphism, and to prevent the user from writing a certain class of programs.

Issues of *soundness* can only arise in relation to a well defined semantics. The usual formulation being "Soundness = Progress + Preservation" [Pie02], meaning that a well-typed expression must either be a value or be able to progress to the next step in its evaluation; and that its well-typing is preserved during evaluation.

With an ML style semantics, if we fail to deal adequately with the issue of polymorphic update then the last line in *broken* from §2.4 reduces as:

$$\begin{aligned}
&(\mathit{readRef}\ \mathit{ref})\ \ \mathit{True}\\
&\quad\longrightarrow\ \mathit{succ}\ \ \mathit{True}\\
&\quad\longrightarrow\ (\lambda x.\ x+1)\ \ \mathit{True}\\
&\quad\longrightarrow\ \mathit{True}+1
\end{aligned}$$

This term is not a value and cannot be evaluated further as there is no reduction rule specifying how to add one to a boolean value. It is the *combination* of operational and static semantics which is unsound.

On the other hand, if we consider a System-F style translation of *broken* which has been typed without restricting generalisation then we would have:

$$\begin{aligned}
&\mathit{broken} = \lambda\ ().\\
&\qquad \mathbf{let}\ \mathit{ref}\ \ = \Lambda b.\ \mathit{newRef}\ \{b \to b\}\ (\mathit{id}\ \{b\})\ \ \mathbf{in}\\
&\qquad \mathbf{let}\ _\ \ \ = \mathit{writeRef}\ \{\mathit{Int} \to \mathit{Int}\}\ (\mathit{ref}\ \{\mathit{Int}\})\ \ \mathit{succ}\ \ \mathbf{in}\\
&\qquad \mathit{readRef}\ \{\mathit{Bool} \to \mathit{Bool}\}\ (\mathit{ref}\ \{\mathit{Bool}\})\ \ \mathit{True}
\end{aligned}$$

We have inserted type lambdas Λ and type arguments $\{\}$ at generalisation and instantiation points respectively. Notice that *ref* now binds a *function value* instead of an application expression.

From the operational semantics of DDC's core language §4.2.13 we have:

$$\frac{\mathrm{H}\ ;\ t_1 \longrightarrow \mathrm{H}'\ ;\ t_1'}{\mathrm{H}\ ;\ \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \longrightarrow \mathrm{H}'\ ;\ \mathbf{let}\ x = t_1'\ \mathbf{in}\ t_2} \qquad \text{(EvLet1)}$$

$$\mathrm{H}\ ;\ \mathbf{let}\ x = v^\circ\ \mathbf{in}\ t \longrightarrow \mathrm{H}\ ;\ t[v^\circ/x] \qquad \text{(EvLet)}$$

When combined, these two rules say that to evaluate a let-expression we should first reduce the right of the binding to a (weak) value and then substitute this value into the body. While evaluating *broken*, as the right of the *ref* binding is already a value we substitute and end up with:

$$\begin{aligned}
&\mathbf{let}\ _\ \ \ = \mathit{writeRef}\ \{\mathit{Int} \to \mathit{Int}\}\ ((\Lambda b.\ \mathit{newRef}\ \ \ldots)\ \{\mathit{Int}\})\ \ \mathit{succ}\ \ \mathbf{in}\\
&\mathit{readRef}\ \{\mathit{Bool} \to \mathit{Bool}\}\ ((\Lambda b.\ \mathit{newRef}\ \ \ldots)\ \{\mathit{Bool}\})\ \ \mathit{True}
\end{aligned}$$

Note the duplication of the term involving *newRef* and the fact that a new reference containing *id* will be allocated at each occurrence. The re-evaluation

of polymorphic terms corresponds with *polymorphism-by-name* [Ler93]. Also note that the first reference will be updated, but only the second one will be read. Admittedly, the behavior of this expression could be confusing to the programmer, but allowing it does not make our system unsound. Demonstration of unsoundness would require that an expression was well typed, not a value, and could not be reduced further. This expression can be reduced to *True*, and is not a problem in this respect.

Although polymorphism-by-name keeps our System-F style *core* language sound in the presence of polymorphic update, we expect it to be too confusing for the programmer to use in practice. If a value appears to be shared in the source program, then we do not want this sharing to be reduced depending on whether it is assigned a polymorphic type by the type inferencer. As in [GL86] we restrict generalisation to preserve the data sharing properties of programs during translation to and from core. However, as mentioned earlier we don't want to use the value restriction. The next section discusses the possibility of leveraging effect typing to control generalisation, but as we shall see in §2.5 we use another method, namely *closure typing*, to achieve this. Closure typing will help us deal with the problem with polymorphic update, as well as more accurately reason about the sharing properties of data.

2.4.4 Restricting generalisation with effect typing

As we don't want to rely on the value restriction to control generalisation, we must find another way of identifying variables that are free in the store typing. In a language with ML-style references, the sole means of extending the store is by explicitly allocating them with a function such as *newRef*. In this case, the problem of identifying variables free in the store typing reduces to identifying calls to *newRef* and collecting the types of values passed to it. If we treat reference allocation as a computational effect, then we can use effect inference to perform the collection [Wri92]. The rules for the polymorphic type and effect system [TJ92a, TJ92b] are as follows:

$$\boxed{\Gamma \vdash t :: \tau \;;\; \sigma}$$

$$\Gamma,\ x : \tau \vdash x :: \mathrm{Inst}(\tau) \;;\; \emptyset \tag{Var}$$

$$\frac{\Gamma,\ x : \tau_1 \vdash t_2 :: \tau_2 \;;\; \sigma}{\Gamma \vdash \lambda(x : \tau_1).\ t_2 :: \tau_1 \xrightarrow{\sigma} \tau_2 \;;\; \emptyset} \tag{Abs}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 :: \tau_{11} \xrightarrow{\sigma} \tau_{12} \;;\; \sigma_1 \\ \Gamma \vdash t_2 :: \tau_{11} \;;\; \sigma_2\end{array}}{\Gamma \vdash t_1\ t_2 :: \tau_{12} \;;\; \sigma_1 \cup \sigma_2 \cup \sigma} \tag{App}$$

$$\frac{\begin{array}{r}\Gamma \vdash t_1 :: \tau_1 \;;\; \sigma_1 \\ \Gamma,\ x : \mathrm{Gen}(\sigma_1, \Gamma, \tau_1) \vdash t_2 :: \tau_2 \;;\; \sigma_2\end{array}}{\Gamma \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 :: \tau_2 \;;\; \sigma_1 \cup \sigma_2} \tag{Let}$$

$$\frac{\Gamma \vdash t :: \tau \;;\; \sigma_1 \quad \sigma_1 \subseteq \sigma_2}{\Gamma \vdash t :: \tau \;;\; \sigma_2} \tag{Sub}$$

The judgement $\Gamma \vdash e :: \tau \; ; \; \sigma$ reads: In the environment Γ the expression e has type τ and effect σ. The environment Γ maps variables to types. In this presentation effects are gathered together with the set union operator $\cup$ and a pure expression is assigned the effect $\emptyset$. Also note the term $\mathrm{Gen}(\sigma_1, \Gamma, \tau_1)$ in the rule (Let). Generalisation is restricted to variables that are not free in either the type environment or the effect caused by evaluating the body of a let-binding.

These rules describe the "plumbing" of how effects are attached to types. What's missing is a description of how atomic effects are introduced to the system via *newRef*.

In Wright's system [Wri92], *newRef* is given the type:

$$newRef :: \forall a\ e.\ a \xrightarrow{a\ e} Ref\ a$$

Here, a is a type variable and its presence on the function arrow indicates that it has the effect of allocating a new reference containing a value of that type. The effect variable e combined with the subsumption rule (Sub) allows the function to be treated as causing *any* effect so long as it includes a. This is used when passing arguments to higher order functions, which is discussed in §2.3.6. Although this system collects the requisite type variables, it is limited by the fact that *all* the effects caused by the allocation of references appear in a function's type, even if they are used entirely locally to that function.

For example, with the following program:

$$\begin{array}{ll} id & :: \forall a\ e.\ a \xrightarrow{e} a \\ id & = \lambda x.\ x \\ \\ idRef & :: b \xrightarrow{b} b \\ idRef & = (\lambda x.\ \mathbf{let}\ \ ref = newRef\ \ x\ \mathbf{in}\ \ readRef\ \ ref) \end{array}$$

In the type for *id*, we can generalise a because it does not appear free in either the type environment or the effect caused by the function. The second function *idRef* behaves identically to *id*, except that it creates a reference to its argument before returning it. Even though this reference is not accessible once *idRef* returns, the effect caused by allocating it appears in its type, which prevents b from being generalised. Although these two functions behave identically from a *callers* point of view, they cannot be used interchangeably as they have different types.

2.4.5 Observation criteria

In [TJ92b] Talpin and Jouvelot extend Wright's effect system with regions. As in Disciple, their regions denote sets of locations which may alias, but they attach region variables to reference types only. In their system, effects are caused when references are read and written to, but also when they are allocated. Each effect carries with it the type of the reference being acted upon, as well as the region it is contained within. The types of the primitive operators are:

$$\begin{array}{lll} newRef & :: & \forall a\ r\ e.\ (a \xrightarrow{e} Ref\ \ r\ a) \rhd e \supseteq Init\ \ r\ a \\ readRef & :: & \forall a\ r\ e.\ (Ref\ \ r\ a \xrightarrow{e} a) \rhd e \supseteq Read\ \ r\ a \\ writeRef & :: & \forall a\ r\ e.\ (Ref\ \ r\ a \xrightarrow{e_1} a \xrightarrow{e_2} ()) \ \rhd e_2 \supseteq Write\ \ r\ a \end{array}$$

Similarly to Wright's system, *newRef* can be treated has having *any* effect, so long as that effect includes *Init* r a. The effect *Init* r a records that a reference in region r was initialised to contain a value of type a. However, in this system the subsumption rule (Sub) is modified to include an observation criterion which allows effects which are not visible to a caller to be masked.

$$\frac{\Gamma \vdash e :: \tau \; ; \; \sigma_1 \quad \sigma_2 \supseteq \mathit{Observe}(\Gamma, \sigma_1, \tau)}{\Gamma \vdash e :: \tau \; ; \; \sigma_2} \qquad \text{(Sub-Obs)}$$

where

$$\begin{aligned}&\mathit{Observe}(\Gamma, \tau, e)\\ &\quad = \{\ \mathit{Init}\ r\ \tau_1,\ \mathit{Read}\ r\ \tau_1,\ \mathit{Write}\ r\ \tau_1 \in \sigma \mid r \in \mathit{fr}(\Gamma) \cup \mathit{fr}(\tau)\ \}\\ &\quad \cup \{\ \varsigma \in \sigma \mid \varsigma \in \mathit{fv}(\Gamma) \cup \mathit{fv}(\tau)\ \}\end{aligned}$$

The function *fv* computes the free type, region and effect variables in its argument, while *fr* returns free region variables only.

With this system, when we type check the definition of *idRef* the body of the λ-expression yields the statement:

$$\begin{aligned}\Gamma,\ x :: a \quad &\vdash (\textbf{let}\ \mathit{ref} = \mathit{newRef}\ x\ \textbf{in}\ \mathit{readRef}\ r) :: a\\ &\ ;\ \mathit{Init}\ r_1\ a\ \cup\ \mathit{Read}\ r_1\ a\end{aligned}$$

Note that although the right of the let-binding allocates and then reads a reference, the region into which the reference is allocated is entirely local to the binding. Applying (Sub-Obs) yields:

$$\begin{aligned}\Gamma,\ x :: a \quad &\vdash (\textbf{let}\ \mathit{ref} = \mathit{newRef}\ x\ \textbf{in}\ \mathit{readRef}\ r) :: a\\ &\ ;\ \emptyset\end{aligned}$$

This allows us to infer the same type for *id* as we do for *idRef*. Leaving the formal proof of soundness in [TJ92b], we can see why the observation criteria works by inspecting our original statement:

$$\begin{aligned}\Gamma,\ x :: a \quad &\vdash (\textbf{let}\ \mathit{ref} = \mathit{newRef}\ x\ \textbf{in}\ \mathit{readRef}\ r) :: a\\ &\ ;\ \mathit{Init}\ r_1\ a\ \cup\ \mathit{Read}\ r_1\ a\end{aligned}$$

Notice that r_1 is not present in the type environment, and is therefore not visible to the expression's calling context. Also, as the type of the expression does include r_1 there will be no "handle" on this region after the expression has finished evaluating. Indeed, as the allocated reference is unreachable after this evaluation, it could be safely garbage collected. Returning to Tofte's (un)proof in §2.4.2, this garbage collection corresponds to removing the associated binding from the store and its store typing, which allows a to be safely generalised.

2.4.6 Effect typing versus arbitrary update

Talpin and Jouvelot's system works well for a language with ML-style references. As update is limited to a distinguished *Ref* type, it is easy for the type system to decide when to introduce *Read*, *Write* and *Init* effects. However, in Disciple, update is not restricted to data of a special type. In our system all data has the potential to be updated.

For example the simple data type *Maybe* is defined as follows:

$$\textbf{data}\ \mathit{Maybe}\ r\ a\ = \mathit{Nothing}\ |\ \mathit{Just}\ a$$

Defining this type furnishes us with a data constructor which can be used to allocate a *Just*.

$$\mathit{Just} :: \forall\ a\ r.\ a \rightarrow \mathit{Maybe}\ r\ a$$

Once a *Just* has been allocated, we can then use the field projection syntax of §2.7 to update it, or not, as we see fit. If we were to use an effect system to control generalisation, how would we know whether this constructor should cause an *Init* effect? Only the allocation of a mutable value should cause an effect, but mutability depends on whether or not the value may ever be updated, not *vice versa*. The mutability of the object will be inferred by our type system, but this property is not immediately visible at the point where the object is allocated.

2.5 Closure typing

Leroy's *closure-typing* [LW91, Ler92] is a system for modeling data sharing due to the inclusion of free variables in the bodies of functions. We use closure typing as an alternate solution to the problem of polymorphic update, as well as to reason about the sharing properties of regions.

Consider the following function:

$$addT = \lambda x.\ \lambda y.\ (x + y,\ x)$$

If we take addition (+) to operate on values of type *Int*, then we could give $addT$ the following type:

$$addT :: Int \rightarrow Int \rightarrow Pair\ \ Int\ \ Int$$

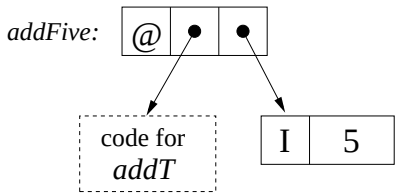
If we then partially apply $addT$ to the value 5, the first argument of its type is satisfied and we end up with a function that accepts an integer and produces a pair:

$$addFive :: Int \rightarrow Pair\ \ Int\ \ Int$$
$$addFive = addT\ \ 5$$

addFive can be further applied to yield the pair, but what happened to the first value we provided? Assume evaluation proceeds via template instantiation after the pure lambda calculus. In this case we could reason that the argument 5 was bound to the formal parameter x and then substituted into the body of the outer lambda abstraction, that is:

$$\begin{aligned} addFive &\longrightarrow addT\ \ 5 \\ &\longrightarrow (\lambda x.\ \lambda y.\ (x + y,\ x))\ 5 \\ &\longrightarrow \lambda y.\ (5 + y,\ 5) \end{aligned}$$

This call-by-name reasoning is applicable to a pure language such as Haskell, but as we intend to use destructive update we must take a more operational approach. If we instead consider an implementation based on super-combinators [Hug83], we would treat $addT$ as a single supercombinator, and the partial application $(addT\ \ 5)$ as the construction of a thunk containing pointers to the supercombinator code and argument value:



When *addFive* is applied to its final argument, the code for $addT$ is called directly. $addT$'s first argument comes from the thunk, and the second is supplied by the application. This is how DDC operates.

With this system, every use of *addFive* shares the same '5' object. Using region annotations on data types and closure annotations on functions[7], we give *addFive* a type which makes the sharing explicit.

[7] We modify Leroy's syntax for closures to be similar to the one used for effects.

$$\begin{array}{lll} \mathit{addFive} & :: & \forall r_1\ r_2\ r_3 \\ & . & \mathit{Int}\ r_1 \xrightarrow{c_1} \mathit{Pair}\ r_2\ (\mathit{Int}\ r_3)\ (\mathit{Int}\ r_4) \\ & \rhd & c_1 = x : \mathit{Int}\ r_4 \end{array}$$

On the left of the function arrow, the argument type $\mathit{Int}\ r_1$ says that *addFive* accepts an integer from a region which we name r_1. On the right of the arrow, we see that the function produces a pair of integer components. As r_3 is quantified, we infer that the first component has been freshly allocated into a new region (addition returns a fresh result). The data constructor representing the pair is also fresh, so r_2 is quantified as well. On the other hand, r_4 is *not* quantified, which indicates that the second component of the pair will be in the same region each time *addFive* is called.

The closure variable c_1 attached to the function arrow indicates that the definition of *addFive* creates a shared value, and the term $x : \mathit{Int}\ r_4$ records its type. The "x :" portion of $x : \mathit{Int}\ r_4$ is called the *closure tag*, and we treat it as an operator that lifts the type term $\mathit{Int}\ r_4$ into a closure term. In this example, the variable x corresponds to the occurrence that is free in the innermost lambda-abstraction in the definition of *addFive*. For the types of primitive functions such as data constructors, although there is no associated source code we still use names such as x, y, z for consistency.

Note that our type system tracks variable names such as x as a notational convenience, but does not make use of them for checking purposes. We have found it useful for such variables to be included in the types presented to the user, as without them it can be very difficult to determine why an inferred type signature includes a particular closure term. However, if desired we could replace all such variables with an underscore to indicate they are ignored by the type system proper.

2.5.1 Dangerous type variables

In [Ler92] Leroy defines dangerous variables to be the ones that are free in a live reference type. For Disciple this is equivalent to being free under a mutable type constructor.

Consider the following type:

$$\begin{array}{lll} \mathit{thing} & :: & \mathit{Maybe}\ r_1\ (a \to a) \\ & \rhd & \mathit{Mutable}\ r_1 \end{array}$$

with

$$\begin{array}{ll} \textbf{data}\ \mathit{Maybe}\ r_1\ a & \\ \quad = & \mathit{Nothing} \\ \quad \mid & \mathit{Just}\ \{x :: a\} \end{array}$$

In the type of *thing*, a is dangerous because it corresponds to a value that we are able to update at runtime. For example, the following code creates a *Just* constructor containing the function *id*, updates it to hold the less general function *succ*, then tries to apply that function to a string. This example is similar to one from §2.4, except that we are using the Disciple projection syntax to update the mutable object. The term $\mathit{thing}\ \odot_{\#}\ x$ creates a reference to the x field in the *Just* constructor. If we view references as being akin to pointers, then $\mathit{thing}\ \odot_{\#}\ x$ has a similar meaning to the C expression `&(thing.x)`. Projections are discussed further in §2.7.

Once the reference is created, we use the $:=_{\#}$ operator to update the field (via the reference):

$$\begin{array}{lll} \textbf{do} & \mathit{thing} & = \mathit{Just}\ \mathit{id} \\ & \mathit{thing}\ \odot_{\#}\ x & :=_{\#}\ \mathit{succ} \\ & \mathit{trouble} & = \textbf{case}\ \mathit{thing}\ \textbf{of}\ \mathit{Just}\ f \rightarrow\ f\ \text{“}\texttt{die!}\text{”} \end{array}$$

When generalising the type of *thing* we must hold a monomorphic. If we were to give it the following polymorphic type, then the above code would pass the type checker, but would have an undefined result at runtime.

$$\begin{array}{lll} \mathit{thing}_{bad} & :: & \forall a.\ \mathit{Maybe}\ r_1\ (a \rightarrow a) \\ & \rhd & \mathit{Mutable}\ r_1 \end{array}$$

In general, to determine the dangerous variables in a type we must inspect the definitions of the data types involved.

For example, the following type has three separate region variables, which gives us three places to attach mutability constraints:

$$\begin{array}{l} \textbf{data}\ \mathit{TwoThings}\ r_1\ r_2\ r_3\ a\ b \\ \quad = \quad \mathit{Thing1}\ (\mathit{Maybe}\ r_2\ a) \\ \quad \mid \quad \mathit{Thing2}\ (\mathit{Maybe}\ r_3\ b) \end{array}$$

Here is an example signature that uses *TwoThings*:

$$\begin{array}{lll} \mathit{foo} & :: & \mathit{TwoThings}\ r_4\ r_5\ r_6\ (d \rightarrow \mathit{Int}\ r_7)\ (\mathit{Char}\ r_8) \\ & \rhd & \mathit{Const}\ r_4 \\ & , & \mathit{Mutable}\ r_5 \\ & , & \mathit{Const}\ r_6 \end{array}$$

In this case, as r_5 is mutable we must hold d and r_7 monomorphic. Note that region, effect and closure variables can be dangerous as well. As r_5 is mutable we cannot generalise r_7 because the associated *Maybe* object might be updated to hold a function that does not allocate a fresh return value. On the other hand, we can allow r_8 to be polymorphic as r_6 is constant. If r_4 was mutable then all of r_5, r_6, d, r_7 and r_8 would have to be monomorphic, because this would let us update either of the *Thing* objects to hold a different *Maybe*.

A formal description of which variables are dangerous is given in §3.2.

2.5.2 Closure typing and hidden communication

Consider the following function, also from [Ler92]:

$$\begin{array}{l} \mathit{makeGetSet}\ x \\ \quad \textbf{do} \quad \mathit{ref} \quad\ = \mathit{newRef}\ x \\ \qquad\ \ \mathit{get}\ () \quad = \mathit{readRef}\ \mathit{ref} \\ \qquad\ \ \mathit{set}\ z \quad = \mathit{writeRef}\ \mathit{ref}\ z \\ \qquad\ \ \mathit{Pair}\ \mathit{get}\ \mathit{set} \end{array}$$

This function allocates a reference to the supplied value x, and then returns a pair of functions to get and set the value in the reference.

In Disciple, without closure information and before generalisation, the type of *makeGetSet* is:

$$\begin{array}{lll} \mathit{makeGetSet} & :: & a \to \mathit{Pair}\ r_1\ (() \xrightarrow{e_1} a)\ (a \xrightarrow{e_2} ()) \\ & \rhd & e_1 \sqsupseteq \mathit{Read}\ r_2 \\ & , & e_2 \sqsupseteq \mathit{Write}\ r_2 \\ & , & \mathit{Mutable}\ r_2 \end{array}$$

There are two problems with this type. Firstly, as r_2 is not mentioned in the body (or the type environment), the read and write effects will be masked as per §2.3.7. This is invalid because the order in which these applications take place at runtime certainly matters, so they must retain their effect terms. Secondly, if we were to allow a to be generalised then we would have an unsound system once again.

For example:

$$\mathit{makeGetSet}_{bad} :: \forall a\ r_1.\ a \to \mathit{Pair}\ r_1\ (() \to a)\ (a \to ())$$

```
broken ()
  do getset  = makeGetSet_bad id
     set2    = snd getset
     set2 succ
     get2    = fst getset
     get2 () "die!"
```

By allowing the mutable object *ref* to be free in the closure of the get and set functions, we have created a communication channel between them that is not visible in their types. This is not a problem in itself, but the addition of let-polymorphism allows each function to gain a different understanding of what type of data is being sent across the channel. Note that with the bad type for *makeGetSet*, the inferred type of *getset* includes a quantifier for b:

$$\mathit{getset} :: \forall b.\ \mathit{Pair}\ r_1\ (() \to (b \to b))\ ((b \to b) \to ())$$

As we have used a let-binding to define *get2* and *set2*, the types of the two components are re-generalised and we end up with:

$$\begin{array}{l} \mathit{get2} :: \forall c.\ () \to (c \to c) \\ \mathit{set2} :: \forall d.\ (d \to d) \to () \end{array}$$

The use of *set2* updates the shared reference to contain a function, *succ*, that only accepts integers. Unfortunately, with *get2* we can then read it back and pretend that it accepts a string.

Adding closure information to our types remedies this problem. Here is the new type of *makeGetSet*, with closure information, and before generalisation:

$$\begin{array}{lll} \mathit{makeGetSet} & :: & a \to \mathit{Pair}\ r_1\ (() \xrightarrow{e_1\ c_1} a)\ (a \xrightarrow{e_2\ c_2} ()) \\ & \rhd & e_1 \sqsupseteq \mathit{Read}\ r_2 \\ & , & e_2 \sqsupseteq \mathit{Write}\ r_2 \\ & , & c_1 \sqsupseteq \mathit{ref} : \mathit{Ref}\ r_2\ a \\ & , & c_2 \sqsupseteq \mathit{ref} : \mathit{Ref}\ r_2\ a \\ & , & \mathit{Mutable}\ r_2 \end{array}$$

The constraints on c_1 and c_2 show that the get and set functions returned in the pair can access a shared mutable value, and that the type of this value

contains a variable a. Note that the lattice structure for closures is identical to that for effects, which was discussed in §2.3.1. For closures we take $\sqsupseteq$ as being a synonym for the superset operator $\supseteq$, and use $\vee$ as a synonym for $\cup$. There is no $\top$ element for closures, but we stick with the lattice notation for consistency with effect types.

Returning to *makeGetSet*, note that this function allocates the reference itself. This can be determined from the fact that the primary region variable of the reference, r_2, is not reachable from the closure annotation on the outermost (leftmost) function arrow. Once we apply *makeGetSet* to its argument, the reference is created and subsequently shared. In our example this is done in the binding for *getset*.

After generalisation, the new type of *getset* is:

$$\begin{array}{lll} \mathit{getset} & :: & \forall e_1\ e_2\ c_1\ c_2 \\ & . & \mathit{Pair}\ r_1\ (() \xrightarrow{e_1\ c_1} (b \to b))\ ((b \to b) \xrightarrow{e_2\ c_2} ()) \\ & \rhd & e_1 \sqsupseteq \mathit{Read}\ r_2 \\ & , & e_2 \sqsupseteq \mathit{Write}\ r_2 \\ & , & c_1 \sqsupseteq \mathit{ref} : \mathit{Ref}\ r_2\ (b \to b) \\ & , & c_2 \sqsupseteq \mathit{ref} : \mathit{Ref}\ r_2\ (b \to b) \\ & , & \mathit{Mutable}\ r_2 \end{array}$$

In this type we have two "outermost" function arrows, which are the two in the *Pair*. The type $(b \to b)$ is in the closure of these outermost functions, and the type variable b lies underneath a region variable r_2 that is constrained to be *Mutable*. This means that b is dangerous and cannot be generalised. Note that r_2 is not generalised either, though this restriction is due to the fact that r_2 is present in the outermost closure. This point is discussed in the next section.

Returning to our example, after performing the *fst* and *snd* projections, our new types for *get2* and *set2* are:

$$\begin{array}{lll} \mathit{get2} & :: & \forall e_1\ c_1.\ () \xrightarrow{e_1\ c_1} (b \to b) \\ & \rhd & e_1 \sqsupseteq \mathit{Read}\ r_2 \\ & , & c_1 \sqsupseteq \mathit{ref} : \mathit{Ref}\ r_2\ (b \to b) \\ & , & \mathit{Mutable}\ r_2 \end{array}$$

$$\begin{array}{lll} \mathit{set2} & :: & \forall e_2\ c_2.\ (b \to b) \xrightarrow{e_2\ c_2} () \\ & \rhd & e_2 \sqsupseteq \mathit{Write}\ r_2 \\ & , & c_2 \sqsupseteq \mathit{ref} : \mathit{Ref}\ r_2\ (b \to b) \\ & , & \mathit{Mutable}\ r_2 \end{array}$$

The effect information in the types of these functions ensures that uses of them will not be reordered during optimisation. The closure annotations capture the fact that they can communicate via a shared mutable value, and ensures that both functions agree on its type.

2.5.3 Material regions and sharing

Recall from section §2.2.3 that *material* region variables are the ones that represent objects that are shared between all uses of a bound variable. For example:

$$\begin{array}{l} \mathit{five} :: \mathit{Int}\ \ r \\ \mathit{five} = 5 \end{array}$$

Here, r is clearly material, because every use of *five* references the same '5' object. On the other hand, consider:

$$\begin{array}{l} \mathit{addTwo} :: \forall r_1\ r_2.\ \mathit{Int}\ \ r_1 \rightarrow \mathit{Int}\ \ r_2 \\ \mathit{addTwo}\ \ x\ \ \ = \mathit{succ}\ (\mathit{succ}\ \ x) \end{array}$$

Neither r_1 or r_2 are material in the type of *addTwo*. These variables represent the locations of objects passed to, and returned from, the function. They do not represent locations of objects that are shared between uses of it. Without further information, we take regions in the argument positions of function types to be immaterial.

Note that with the constructors at hand, we cannot be sure that no *function objects* are shared between calls to *addTwo*. If *succ* was defined as the partial application of some more primitive function, then every use of *succ* would refer to the same thunk. However, for our purposes sharing only matters if the shared objected has the potential to be destructively updated, and thunks cannot be updated.[8]

The following example defines a function that references a shared data object:

$$\begin{array}{l} \mathit{makeFive}\ () \\ = \mathbf{do}\ \ \ \ x\ \ \ \ \ \ \ \ \ \ = 5 \\ \qquad\ \ \ \mathit{retFive}\ ()\ \ \ = x \\ \qquad\ \ \ \mathit{retFive} \end{array}$$

If we wrote down a type for *makeFive* which included region variables but not closure information then we would have:

$$\mathit{makeFive} :: \forall r.\ () \rightarrow () \rightarrow \mathit{Int}\ \ r$$

As r is quantified, *makeFive* should return a freshly allocated *Int* object each time it is called. This is certainly true if we apply both arguments, but we can invalidate the meaning of the quantifier by supplying only one. To see this more clearly, consider the supercombinator translation:

$$\begin{array}{l} \mathit{makeFive}'\ () \\ = \mathbf{do}\ \ \ \ x\ \ \ \ \ \ \ = 5 \\ \qquad\ \ \ \mathit{retFive}'\ \ x \\ \\ \mathit{retFive}'\ \ x'\ ()\ \ \ = x' \end{array}$$

makeFive$'$ and *retFive*$'$ are the result of lambda-lifting [Joh85] our original function. Note that the free variable in the definition of *retFive* is passed explicitly to its lifted version. As *makeFive*$'$ returns the value *retFive*$'$ x, which evaluates to a thunk, the same '5' object will be returned each time *makeFive*$'$ is provided with its final argument.

[8] They can be overwritten by the runtime system during lazy evaluation, but this is not visible in the programming model.

Consider then a binding that partially applies *makeFive*:

$$\begin{array}{l} \mathit{makeFiveUnit} :: \forall r.\ () \to \mathit{Int}\ \ r \\ \mathit{makeFiveUnit} = \mathit{makeFive}\ () \end{array}$$

Although the type of *makeFiveUnit* says that its return value should be freshly allocated, we have just seen that the evaluation of *makeFive* () will produce a function that returns the same '5' object every time. Following the standard restriction for generalisation, we have not quantified over variables free in the type environment. This environment consists of the type of *makeFive*, which has no free variables, so that does not help us here. The standard restriction prevents types from becoming out of sync with their context, but it does not model sharing due to free variables in the body of function definitions.

Once again, closure typing comes to our rescue. When we include closure information, the types of *makeFive* and *makeFiveUnit* become:

$$\begin{array}{lll} \mathit{makeFive} & :: & \forall r.\ () \to () \xrightarrow{c} \mathit{Int}\ \ r \\ & \rhd & c = x : \mathit{Int}\ \ r \\ \mathit{makeFiveUnit} & :: & () \xrightarrow{c} \mathit{Int}\ \ r \\ & \rhd & c = x : \mathit{Int}\ \ r \end{array}$$

The type of *makeFive* now includes the fact that when the first () is applied, it allocates an object in a region it names r, and this object is shared by all calls to the returned function.

The type of *makeFiveUnit* preserves this sharing information. Region variables that are reachable from the closure annotation on the outer most function arrow of a type are material, and material region variables are not generalised.

2.5.4 Material regions and algebraic data types

When we come to generalise the type of a binding, and the type contains only simple constructors like *Int* and $\to$, then we can determine which region variables are material directly from the type. However, when dealing with algebraic data types, we also need their definitions.

Consider the following:

$$\begin{array}{lll} \textbf{data}\ \mathit{IntFun}\ \ r_{1..4}\ e_1\ c_1 & & \\ \quad = & \mathit{SInt}\ (\mathit{Int}\ \ r_2) & \\ \quad \mid & \mathit{SFun}\ (\mathit{Int}\ \ r_3 \xrightarrow{e_1\ c_1} \mathit{Int}\ \ r_4) & \end{array}$$

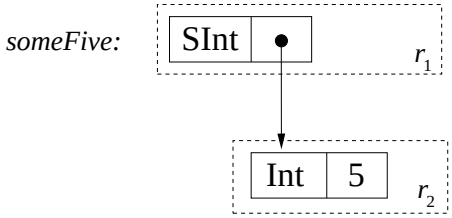
This definition implicitly generates the following constructors. Note that we use $r_1..r_4.$ as shorthand for $r_1\ r_2\ r_3\ r_4$. Also, r_1 is used as the primary region variable of the type, but is not present in the types of the constructor arguments.

$$\begin{array}{lll} \mathit{SInt} & :: & \forall r_{1..4}\ e_1\ c_1 \\ & . & \mathit{Int}\ \ r_2 \to \mathit{IntFun}\ \ r_{1..4}\ e_1\ c_1 \\ \mathit{SFun} & :: & \forall r_{1..4}\ e_1\ c_1 \\ & . & (\mathit{Int}\ \ r_3 \xrightarrow{e_1\ c_1} \mathit{Int}\ \ r_4) \to \mathit{IntFun}\ \ r_{1..4}\ e_1\ c_1 \end{array}$$

The *SInt* constructor creates an object containing a pointer to an *Int*. The region variable r_1 is primary as it is first in the list, so we take the outer *SInt*

constructor to be in this region. The *Int* component is in region r_2, so an application of *SInt* would produce:

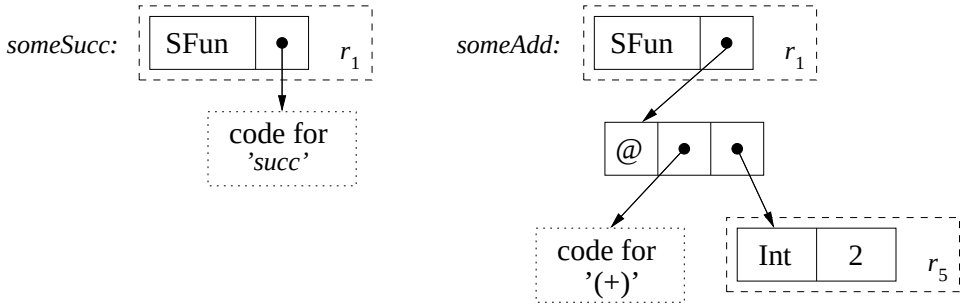
$$\mathit{someFive} \quad = \mathit{SInt}\ 5$$



As the outer constructor always appears in the primary region, the primary region variable is material. Because an *SInt* object contains an *Int* in a region named r_2, this variable is also material.

On the other hand, when we use *SFun*, the constructed object will contain a pointer to either the code for the function argument, or a thunk, depending on whether the argument was partially applied:

$$\mathit{someSucc} \quad = \mathit{SFun}\ \ \mathit{succ} \qquad\qquad \mathit{someAdd} \quad = \mathit{SFun}\ ((+)\ 2)$$



Note that with the constructors at hand, there no way to create an *IntFun* object that actually includes data in the r_3 or r_4 regions. Because of this, they are immaterial, and the generalisation of immaterial regions is not restricted as per the previous section. The type of *someSucc* above is:

$$\begin{array}{lll} \mathit{someSucc} & :: & \forall r_3\ r_4\ e_1.\ \mathit{IntFun}\ r_{1..4}\ e_1\ \bot \\ & \rhd & e_1 \quad \sqsupseteq \mathit{Read}\ r_3 \end{array}$$

Note that although our *someSucc* object does not include data in region r_2, that region is not quantified here. In general, if a particular value has type $\mathit{IntFun}\ r_{1..4}\ e_1\ c_1$ then we will not know what data constructor was used to create it. We must rely on the data type definition to determine which regions are material.

The type of *someAdd* is similar, except that its closure variable is constrained to contain the type of the argument in the partial application of (+):

$$\begin{array}{lll} \mathit{someAdd} & :: & \forall r_3\ r_4\ e_1\ c_1.\ \mathit{IntFun}\ r_{1..4}\ e_1\ c_1 \\ & \rhd & e_1 \quad \sqsupseteq \mathit{Read}\ r_3 \vee \mathit{Read}\ r_5 \\ & , & c_1 \quad \sqsupseteq \mathit{Int}\ r_5 \end{array}$$

The material regions of a type are defined formally in §3.2.4.

2.5.5 Strong, mixed and absent region variables

When an algebraic data type is defined we do not restrict the ways in which region variables are used. Due to this, a particular variable may occur in both a material and immaterial position. For example:

$$\begin{array}{lll}\textbf{data}\ \mathit{IntFunMixed}\ r_{1..5}\ e_1\ c_1 & & \\ \quad = & \mathit{SIntX} & (\mathit{Int}\ r_2) \\ \quad | & \mathit{SCharX} & (\mathit{Int}\ r_4) \\ \quad | & \mathit{SFunX} & (\mathit{Int}\ r_2 \xrightarrow{e_1\ c_1} \mathit{Int}\ r_3)\end{array}$$

In the first constructor, r_2 is used as the primary region variable of *Int*, which makes it material. In the third constructor, r_2 is used as part of the type of a function parameter, so it is also immaterial. In this situation we say that r_2 is *mixed material*.

If a region variable is *only* ever used in a material position, then it is *strongly material*. In the above definition, r_1 is strongly material because it is used as the primary region variable for *IntFunMixed*, and not in the type of a function parameter. The variable r_4 is also strongly material. We will use this concept when we discuss the polymorphic copy function in §2.6.

If a region variable is present as a parameter of the type constructor being defined, but not one of the data constructors, then we say it is *absent*. The variable r_5 is absent in the above definition. As absent region variables cannot correspond to real regions in the store, all absent variables are also immaterial. The reverse is not true, as r_3 is immaterial, but not absent.

2.5.6 Pure effects and empty closures

In the previous two sections, the definitions of *IntFun* and *IntFunMixed* include effect and closure variables as arguments to the type constructor. This allows these data types to be polymorphic in the effect and closure of the contained function. Alternatively, we could omit these variables as long as we constrained the types of *SFun* and *SFunX* so that the effect of the contained function was pure, and its closure contained no elements.

A closure that has no elements is said to be *empty*. Emptiness of closures is related to purity of effects. Recall from §2.3.9 that a pure effect is written $\bot$ and we can require an effect to be pure with the *Pure* constraint. Likewise, we write empty closures as $\bot$ and require a closure to be empty with the *Empty* constraint. We sometimes annotate $\bot$ with its kind, such as $\bot_!$ and $\bot_\$$ to distinguish between its two readings, but the kind is usually clear from context.

By omitting effect and closure variables, and restricting ourselves to a single region we will now define a diet version of *IntFun* that has a single parameter instead of six. This new data type can still contain an *Int* or function value, but the set of functions it could hold is reduced:

$$\begin{array}{lll}\textbf{data}\ \mathit{IntFunDiet}\ r_1 & & \\ \quad = & \mathit{SIntD} & (\mathit{Int}\ r_1) \\ \quad | & \mathit{SFunD} & (\mathit{Int}\ r_1 \to \mathit{Int}\ r_1)\end{array}$$

This modified data type definition generates the following constructors:

$$
\begin{array}{lll}
\mathit{SIntD} & :: & \forall r_1.\ \mathit{Int}\ r_1 \to \mathit{IntFunDiet}\ r_1 \\
\\
\mathit{SFunD} & :: & \forall r_1\ e_1\ c_1 \\
 & . & (\mathit{Int}\ r_1 \xrightarrow{e_1\ c_1} \mathit{Int}\ r_1) \to \mathit{IntFunDiet}\ r_1 \\
 & \rhd & \mathit{Pure}\ e_1 \\
 & , & \mathit{Empty}\ c_1
\end{array}
$$

Note that although r_1 is repeated in the first parameter of *SFunD*, this doesn't require its argument to be a function which simply passes the *Int* through unchanged. The type of a function like *succ* can be instantiated so that both its region variables are the same. Due to this we can still construct (*SFunD succ*) as per the figure in §2.5.4, though the single region will be forced *Const* due to purification of the function's *Read* effect. On the other hand, we can no longer construct (*SFunD* ((+) 2)) as its type would include a closure term due to the partial application, rendering it non-empty. See §5.2.4 for a possible way of addressing this limitation.

2.5.7 Closure trimming

The closure annotation attached to a function type lists the types of all free variables in that function's definition. However, not all of this information is useful to our analysis. As we only restrict the generalisation of material region variables, we only need to retain closure terms that contain them. The rest of the closure information can be trimmed out, and doing so is an important optimisation in practice.

Consider the following program:

$$
\begin{array}{lll}
x & & = 5 \\
\mathit{fun} & () & = x + 1 \\
\mathit{fun2} & () & = \mathit{fun} \\
\mathit{fun3} & () & = \mathit{fun2} \\
\mathit{fun4} & () & = \mathit{fun3}
\end{array}
$$

This is a simple program, but as each successive binding refers to the binding above it, the closure terms in their types can become very large.

If x has type $\mathit{Int}\ r_1$, then *fun* has the following signature:

$$
\begin{array}{lll}
\mathit{fun} & :: & \forall r_2.\ () \xrightarrow{e_1\ c_1} \mathit{Int}\ r_2 \\
 & \rhd & e_1 = \mathit{Read}\ r_1 \\
 & , & c_1 = x : \mathit{Int}\ r_1
\end{array}
$$

This says that *fun* accepts a unit value and produces a freshly allocated integer. The closure constraint $c_1 = x : \mathit{Int}\ r_1$ says that the function refers to this object via the free variable x. When it evaluates, the addition operator reads the integer bound to x, hence the $\mathit{Read}\ r_1$ effect. It also reads the constant integer 1, but as this constant is local to the function the effect is masked.

Here is the type for *fun2* :

$$\begin{array}{lll} \mathit{fun2} & :: & \forall r_2.\ () \xrightarrow{c_2} () \xrightarrow{e_1\ c_1} \mathit{Int}\ \ r_2 \\ & \rhd & e_1 \quad = \mathit{Read}\ \ r_1 \\ & , & c_1 \quad = x : \mathit{Int}\ \ r_1 \\ & , & c_2 \quad = (\mathit{fun} \quad : \quad \forall r_3.\ () \xrightarrow{e_3\ c_3} \mathit{Int}\ \ r_3 \\ & & \qquad\quad \rhd \quad e_3 \quad = \mathit{Read}\ \ r_1 \\ & & \qquad\quad , \quad c_3 \quad = x : \mathit{Int}\ \ r_1) \end{array}$$

Note that *fun2* refers to *fun*, so the full type of *fun* appears in its closure. However, as we only use closure terms to reason about the sharing properties of data, we gain no benefit from carrying around information about the effects associated with a variable like *fun*. We also gain no benefit from retaining its argument and return types. We extend the concept of materiality to value types, and say that the argument and return positions of functions are immaterial because they do not represent objects in the store. Lastly, if we erase the return type $\mathit{Int}\ \ r_3$ then we do not need the quantifier $\forall r_3$. The only information about *fun* that we *do* need to keep is that it references a material object of type $\mathit{Int}\ \ r_1$. Using these observations we trim the type of *fun2* to get:

$$\begin{array}{lll} \mathit{fun2} & :: & \forall r_2.\ () \xrightarrow{c_2} () \xrightarrow{e_1\ c_1} \mathit{Int}\ \ r_2 \\ & \rhd & e_1 \quad = \mathit{Read}\ \ r_1 \\ & , & c_1 \quad = x : \mathit{Int}\ \ r_1 \\ & , & c_2 \quad = \mathit{fun} : \mathit{Int}\ \ r_1 \end{array}$$

Trimming closures prevents the types of functions from "blowing up". Without closure trimming the closure term of a top level function like *main* would include all the types of all functions used in the program. In practice, most closure terms can be erased totally. For example, the definition of our *addTwo* function references the free variable *succ*. As *succ* contains no material closure components, neither does *addTwo*.

$$\begin{array}{l} \mathit{addTwo} :: \forall r_1\ r_2.\ \mathit{Int}\ \ r_1 \rightarrow \mathit{Int}\ \ r_2 \\ \mathit{addTwo}\ \ x \quad = \mathit{succ}\ (\mathit{succ}\ \ x) \end{array}$$

In our current implementation we only trim out closure information concerning *immaterial* region variables. Section §5.2.4 presents some ideas for also trimming out information concerning region variables that are constrained to be constant.

2.6 Type classing

In this section we discuss value type classes in Disciple. The general mechanism is similar to that used in Haskell, except that we need a special *Shape* constraint on types to be able to write useful class declarations.

As our current implementation does not implement dictionary passing, we limit ourselves to situations where the overloading can be resolved at compile time. For this reason, none of our class declarations have superclasses, and we do not support value type classes being present in the constraint list of a type. This in turn allows us to avoid considering most of the subtle issues discussed in [PJJM97]. We have made this restriction because we are primarily interested in using the type class mechanism to manage our region, effect and closure information. Exploring the possibilities for interaction between the various kinds of constraints represents an interesting opportunity for future work. There is also the possibility of defining multi-parameter type classes that constrain types of varying kinds.

2.6.1 Copy and counting

The need for a *Shape* constraint arises naturally when we consider functions that copy data. For example, the *copyInt* function which copies an integer value has type:

$$\begin{array}{lll} \mathit{copyInt} & :: & \forall r_1\ r_2.\ \mathit{Int}\ \ r_1 \xrightarrow{e_1} \mathit{Int}\ \ r_2 \\ & \rhd & e_1 = \mathit{Read}\ \ r_1 \end{array}$$

We will assume that this function is defined as a primitive. As r_2 is quantified we know that *copyInt* allocates the object being returned, which is what we expect from a copy function.

In Disciple programs, *copyInt* can be used to initialise mutable counters. For example:

```
startValue :: Int  r1 ▷ Const  r1
startValue = 5

fun ()
 = do   count   = copyInt startValue
        ...
        count   := count -1
        ...
```

startValue is defined at top level. In Disciple, if a top level value is not explicitly constrained to be *Mutable* then *Const* constraints are added automatically. We have included this one manually for the sake of example.

In the definition *fun* we have a counter that is destructively decremented as the function evaluates. As the type of (:=) (sugar for *updateInt*) requires its argument to be mutable, we cannot simply initialise the counter with the binding *count* = *startValue*. This would make the variable *count* an alias for the object bound to *startValue*. This in turn would require both *count* and *startValue* to have the same type, creating a conflict between the mutability constraint on *count* and the constancy constraint on *startValue*. We instead use *copyInt* to make a fresh copy of *startValue*, and this use object to initialise *count*.

2.6.2 Type classes for copy and update

After integers, another common data type in functional programs is the list. In Disciple we can declare the list type as:

$$
\begin{array}{ll}
\textbf{data} & \mathit{List}\ r_1\ a \\
= & \mathit{Nil} \\
\mid & \mathit{Cons}\ a\ (\mathit{List}\ r_1\ a)
\end{array}
$$

This declaration introduces the data constructors *Nil* and *Cons* which have the following types:

$$
\begin{array}{ll}
\mathit{Nil} & :: \forall r_1\ a.\ \mathit{List}\ r_1\ a \\
\mathit{Cons} & :: \forall r_1\ a.\ a \rightarrow \mathit{List}\ r_1\ a \xrightarrow{c_1} \mathit{List}\ r_1\ a \\
 & \quad \rhd\ c_1 = x : a
\end{array}
$$

Note that in the type of *Nil*, the region variable r_1 is quantified. This indicates that *Nil* behaves as though it allocates a fresh object at each occurrence.[9] On the other hand, in the type of *Cons* the region variable r_1 is shared between the second argument and the return type. This indicates that the returned object will contain a reference to this argument.

Using our list constructors, and the *copyInt* function from the previous section, we define *copyListInt* which copies a list of integers:

$$
\begin{array}{lll}
\mathit{copyListInt} & :: & \forall r_1\ r_2\ r_3\ r_4 \\
 & . & \mathit{List}\ r_1\ (\mathit{Int}\ r_2) \xrightarrow{e_1} \mathit{List}\ r_3\ (\mathit{Int}\ r_4) \\
 & \rhd & e_1 = \mathit{Read}\ r_1 \vee \mathit{Read}\ r_2
\end{array}
$$

$$
\begin{array}{l}
\mathit{copyListInt}\ xx \\
\quad = \textbf{case}\ xx\ \textbf{of} \\
\qquad \begin{array}{ll}
\mathit{Nil} & \rightarrow \mathit{Nil} \\
\mathit{Cons}\, x\ xs & \rightarrow \mathit{Cons}\ (\mathit{copyInt}\ x)\ (\mathit{copyListInt}\ xs)
\end{array}
\end{array}
$$

Once again, the fact that both r_3 and r_4 are quantified indicates that the returned object is freshly allocated. Note that e_1 includes an effect $\mathit{Read}\ r_1$ due to inspecting the spine of the list, as well as $\mathit{Read}\ r_2$ from copying its elements.

As *copyInt* and *copyListInt* perform similar operations, we would like define a type class that abstracts them. If we ignore effect information for the moment, we could try something like:

$$
\begin{array}{ll}
\textbf{class} & \mathit{Copy}\ a\ \textbf{where} \\
\quad \mathit{copy} & :: \quad a \rightarrow a
\end{array}
$$

Unfortunately, this signature for *copy* does not respect the fact that the returned object should be fresh. Our *copyInt* function produces a freshly allocated object, but *Int* instance of the type in the class declaration would be:

$$
\mathit{copy}_{\mathit{Int}} \quad :: \quad \forall r_1.\ \mathit{Int}\ r_1 \rightarrow \mathit{Int}\ r_1
$$

This would prevent us from using our overloaded *copy* function to make local, mutable copies of constant integers as per the previous section. As the argument

[9] However, if the returned object is constrained to be constant then the compiler can reuse the same one each time and avoid the actual allocation.

and return types include the same region variable, any constraints placed on one must be compatible with the other. On the other hand, the following class declaration is too weak:

$$\begin{array}{l}\textbf{class}\ \mathit{Copy}\ a\ \textbf{where}\\ \quad \mathit{copy}\ \ ::\ \ \forall b.\ a \to b\end{array}$$

If the argument of *copy* is an integer, then we expect the return value to also be an integer. What we need is for the argument and return types of *copy* to have the same overall *shape*, while allowing their contained region variables to vary.

We enforce this with the *Shape* constraint:

$$\begin{array}{lll}\textbf{class}\ \mathit{Copy}\ a\ \textbf{where} & & \\ \quad \mathit{copy} & :: & \forall b.\ a \to b \\ & \rhd & \mathit{Shape}\ a\ b\end{array}$$

Shape a b can be viewed as functional dependency [Jon00] between the two types a and b. The functional dependency is bi-directional, so if a is an *Int* then b must also be an *Int*, and if b is an *Int* then so must a. As we do not provide any mechanism for defining *Shape* from a more primitive structure, it is baked into the language.

This handles the argument and return types, though we still need to account for the effect of reading the argument. We do this with the *ReadT* (read type) effect:

$$\begin{array}{lll}\textbf{class}\ \mathit{Copy}\ a\ \textbf{where} & & \\ \quad \mathit{copy} & :: & \forall b.\ a \xrightarrow{e_1} b \\ & \rhd & e_1 = \mathit{ReadT}\ a \\ & , & \mathit{Shape}\ a\ b\end{array}$$

In the class declaration, *ReadT* a says that instances of the *copy* function are permitted to read any region variable present in the type a. Once this declaration is in place, we can add the instances for each of our copy functions:

$$\begin{array}{l}\textbf{instance}\ \mathit{Copy}\ (\mathit{Int}\ r_1)\ \textbf{where}\\ \quad \mathit{copy}\ \ = \mathit{copyInt}\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Copy}\ (\mathit{List}\ r_1\ (\mathit{Int}\ r_2))\ \textbf{where}\\ \quad \mathit{copy}\ \ = \mathit{copyListInt}\end{array}$$

Along with *ReadT*, there is a related *WriteT* that allows a function to have a write effect on any region variable in a type. Similarly, *MutableT* and *ConstT* place constraints on all the region variables in a type.

Next, we will use *WriteT* and *MutableT* to define the type class of objects that can be destructively updated:

$$\begin{array}{lll}\textbf{class}\ \mathit{Update}\ a\ \textbf{where} & & \\ \quad (:=) & :: & \forall b.\ a \to b \xrightarrow{c_1\ e_1} () \\ & \rhd & e_1 = \mathit{WriteT}\ a\ \vee\ \mathit{ReadT}\ b \\ & , & c_1 = x : a \\ & , & \mathit{Shape}\ a\ b \\ & , & \mathit{MutableT}\ a\end{array}$$

This declaration says that instances of (:=) may write to the first argument, read the second argument, hold a reference to the first argument during partial

application, require both arguments to have the same overall shape, and require regions in the first argument to be mutable.

Note that the types in class declarations are *upper bounds* of the possible types of the instances. Instances of (:=) must have a type which is at least as polymorphic as the one in the class declaration, and may not have an effect that is not implied by $WriteT\ a \vee ReadT\ b$. Nor may they place constraints on their arguments other than $Shape\ a\ b$ and $MutableT\ a$. Importantly, after a partial application of just their first arguments, they may not hold references to any material values other than these arguments. This last point is determined by the closure term $x : a$.

2.6.3 Shape and partial application

We now discuss how the *Shape* constraint works during partial application. We will use the overloaded equality function as an example. Here is the *Eq* class declaration:

$$
\begin{array}{lll}
\textbf{class}\ Eq\ a\ \textbf{where} & & \\
\quad (==) & :: & \forall b\ r_1.\ a \to b \xrightarrow{e_1\ c_1} Bool\ r_1 \\
& \rhd & e_1 = ReadT\ a \vee ReadT\ b \\
& , & c_1 = x : a \\
& , & Shape\ a\ b
\end{array}
$$

This declaration says that instances of (==) accept two arguments, and return a fresh boolean. Instances are permitted to read their arguments and hold a reference to the first one when partially applied. The arguments may also be required to have the same shape.

Consider the following binding:

$$isEmpty \quad = \quad (==)\ [\,]$$

This binding partially applies (==), resulting in a function that tests whether a list is empty. To determine the type of *isEmpty* we first instantiate the type of (==):

$$
\begin{array}{lll}
(==) & :: & a' \to b' \xrightarrow{e_1\ c_1} Bool\ r_1' \\
& \rhd & e_1 = ReadT\ a' \vee ReadT\ b' \\
& , & c_1 = x : a' \\
& , & Shape\ a'\ b'
\end{array}
$$

Taking [ ] to have the type $List\ r_2\ c$, we bind it to a' and eliminate the outer function constructor:

$$
\begin{array}{lll}
((==)\ [\,]) & :: & b' \xrightarrow{e_1\ c_1} Bool\ r_1' \\
& \rhd & e_1 = ReadT\ (List\ r_2\ c) \vee ReadT\ b' \\
& , & c_1 = x : List\ r_2\ c \\
& , & Shape\ (List\ r_2\ c)\ b'
\end{array}
$$

The $Shape\ (List\ r_2\ c)\ b'$ constraint requires b' to have the same shape as $List\ r_2\ c$. We satisfy this by giving b the type $List\ r_3\ d$, where r_3 and d are fresh:

$$\begin{array}{lll} ((==)\ [\,]) & :: & \mathit{List}\ r_3\ d \xrightarrow{e_1\ c_1} \mathit{Bool}\ r_1' \\ & \rhd & e_1 = \mathit{ReadT}\ (\mathit{List}\ r_2\ c)\ \vee\ \mathit{ReadT}\ (\mathit{List}\ r_3\ d) \\ & , & c_1 = x : \mathit{List}\ r_2\ c \\ & , & \mathit{Shape}\ (\mathit{List}\ r_2\ c)\ (\mathit{List}\ r_3\ d) \end{array}$$

The effect *ReadT* expresses a read on all region variables in its argument type. As we now know what this argument type is we can reduce the *ReadT* effect to a simpler form. Here, $\mathit{ReadT}\ (\mathit{List}\ r_2\ c)$ can be reduced to $\mathit{Read}\ r_2 \vee \mathit{ReadT}\ c$ and $\mathit{ReadT}\ (\mathit{List}\ r_3\ d)$ can be reduced to $\mathit{Read}\ r_3 \vee \mathit{ReadT}\ d$. As both arguments to our *Shape* constraint are list types, this constraint is partially satisfied, though we still need to ensure that c has the same shape as d:

$$\begin{array}{lll} ((==)\ [\,]) & :: & \mathit{List}\ r_3\ d \xrightarrow{e_1\ c_1} \mathit{Bool}\ r_1' \\ & \rhd & e_1 = \mathit{Read}\ r_2 \vee \mathit{ReadT}\ c \vee \mathit{Read}\ r_3 \vee \mathit{ReadT}\ d \\ & , & c_1 = x : \mathit{List}\ r_2\ c \\ & , & \mathit{Shape}\ c\ d \end{array}$$

This type can be reduced no further, so we will generalise it to create the scheme for *isEmpty*:

$$\begin{array}{lll} \mathit{isEmpty} & :: & \forall c\ d\ r_1\ r_3 \\ & . & \mathit{List}\ r_3\ d \xrightarrow{e_1\ c_1} \mathit{Bool}\ r_1 \\ & \rhd & e_1 = \mathit{Read}\ r_2 \vee \mathit{ReadT}\ c \vee \mathit{Read}\ r_3 \vee \mathit{ReadT}\ d \\ & , & c_1 = x : \mathit{List}\ r_2\ c \\ & , & \mathit{Shape}\ c\ d \end{array}$$

Note that as per §2.5.3 we have not generalised r_2 because it appears in the outermost closure of the function. At runtime, the application of (==) to [ ] will build a thunk containing a pointer to the function and the empty list. This empty list is shared between all uses of *isEmpty*.

2.6.4 Shape constraints and rigid type variables

Consider the following Haskell type class declaration:

$$\begin{array}{l} \textbf{class}\ \mathit{Foo}\ a\ \textbf{where} \\ \quad \mathit{foo} \quad :: \quad \forall b.\ a \to [b] \to [b] \end{array}$$

An instance of this class is:

$$\begin{array}{l} \textbf{instance}\ \mathit{Foo}\ \mathit{Bool}\ \textbf{where} \\ \quad \mathit{foo}\ x\ y \quad = \textbf{if}\ x\ \textbf{then}\ \mathit{tail}\ y\ \textbf{else}\ \mathit{reverse}\ y \end{array}$$

The locally quantified type variable b is called a *rigid type variable*. This highlights the fact that every instance of *foo* must have a similarly general type. For example, the following instance is invalid:

$$\begin{array}{l} \textbf{instance}\ \mathit{Foo}\ \mathit{Char}\ \textbf{where} \\ \quad \mathit{foo}\ x\ y = \textbf{if}\ x == \text{`}a\text{`}\ \textbf{then}\ \mathit{tail}\ y\ \textbf{else}\ [x] \end{array}$$

This non-instance tries to assign *foo* the following type:

$$\mathit{foo}_{\mathit{Char}} :: \mathit{Char} \to [\mathit{Char}] \to [\mathit{Char}]$$

This is strictly less general than the one in the type class declaration, because we cannot apply it to lists whose elements do not have type *Char*.

The *Copy* type class declaration also contains a rigid type variable. Here it is again:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Copy}\ \ a\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{copy} & :: & \forall b.\ a \xrightarrow{e_1} b \\
& \rhd & e_1 = \mathit{ReadT}\ \ a \\
& , & \mathit{Shape}\ \ a\ \ b
\end{array}
\end{array}
$$

Note the local $\forall b$ quantifier. We have said that *copyInt* is a valid instance of *copy* because it produces a freshly allocated object. Recall that *copyInt* has the following type:

$$
\begin{array}{lll}
\mathit{copyInt} & :: & \forall r_1\ r_2.\ \ \mathit{Int}\ \ r_1 \xrightarrow{e_1} \mathit{Int}\ \ r_2 \\
& \rhd & e_1 = \mathit{Read}\ \ r_1
\end{array}
$$

On the other hand, the following instance is *not* valid:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Copy}\ \ \mathit{Char}\ \ \textbf{where} \\
\quad \mathit{copy}\ \ x\ \ = x
\end{array}
$$

This is so because it does not actually copy its argument. We can see this fact in its type:

$$\mathit{copy}_{\mathit{Char}} :: \forall r_1.\ \ \mathit{Char}\ \ r_1 \to \mathit{Char}\ \ r_1$$

This situation is very similar to the one with $\mathit{foo}_{\mathit{Char}}$, because the signature of $\mathit{copy}_{\mathit{Char}}$ is not sufficiently polymorphic to be used as an instance for *copy*.

We now discuss how to determine the required type of an instance function from the type class declaration. The subtle point is in dealing with *Shape* constraints on rigid type variables.

Here is the *Copy* class declaration again. For the sake of example we have added the outer quantifier for a.

$$
\begin{array}{l}
\forall a.\ \textbf{class}\ \mathit{Copy}\ \ a\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{copy} & :: & \forall b.\ a \xrightarrow{e_1} b \\
& \rhd & e_1 = \mathit{ReadT}\ \ a \\
& , & \mathit{Shape}\ \ a\ \ b
\end{array}
\end{array}
$$

Say that we wish to determine the required type of $\mathit{copy}_{\mathit{Int}}$. To do this we instantiate the type class declaration with $\mathit{Int}\ \ r_1$, where r_1 is fresh. We can then re-generalise the declaration for r_1, to get a $\forall r_1$ quantifier at top level:

$$
\begin{array}{l}
\forall r_1.\ \textbf{class}\ \mathit{Copy}\ (\mathit{Int}\ \ r_1)\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{copy} & :: & \forall b.\ \ \mathit{Int}\ \ r_1 \xrightarrow{e_1} b \\
& \rhd & e_1 = \mathit{ReadT}\ (\mathit{Int}\ \ r_1) \\
& , & \mathit{Shape}\ (\mathit{Int}\ \ r_1)\ \ b
\end{array}
\end{array}
$$

Reducing the *ReadT* effect and the *Shape* constraint gives:

$$
\begin{array}{l}
\forall r_1.\ \textbf{class}\ \mathit{Copy}\ (\mathit{Int}\ \ r_1)\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{copy} & :: & \forall r_2.\ \ \mathit{Int}\ \ r_1 \xrightarrow{e_1} b \\
& \rhd & e_1 \ \ = \mathit{Read}\ \ r_1 \\
& , & b \ \ = \mathit{Int}\ \ r_2
\end{array}
\end{array}
$$

Reduction of the shape constraint has introduced the new type constraint $b = \mathit{Int}\ \ r_2$ where r_2 is fresh. This makes b have the same shape as the function's

first argument. We have also replaced $\forall b$ with $\forall r_2$. Every time the reduction of a *Shape* constraint on a quantified type variable introduces a new region variable, we quantify the new variable instead of the old one. Substituting for b completes the process:

$$\begin{array}{l} \forall r_1.\ \textbf{class}\ \mathit{Copy}\ (\mathit{Int}\ \ r_1)\ \textbf{where} \\ \quad \mathit{copy} \quad :: \quad \forall r_2.\ \mathit{Int}\ \ r_1 \xrightarrow{e_1} \mathit{Int}\ \ r_2 \\ \qquad\quad \rhd \quad e_1 = \mathit{Read}\ \ r_1 \end{array}$$

We can now extract the required type for $\mathit{copy}_{\mathit{Int}}$ by appending the outer quantifier, and the top-level $\mathit{Copy}\ (\mathit{Int}\ \ r_1)$ constraint:

$$\begin{array}{lll} \mathit{copy}_{\mathit{Int}} & :: & \forall\ r_1\ r_2.\ \mathit{Int}\ \ r_1 \xrightarrow{e_1} \mathit{Int}\ \ r_2 \\ & \rhd & e_1 = \mathit{Read}\ \ r_1 \\ & , & \mathit{Copy}\ (\mathit{Int}\ \ r_1) \end{array}$$

If we are performing this process to check whether a given instance function is valid, then we have already satisfied the $\mathit{Copy}\ (\mathit{Int}\ \ r_1)$ constraint. Discharging it gives:

$$\begin{array}{lll} \mathit{copy}_{\mathit{Int}} & :: & \forall\ r_1\ r_2.\ \mathit{Int}\ \ r_1 \xrightarrow{e_1} \mathit{Int}\ \ r_2 \\ & \rhd & e_1 = \mathit{Read}\ \ r_1 \end{array}$$

This is the expected type for an *Int* instance of *copy*. If the type of a provided instance function cannot be instantiated to this type, then it is invalid.

2.6.5 Shape constraints and immaterial regions

Consider the *IntFun* type from §2.5.4:

$$\begin{array}{lll} \textbf{data}\ \mathit{IntFun}\ \ r_{1..4}\ e_1\ c_1 & & \\ \quad = & \mathit{SInt} & (\mathit{Int}\ \ r_2) \\ \quad \mid & \mathit{SFun} & (\mathit{Int}\ \ r_3 \xrightarrow{e_1\ c_1} \mathit{Int}\ \ r_4) \end{array}$$

Using the class instantiation process from the previous section, the type of a *copy* instance function for *IntFun* must be at least as polymorphic, and no more effectful, closureful[10] or otherwise constrained than:

$$\begin{array}{l} \mathit{copy}_{\mathit{IntFun}} \\ \quad :: \quad \forall r_{1..8}\ e_1\ c_1 \\ \quad . \quad \mathit{IntFun}\ \ r_{1..4}\ e_1\ c_1 \xrightarrow{e_2} \mathit{IntFun}\ \ r_{5..8}\ e_1\ c_1 \\ \quad \rhd \quad e_2 = \mathit{Read}\ \ r_1 \vee \mathit{Read}\ r_2 \vee \mathit{Read}\ \ r_3 \vee \mathit{Read}\ \ r_4 \end{array}$$

Unfortunately, we don't have any way of writing a copy function for *IntFun* that has this type. We could try something like:

$$\begin{array}{l} \mathit{copy}_{\mathit{IntFun}}\ \ xx \\ \quad = \textbf{case}\ xx\ \textbf{of} \\ \qquad \begin{array}{ll} \mathit{SInt}\ \ i & \rightarrow \mathit{SInt}\ (\mathit{copyInt}\ \ i) \\ \mathit{SFun}\ \ f & \rightarrow \mathit{SFun}\ \ f \end{array} \end{array}$$

For the *SInt* alternative we have just used *copyInt* to copy the contained integer. However, we have no way of copying a function value, nor are we sure what it would mean to do so. Instead, we have simply reused the variable f on

[10] The author bags new word credit for "closureful".

the right of the second alternative. Unfortunately, this gives $copy_{IntFun}$ the following type:

$$
\begin{array}{ll}
copy_{IntFun} & \\
\quad :: & \forall r_{1..6}\ e_1\ c_1 \\
\quad . & IntFun\ \ r_{1..4}\ e_1\ c_1 \xrightarrow{e_2} IntFun\ \ r_{5..6}\ r_{3..4}\ e_1\ c_1 \\
\quad \rhd & e_2 = Read\ \ r_1 \vee Read\ \ r_2
\end{array}
$$

Note that in the return type of this function, r_5 and r_6 are fresh but r_3 and r_4 are not. The first two parameters of *IntFun* are material region variables that correspond to actual objects in the store. We could reasonably expect an instance function to copy these. On the other hand, the second two parameters are immaterial. For the *SFun* alternative, the best we can do is to pass f through to the return value, but doing this does not freshen the region variables in its type.

Our solution is to modify the reduction rule for *Shape* so that all value type and region variables that are not strongly material are identified. That is, if a particular variable in a data type definition does *not* always correspond to actual data in the store, then we will not freshen that variable when reducing *Shape*.

We also define the rule for reducing *ReadT* so that read effects on immaterial region variables are discarded. Immaterial regions do not correspond with real data in the store, so reading them does nothing.

Using these new rules, and the instantiation process from the previous section, the required type for $copy_{IntFun}$ becomes:

$$
\begin{array}{ll}
copy_{IntFun} & \\
\quad :: & \forall r_{1..6}\ e_1\ c_1 \\
\quad . & IntFun\ \ r_{1..4}\ e_1\ c_1 \xrightarrow{e_2} IntFun\ \ r_{5..6}\ r_{3..4}\ e_1\ c_1 \\
\quad \rhd & e_2 = Read\ \ r_1 \vee Read\, r_2
\end{array}
$$

This is the same type as our instance function, so we can accept it as valid.

2.7 Type directed projections

In §1.3.2 we discussed how references (and pointers) are used to update values within container structures, without knowledge of the surrounding container. We also discussed how they are used to update values that are shared by several different parts of the program, without needing information about how they are shared. On the other hand, in §1.7 we saw how the use of ML style references can lead to a large amount of refactoring effort when writing programs. This is because the reference appears in the value types of terms that use them, and we must use an explicit function call to read a reference when we want the contained value.

The Disciple projection system provides a mechanism to create references on the fly, so we can use them for shared update without the need to change the structure of value types. The fact that we can provide this mechanism while still tracking enough information to perform compile type optimisations is the primary reason we have developed the type system discussed in this chapter. We also provide a separate name space associated with each type constructor, and projection functions are placed in the name space corresponding to the type of value they project. This avoids the problem with Haskell style records, also discussed in §1.3.2, where the names of projection functions pollute the top-level scope of the program. In this thesis we restrict ourselves to associating namespaces with *constructors* instead of general types. This is to avoid issues with overlapping types such as $List\ a$ and $List\ Int$.

Projections are complementary to type classes. For example, when performing type inference for an expression like $show\ x$, the variable x may have a polymorphic type. As the instance function to use for *show* may be resolved at run time via a dictionary passing mechanism[11], the compiler itself will not know which instance function will be used. Due to this, the type of *show* in the class definition must be an upper bound of the types of all possible instances.

On the other hand, when performing type inference for the projection $x \odot \mathit{field1}$, we require the type of x to resolve to something that includes an outer constructor. We use this constructor to determine how to implement the projection of *field1*. This in turn allows each of the projections named *field1* to return values of *different* types.

[11] At least it can in a mature compiler like GHC. Our prototype implementation does not yet support dictionary passing, though we are not aware of any barrier to adding it.

2.7.1 Default projections

Consider the following data type definition.

$$\textbf{data}\ \ \mathit{Vec2}\ \ r\ a = \mathit{Vec2}\ \{\ x :: a;\ y :: a\ \}$$

x and y are the field names of the constructor. In Haskell, this definition would introduce x and y as record selectors in the top level scope. In Disciple, we instead get two projections $\odot\ x$ and $\odot\ y$ that can be applied to values of type $\mathit{Vec2}\ \ r\ a$, for any r or a. As our type expressions may contain commas, we use a semicolon as a field separator instead of a comma. Also, $\odot$ is an infix operator, and $\odot\ x$ is written `.x` in the concrete syntax. Here is an expression which uses the two projections:

$$\begin{array}{lll}\textbf{do} & \mathit{vec} & = \mathit{Vec2}\ \ 2.0\ 3.0\\ & \mathit{angle} & = \mathit{sqrt}\ (\mathit{square}\ \ \mathit{vec} \odot x + \mathit{square}\ \ \mathit{vec} \odot y)\end{array}$$

The projection operator $\odot$ binds more tightly than function application, so $\mathit{square}\ \ \mathit{vec} \odot x$ should be read as $\mathit{square}\ (\mathit{vec} \odot x)$. If we do not have a handy object of the required type then we can refer to the projection functions in a particular namespace directly with the & operator. For example, we could rewrite the above expression as:

$$\begin{array}{lll}\textbf{do} & \mathit{vec} & = \mathit{Vec2}\ \ 2.0\ 3.0\\ & \mathit{angle} & = \mathit{sqrt}\ (\mathit{square}\ (\mathit{Vec2}\ \&x\ \ \mathit{vec})\\ & & \qquad + (\mathit{square}\ (\mathit{Vec2}\ \&y\ \ \mathit{vec}))\end{array}$$

The projections associated with field names are called *default projections*. These are introduced automatically by the language definition. For *Vec2* the two projection functions are:

$$\begin{array}{lll}\mathit{Vec2}\ \ \&x & :: & \forall r_1\ a.\ \ \mathit{Vec2}\ \ r_1\ a \xrightarrow{e_1} a\\ & \rhd & e_1 = \mathit{Read}\ \ r_1\\ \mathit{Vec2}\ \ \&x\ \ (\mathit{Vec2}\ \ x\ y) & & = x\end{array}$$

$$\begin{array}{lll}\mathit{Vec2}\ \ \&y & :: & \forall r_1\ a.\ \ \mathit{Vec2}\ \ r_1\ a \xrightarrow{e_1} a\\ & \rhd & e_1 = \mathit{Read}\ \ r_1\\ \mathit{Vec2}\ \ \&y\ \ (\mathit{Vec2}\ \ x\ y) & & = y\end{array}$$

This syntax is similar to the use of :: in C++ to define class methods. For example, the name of a method in a class named *Vec2* would be $\mathit{Vec2} :: x$.

2.7.2 Ambiguous projections and type signatures

Ambiguous projections arise when we project a value whose type is not constrained to include an outer constructor. For example, the projections in the following code are ambiguous:

$$\mathit{tupleOfVec} = \lambda\, \mathit{vec}\,.\ (\mathit{vec}\ \odot\ x,\ \ \mathit{vec}\ \odot\ y)$$

Without further information, the type of *vec* in this code is just a variable. If our program included more than one data type that had an x or y field, then there would be no way of knowing which projection function to use.

The programmer can resolve this problem by providing a type signature that constrains the type of *vec*. For example:

$$\begin{aligned}&\mathit{tupleOfVec} :: \mathit{Vec2}\ a \rightarrow (a,\ a)\\ &\mathit{tupleOfVec} = \lambda\, \mathit{vec}\,.\ (\mathit{vec}\ \odot\ x,\ \ \mathit{vec}\ \odot\ y)\end{aligned}$$

Note that we do not need to provide region, effect or closure information in type signatures. The fact that this information is missing from the above signature can be determined from the kind of *Vec2*, and it can be filled in by the type inference process.

2.7.3 Pull back projections

Pull back projections allow the programmer to create references to the fields of a record. For example, a reference to the x field of our *Vec2* type can be created with $\mathit{vec} \odot_{\#} x$, pronounced "*vec* pull x". If the type of *vec* is $\mathit{Vec2}\ \ r_1\ a$ then the type of $\mathit{vec} \odot_{\#} x$ is $\mathit{Ref}\ \ r_1\ a$. If we imagine *Ref* types being equivalent to pointers in C, then $\mathit{vec} \odot_{\#} x$ has the same meaning as the C expression `&(vec.x)`. The $:=_{\#}$ function (pronounced "update ref") is then used to update the value of the field. Note that $\mathit{vec} \odot_{\#} x$ and $:=_{\#}$ are written as `vec#x` and `#=` in the concrete syntax.

Here is the type of $:=_{\#}$

$$\begin{aligned}(:=_{\#}) \quad :: \quad & \forall r_1\ a.\ \ \mathit{Ref}\ \ r_1\ a \longrightarrow a \xrightarrow{e_1\ c_1} ()\\ \triangleright \quad & e_1 = \mathit{Write}\ \ r_1\\ , \quad & c_1 = x : \mathit{Ref}\ \ r_1\\ , \quad & \mathit{Mutable}\ \ r_1\end{aligned}$$

Here is an example that creates a vector then updates one of its components:

$$\begin{aligned}\textbf{do}\quad & \mathit{vec} \quad = \mathit{Vec2}\ \ 2.0\ 3.0\\ & \mathit{ref} \quad = \mathit{vec} \odot_{\#} x\\ & \ldots\\ & \mathit{ref} \quad :=_{\#}\ \ 5.0\\ & \ldots\end{aligned}$$

After the update statement has been executed, the projection $\mathit{vec}\ \odot\ x$ will return the value 5.0 instead of 2.0. Pull back projection functions can also be accessed directly. Here are the names and types of the pull back projections for the x and y fields:

$$\begin{aligned}&\mathit{Vec2}\ \ \&\ x_{\mathit{pull}} \quad :: \forall r_1\ a.\ \ \mathit{Vec2}\ \ r_1\ a \rightarrow \mathit{Ref}\ \ r_1\ a\\ &\mathit{Vec2}\ \ \&\ y_{\mathit{pull}} \quad :: \forall r_1\ a.\ \ \mathit{Vec2}\ \ r_1\ a \rightarrow \mathit{Ref}\ \ r_1\ a\end{aligned}$$

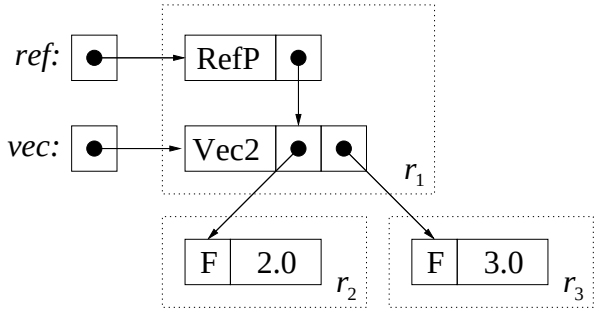
Note that the created reference shares the same region variable as the projected value. Also note that as *Vec2* only has a single data constructor, the functions x_{pull} and y_{pull} are pure. This is because when we evaluate an expression like $vec \odot_{\#} x$, we do not need to access the *vec* object at all. We simply allocate a new reference that contains a pointer into it. This can be done based on the address of the *vec* object, the object itself is not needed. For example, if we say:

$$\begin{aligned} vec &:: Vec2\ r_1\ (Float\ r_2)\ (Float\ r_3) \\ vec &= Vec2\ 2.0\ 3.0 \end{aligned}$$

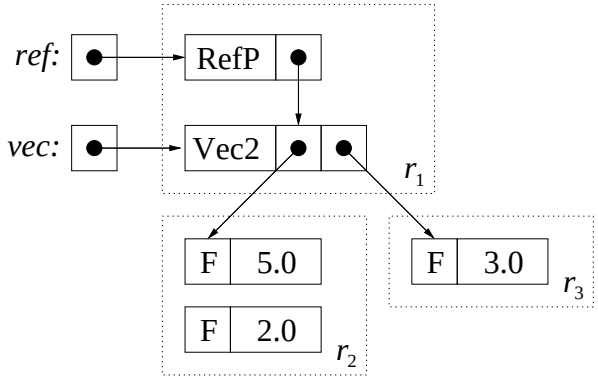
then we would have:

$$\begin{aligned} ref &:: Ref\ r_1\ (Float\ r_2) \\ ref &= vec \odot_{\#} x \end{aligned}$$

which produces the following objects in the store:



We use the tag *RefP* to record the fact that the *ref* object is a pull back reference that points into another object, as opposed to a regular ML style reference. When we execute the statement $ref :=_{\#} 5.0$, it is the pointer inside the *vec* object that is updated, not the *Float* object itself:



This leaves the old 2.0 object to be reclaimed by the garbage collector.

The benefit of this system over ML style references is that we are able to update data structures without needing *Ref* in their type definitions, which addresses the refactoring problem discussed in §1.7. Note that in the above diagram, both the *Vec2* and *RefP* objects are in the same region, r_1. This means that when we use a function like $(:=_{\#})$ to update the vector via the reference, the vector object will also be marked as mutable.

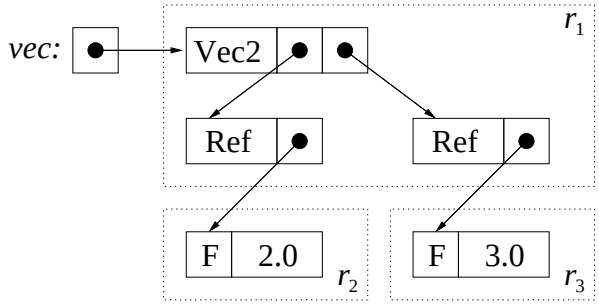
Although we don't *need* ML style references, Disciple does support them, and we can equally define:

$$\textbf{data}\ \ \mathit{Vec2}\ \ r_1\ a = \mathit{Vec2}\ \{\ x :: \mathit{Ref}\ \ r_1\ a;\ \ y :: \mathit{Ref}\ \ r_1\ a\ \}$$

In this case we would construct a vector with:

$$\begin{aligned} \mathit{vec}\ \ &:: \mathit{Vec2}\ \ r_1\ (\mathit{Float}\ \ r_2)\ (\mathit{Float}\ \ r_3) \\ \mathit{vec}\ \ &= \mathit{Vec2}\ (\mathit{Ref}\ \ 2.0)\ (\mathit{Ref}\ \ 3.0) \end{aligned}$$

This produces the following objects in the store:



Here, the reference objects include the constructor tag *Ref*, instead of *RefP* as before. This indicates that to update these references, the pointer in the object itself should be modified, not the word that is pointed to.

2.7.4 Custom projections

Along with the default field projections introduced by data type declarations, the programmer can also define their own custom projection functions. In fact, any variables they desire can be added to the name space associated with a type constructor, whether they are bound to functions that perform true projections, or not. For example, we can add a *magnitude* function to the *Vec2* name space with:

$$\begin{aligned} &\textbf{project}\ \ \mathit{Vec2}\ \ \textbf{where} \\ &\quad \mathit{magnitude}\ (\mathit{Vec2}\ \ x\ \ y) \\ &\qquad = \mathit{sqrt}\ (\mathit{square}\ \ x + \mathit{square}\ \ y) \end{aligned}$$

We use $\odot\ \mathit{magnitude}$ to invoke this new projection. For example:

$$\begin{aligned} &\textbf{do}\ \ \ \mathit{vec} \quad\ = \mathit{Vec2}\ \ 2.0\ 3.0 \\ &\qquad \mathit{putStr}\ \ (\text{“}\texttt{The magnitude is:}\text{”}\ ++\ (\mathit{show}\ \ \mathit{vec} \odot \mathit{magnitude})) \end{aligned}$$

Unlike default projections, custom projections can be defined to take extra arguments. For example, here is a projection to determine the dot product of two vectors:

$$\begin{aligned} &\textbf{project}\ \ \mathit{Vec2}\ \ \textbf{where} \\ &\quad \mathit{dot}\ (\mathit{Vec2}\ \ \mathit{x1}\ \ \mathit{y1})\ (\mathit{Vec2}\ \ \mathit{x2}\ \ \mathit{y2}) \\ &\qquad = \mathit{x1} * \mathit{x2} + \mathit{y1} * \mathit{y2} \end{aligned}$$

We can then use it as:

$$\begin{aligned} &\textbf{do}\ \ \ \mathit{vec} \quad\ = \mathit{Vec2}\ \ 2.0\ 3.0 \\ &\qquad \mathit{vec2}\ \ = \mathit{Vec2}\ \ 4.0\ 5.0 \\ &\qquad \mathit{putStr}\ \ (\text{“}\texttt{The product is:}\text{”}\ ++\ (\mathit{show}\ \ \mathit{vec}\ \odot \mathit{dot}\ \ \mathit{vec2})) \end{aligned}$$

This allows a style of programming similar to using local methods in object oriented languages. For example, in Java we would write `vec.dot(vec2)`. With Disciple code, we find it helpful to view the projection $\odot$ *dot* as a single operator. This highlights the similarities with the equivalent expression in vector calculus, $\overline{v_1} \bullet \overline{v_2}$.

Disciple also provides a punning syntax for adding variables to projection namespaces. This allows the programmer to add variables defined elsewhere in the module, and helps reduce the level of indenting in the code. For example, we could define our *magnitude* and *dot* projections with:

$$\textbf{project}\ \ \mathit{Vec2}\ \ \textbf{with}\ \{\mathit{magnitude}, \mathit{dot}\}$$

$$\begin{aligned}&\mathit{magnitude}\ (\mathit{Vec2}\ \ x\ \ y)\\ &\quad = \mathit{sqrt}\ (\mathit{square}\ \ x + \mathit{square}\ \ y)\end{aligned}$$

$$\begin{aligned}&\mathit{dot}\ (\mathit{Vec2}\ \ \mathit{x1}\ \ \mathit{y1})\ (\mathit{Vec2}\ \ \mathit{x2}\ \ \mathit{y2})\\ &\quad = \mathit{x1} * \mathit{x2} + \mathit{y1} * \mathit{y2}\end{aligned}$$

We find this syntax useful when writing library code. Our usual approach is to define all the "helper" functions for a particular data type in the same module that declares it. These helper functions are present in the top level scope of the module, but are not exported from it directly. We use the punning syntax to add the helper functions to the projection namespace for the data type. We then export the data type name, and the projection namespace along with it. This allows us to write the majority of our program in the familiar Haskell style, while reducing the opportunity for name clashes between modules.

2.8 Comparisons with other work

2.8.1 FX. 1986 – 1993. Gifford, Lucassen, Jouvelot and Talpin.

Although Reynolds [Rey78] and Popek *et al* [PHL+77] had discussed the benefits of knowing which parts of a program may interfere with others, Gifford and Lucassen [GL86] were the first to annotate a subroutine's type with a description of the effects it may perform. This allowed reasoning about effects in languages with first class functions, whereas previous work based on flow analysis [Ban79] was limited to first order languages. A refined version of their system is embodied in the language FX [GJSO91], which has a Scheme-like syntax. We consider FX to be a spiritual predecessor of Disciple.

In Gifford and Lucassen's original system [GL86], the types of subroutines are written $\tau \rightarrow_C \tau$ where C is an "effect class" and can be one of PROCEDURE, OBSERVER, FUNCTION or PURE. Subroutines marked PROCEDURE are permitted to read, write and allocate memory. OBSERVER allows a subroutine to read and allocate memory only. FUNCTION allows a subroutine to allocate memory only. A subroutine marked PURE may not read, write or allocate memory. Correctness dictates that subroutines marked PURE cannot call subroutines marked FUNCTION, those cannot call subroutines marked OBSERVER, and they cannot call subroutines marked PROCEDURE.

In this system, the concept of purity includes idempotence, and a subroutine that allocates its return value is not idempotent. Although such a subroutine cannot interfere with other parts of the program, the fact that it might allocate memory must be accounted for when transforming it. We will return to this point in §4.4.5. Note that in Disciple we use quantification of region variables to track whether a function allocates its return value, and our definition of purity includes functions that do so.

In [LG88] Gifford and Lucassen introduce the polymorphic effect system. This system includes region variables, quantification over region and effect variables, and effect masking. The primitive effects are *Read* r, *Write* r and *Alloc* r, and $e_1 \vee e_2$ is written `maxeff` e_1 e_2. Their language uses explicit System-F style type, region and effect abstraction and applications, which makes their example programs quite verbose. Their system also includes region unions, where the region type `union` r_1 r_2 represents the fact that a particular object may be in either region r_1 or region r_2. Disciple does not yet include region unions as they complicate type inference. This point is discussed in §5.2.2.

In [JG91] Jouvelot and Gifford describe an algebraic reconstruction algorithm for types and effects. They separate type schemes into two parts, the value type and a set of effect constraints, which gives us the familiar $\forall \overline{a}.\ \tau \rhd \Omega$ for type schemes. Here, $\overline{a}$ is a collection of type variables, τ is the body of the type and Ω are the constraints. On the other hand, the left of the constraints in their work can be full effect terms, not just variables. They present a proof of soundness, but only a single example expression. They also remark that they were still working on the implementation of their system in FX, so its practicality could be assessed.

In [TJ92a] Talpin and Jouvelot abandon the explicit polymorphism present in previous work, require the left of effect constraints to be a variable, and

introduce sub-effecting. This allows their new system to have principle types. Sub-effecting is also used to type if-expressions, as the types of both alternatives can be coerced into a single upper bound. Finally, in [TJ92b] they present the Type and Effect Discipline and address the problem of polymorphic update §2.4. They use effect information to determine when to generalise the type of a let-bound variable, instead of relying on the syntactic form of the expression as they did in [JG91]. We have based Disciple on this work.

2.8.2 C++ 1986 Bjarne Stroustrup.

The C++ language [Str86, Cpp08] includes some control over the mutability of data. In C++ a pointer type can be written `*const`, which indicates that the data it points to cannot be updated via that pointer. Pointers can also be explicitly defined as mutable. Fields in structures and classes can be defined as either mutable or constant, though they default to mutable due to the need to retain backwards compatibility with C. C++ also provides some limited control over side effects whereby a `const` qualifier can be attached to the prototype of a class method. This indicates that it does not (or at least should not) update the attributes of that class. However, this can be circumvented by an explicit type cast, or by accessing the attribute via a non-const pointer.

`const` annotations are also supported in C99 [C05]. Some C compilers including GCC [GCC09] provide specific, non-standard ways to annotate function types with mutability and effect information. For example in GCC the programmer can attach a purity attribute to a function that allows the optimiser to treat it as being referentially transparent. Attributes can also be added to variables to indicate whether or not they alias others. Of course, these attributes are compiler pragmas and not checked type information, and neither C++ or C99 has type inference. With DDC we can infer such information directly from the source program, and the type system for our core language ensures that it remains valid during program transformation.

More recent work based on Java [BE04] can ensure that `const` qualified objects remain constant, and [FFA99] presents a general system of type qualifiers that includes inference. However, neither of these systems include region or effect information, or discuss how to add qualifiers to Haskell style algebraic data types. In [FFA99] the authors mention that some effect systems can be expressed as type qualifier (annotation) systems, but state that the exact connection between effect systems and type qualifiers was unclear. In this chapter we have shown how to re-use Haskell's type classing system to qualify both region and effect information, which brings regions, effects and qualifiers into single framework.

2.8.3 Haskell and unsafePerformIO. 1990 Simon Peyton Jones *et al.*

The Haskell Foreign Function Interface (FFI) [Cha02] provides a function *unsafePerformIO* that is used to break the monadic encapsulation of IO actions. It has the following type:

$$unsafePerformIO :: IO\,a \rightarrow a$$

Use of this function discards the guarantees provided by a pure language, in favour of putting the programmer in direct control of the fate of the program. Using *unsafePerformIO* is akin to casting a type to `void*` in C. When a programmer is forced to use *unsafePerformIO* to achieve their goals, it is a sign that the underlying system cannot express the fact that the program is still safe. Of course, this assumes the programmer knows what they're doing and the resulting program actually *is* safe.

As Disciple includes an effect system which incorporates masking, the need for a function like *unsafePerformIO* is reduced. As discussed in §2.3.7, if a particular region is only used in the body of a function, and is not visible after it returns, then effects on that region can be masked. In this case the system has proved that resulting program *is* actually safe.

On the other hand functions like *unsafePerformIO* allow the programmer to mask top level effects, such as *FileSystem*. For example, we might know that a particular file will not be updated while the program runs, so the effect of loading the file can be safely masked. In these situations the type system must always "trust the programmer", as it cannot hope to reason about the full complexity of the outside world.

2.8.4 Behaviors and Trace Effects. 1993 Nielson and Nielson *et al*

In [NN93] Nielson and Nielson introduce *behaviours*, which are a richer version of the FX style effect types. As well as containing information about the actions a function may perform, behaviours include the order in which these actions take place. They also represent whether there is a non-deterministic choice between actions, and whether the behaviour is recursive. Having temporal information in types can be used to, say, enforce that files must be opened before they are written to. Skalka *et al*'s recent work [SSh08] gives a unification based inference algorithm for a similar system. For Disciple, we have been primarily concerned with optimisation and have so far avoided adding temporal information to our effect types. However, we expect that Disciple's main features such as mutability inference and purity constraints are reasonably independent of temporal information, and adding it represents an interesting opportunity for future work.

2.8.5 λ_{var}. 1993 – 1994 Odersky, Rabin, Hudak, Chen

In [ORH93] Odersky, Rabin and Hudak present an untyped monadic lambda calculus that includes assignable variables. Interestingly, their language includes a keyword **pure** that provides effect masking. **pure** is seen as the opposite of the monadic **return** function. This work is continued in Rabin's thesis [Rab96]. The Imperative Lambda Calculus [SRI91, YR97] is a related system.

In [CO94] Chen an Odersky present a type system for λ_{var} to verify that uses of **pure** are safe. This is done by stratifying the type system into two layers, that of pure expressions and that of commands. Their inference algorithm uses a simple effect system that does not distinguish between pure and impure lambda bound functions. They note that using the region variables of Talpin and Jouvelot's system [TJ92a] would give better results.

2.8.6 MLKit. 1994 Tofte, Talpin, Birkedal

MLKit [TBE+06], uses regions for storage management, whereas DDC uses them to help reason about the mutability and sharing properties of data. In MLKit, region annotations are only present in the core language. As in DDC, MLKit supports region polymorphism, so functions can be written that accept their arguments from any region, and output their result into any region. Unlike DDC, MLKit adds region annotations to function types, as the runtime objects that represent functions are also allocated into regions.

MLKit performs type inference with a two stage process [TB98]. The SML typing of the program is determined first, and region annotations are added in a separate analysis. This helps when performing type inference in the presence of polymorphic recursion, which is important for storage efficiency. Although polymorphic recursion of value types is known to make the general type inference problem undecidable [Myc84], in MLKit it is supported on the region information only, via a fixed point analysis. As DDC does not use regions for storage management, polymorphic recursion is not as important, and we do not support it.

2.8.7 Functional Encapsulation. 1995. Gupta

In [Gup95] Gupta presents a system to convert mutable objects to constant ones for the parallel language Id. As discussed in §2.3.7 this is needed for objects that are constructed imperatively, but are used functionally thereafter. Like our own system, Gupta's is based on Leroy's closure typing §2.5. He presents a term **close** t_1 whose result has the same value as t_1, except that the type system statically enforces that it will no longer be updated. The type of **close** t_1 can also be generalised, because the return value is guaranteed not to suffer the problem of polymorphic update §2.4. **close** is interesting because it serves as the dual of the *effect* masking operator, **pure**, which appears in λ_{var} [ORH93].

As in our own system, Gupta uses region variables to track the mutability of objects. Instead of using region constraints, region variables are only attached to the types of mutable objects. All constant objects are annotated with the null region ϵ.

In his conclusion, Gupta laments that **close** had not yet been implemented in the Id compiler, and it still relied on "hacks". The Id language was reincarnated as a part of pH [NAH+95], but **close** did not make it into the language specification. Being based on Haskell, it ended up using state monads to provide its impure features. Although we have not yet implemented mutability masking in DDC, it is a highly desirable feature and is first in line for future work §5.2.1.

2.8.8 Objective Caml. 1996 Leroy, Doligez, Garrigue, Rémy and Jérôuillon.

As well as *Ref* types, O'Caml [LDG+08] supports mutable record fields. In fact, the *Ref* constructor is expressed as a record with a single mutable field. Mutable fields are declared with the **mutable** keyword. Fields that are not declared as mutable default to constant. Mutable fields are updated with the $\leftarrow$ operator.

The following example is from the O'Caml 3.11 manual:

$$\textbf{type}\ \ \textit{mutable\_point} \quad = \{\textbf{mutable}\ x : \textit{float};\ \textbf{mutable}\ y : \textit{float}\};;$$
$$\textbf{let}\ \ \textit{translate}\ \ p\ dx\ dy \quad = \ p.x \leftarrow p.x + dx;\ p.y \leftarrow p.y + dy;;$$

One of the benefits of mutable record fields over *Ref* types is that we do not need to sprinkle calls to *readRef* throughout our code. This reduces the refactoring effort required when the mutability of an object is changed. However, unlike Disciple, O'Caml does not support mutability polymorphism, so two records that have the same overall structure but differ in the mutabilities of their fields have incompatible types. A constant list has a different type to a mutable list, and the standard O'Caml libraries only provide the constant version. This point was also discussed in §1.7.

2.8.9 Ownership Types. 1998 Clarke, Potter and Noble

Ownership typing provides a mechanism to prevent references to the internal representation objects from being inadvertently "leaked" to clients. For example, consider the following class:

```
class SavingsAccount {
     private:
     Integer balance;

     public:
     Integer getBalance()         { return balance; }
     void    accumulateInterest() { balance = balance * 1.1; }
}
```

In a language such as Java, boxed integers of type `Integer` are passed by reference. Although `balance` has been marked as private, client classes are able to change this field by destructively updating the value returned by `getBalance`. Note that we cannot simply add a `const` annotation to the `balance` field, as this would also prevent `accumulateInterest` from changing its value. Possible solutions to this problem include returning a physical copy of the `balance` value from `getBalance`. We could also mark `getBalance` as returning a `const Integer`, and perform a type cast in its return statement. However, both of these solutions rely on the programmer actually noticing the problem in the first place.

With the system described in [CPN98], the programmer can mark the `balance` field as belonging to the internal representation of `SavingsAccount` (and possibly its super classes). The type system then ensures that methods outside this class cannot gain a reference to this field. Ownership typing is related to region typing because both systems provide control over the possible aliasing of data. For example, for the `SavingsAccount` example we must also handle the case where a reference to `balance` is written into an array, then read out, then returned to some client method. Although both region and ownership typing systems share some ideas, the system in [CPN98] is based around a first order object oriented language, instead of a higher order functional language like ours. In DDC we use region typing to reason about aliasing, but do not have an object oriented class system, and make no attempt to enforce similar ownership properties.

2.8.10 Calculus of Capabilities and Cyclone. 1999 Crary, Walker, Grossman, Hicks, Jim, and Morrisett

Cyclone [JMG+02] is a type-safe dialect of C which uses regions for storage management. Its type system derives from Crary, Walker and Morrisett's work on the Calculus of Capabilities [CWM99]. The Vault [DF01] and RC [GA01] languages are related.

Cyclone's type safety is achieved in part by using region typing to track the lifetimes of objects, and to ensure that programs do not dereference dangling pointers [GMJ+02]. Cyclone has region polymorphism and parametric value polymorphism [Gro06], but not mutability polymorphism. Being an imperative style language, programs tend to be expressed using update and pointer manipulation. Allocation is explicit, though deallocation can be performed via the region system, or implicitly via garbage collection.

As in C, higher order functions can be introduced using function pointers. Cyclone supports existential types, and these can be used to express type safe function closures. Cyclone does not support full Hindley-Milner style type reconstruction, but instead relies on user provided type annotations. Region annotations are attached to pointer types in the source language, though many annotations can be elided and subsequently reconstructed by using intra-function type inference and defaulting rules.

The main technical feature that the Calculus of Capabilities (CC) has over the DDC core language is that the capability to perform an action can be revoked. The CC can then statically ensure that that a revoked capability is no longer used by the program. This mechanism is used in Cyclone's region system, where the capability to access a particular region is revoked when the region is deallocated. In contrast, in DDC a capability such as the ability to update a region cannot be revoked by the programmer. We have discussed mutability masking in §2.3.7, but have not implemented it. On the other hand, DDC supports full type inference (apart from ambiguous projections, which are an orthogonal issue).

Although Cyclone is an imperative language, its use of regions in the source language means that it shares some common ground with Disciple. For example, here is the type of sets from [GMJ+02]:

```
struct  Set<α, ρ> {
   list_t <α, ρ> elts;
   int (*cmp)(α, α; regions_of(α));
}
```

The ρ annotation is the primary region variable and α is a type variable. The term `regions_of(`α`)` is an effect that represents the fact that the comparison function `cmp` on two values of type α could access any region contained in that type. In this respect `regions_of(`α`)` has the same meaning as $ReadT\ a \vee WriteT\ a$ from §2.6. Note that as Cyclone is based on C, most data is mutable. In such a language there is less to be gained by separating effects on regions into reads and writes. A general region access effect suffices.

2.8.11 BitC. 2004 Shapiro, Sridhar, Smith, Doerrie

BitC [SSD08a] is a Scheme-like language targeted at systems programming. One of its stated aims is to offer *complete mutability*, meaning that any location – whether on the stack, heap or within unboxed structures – can be mutated [SSS08]. BitC supports the imperative variables that we decided not to in §2.1.

The operational semantics of BitC includes an explicit stack as well as a heap, and the arguments of functions are implicitly copied onto the stack during application. This allows the local copies to be updated in a way that is not visible to the caller, a behaviour demonstrated by the following C program:

```
int fun(int x)
{
      x = x + 1;
      return x;
}
```

BitC includes mutability inference, and inferred type for the BitC version of `fun` will be:

```
(mutable int) -> (mutable int)
```

Note that the fact that `x` is updated locally to `fun` has "leaked" into its type. We do not actually need to pass a mutable integer to `fun`, because only the fresh copy, located in the stack frame for the call, will be updated. For this reason, BitC introduces the notion of *copy compatibility*, which is similar to the property expressed by our *Shape* constraint from §2.6.2. We can pass a `const int` to a function expecting a `mutable int`, because the first will be implicitly copied during the call.

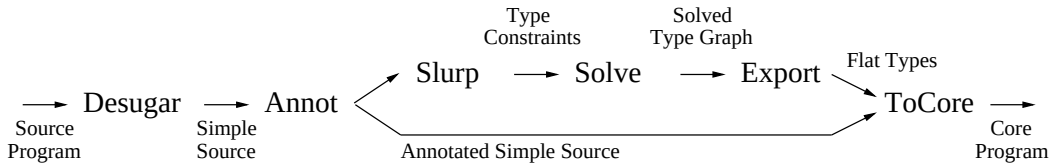
Although [SSD08b] discusses adding effect typing to BitC, it does not mention region variables, so the possible effects are limited to the coarse-grained *pure*, *impure* and *unfixed*. Exploiting effect information during program optimisation is not discussed, and the more recent formal specification of the type system in [SSS09] does not include it.

2.8.12 Monadic Regions. 2006 Fluet, Morrisett, Kiselyov, Shan

In [FM06] Fluet and Morrisett draw on the MLKit and Cyclone work to express a version of the region calculus in a monadic framework. Once again, they focus on using regions for storage management. They trade complexity of the original region type system for complexity of encoding, though the result could serve as a useful intermediate language.

Chapter 3

Type Inference



The above diagram gives an overview of our compilation process. The source program is first desugared into a simpler language, and then annotated with type variables that serve as "hooks" for type constraints. Type constraints relating these hook variables are extracted (slurped) from the annotated program. The constraints are solved, which produces a solution in the form of a type graph. The solution is a graph because it can contain cycles through effect and closure information due to the definitions of recursive functions. This was discussed in §2.3.8. We then extract flat, non-graphical types for each of the hook variables, and use this information to translate the annotated source program into the core language.

This chapter concerns the annotation, constraint slurping, solving and export stages. We give the motivation for our overall approach in §3.1, and discuss the use of type graphs. We define the simplified source language in §3.2, and go on to discuss the annotation process and constraint slurping. Constraint solving is outlined in §3.3 and §3.4, where §3.3 deals with the reduction of monomorphic constraints, and §3.4 discusses type generalisation and the extraction of flat types. In practice we also keep track of how type schemes are instantiated. This information is used when translating the source program to core, but we do not discuss this or other details of the translation in this thesis.

Section §3.5 extends the source language and inference algorithm with support for type directed projections, and §3.6 discusses how to handle mutual recursion in the presence of such projections. Section §3.7 discusses the built-in type class constraints such as *Pure* and *Shape*, and §3.8 considers how to produce reasonable error messages.

3.1 Binding order and constraint based inference

When performing type inference for Haskell, a binding dependency graph is used to determine which groups of bindings are mutually recursive. This graph is also used to sort the binding groups so that their types are generalised before they need to be instantiated [Jon99]. Unfortunately, due to the inclusion of type directed projections, a similar dependency graph cannot be extracted from Disciple programs before type inference proper.

Consider the following program:

$$\begin{aligned} &\mathit{fun1}\ \ x &&= 1 + \mathit{fun2}\ x\ 5 \\ &\mathit{fun2}\ \ x\ y &&= x + y \end{aligned}$$

As the body of *fun1* references *fun2*, we should generalise the type of *fun2* before inferring the type of *fun1*. It is easy to extract such dependencies from Haskell programs because at each level of scope, all bindings must have unique names. However, in Disciple programs, projections associated with different type constructors can share the same name. For example:

project *T1* **where**
  field1 $x = (x,\ 5)$

project *T2* **where**
  field1 $x = (x,$ "`hello`")

We have defined two projections named *field1*, one for constructor *T1* and one for constructor *T2*. Now consider what happens when we perform a *field1* projection in the program:

$$\mathit{fun}\ \ x = \ldots\ x \odot \mathit{field1}\ \ \ldots$$

This $x \odot \mathit{field1}$ projection will be implemented by one of the instance functions above, but we cannot determine which until we know the type of x. For Disciple programs, there is no easy way to determine the binding dependency graph, or to arrange the bindings into an appropriate order before inferring their types. Instead, we must determine how the bindings depend on each other on the fly, during inference.

This implies that our inference algorithm cannot be entirely syntax directed. When inferring the type of *fun*, once we determine which instance function to use for the $\odot\,\mathit{field1}$ projection, we may discover that this type hasn't been inferred yet either. We must then stop what we're doing and work out the type of the instance function, before returning to complete the type of *fun*. In general, this process is recursive. Work on the types of several bindings may need to be deferred so that we can first determine the type of another.

We manage this problem with a constraint based approach, similar to that used by Heeren in the Helium compiler [HHS02, HHS03, Hee05]. We extract type constraints from the desugared source program, solve them, and then use the solution to translate the desugared code into the core language, while adding type annotations. By approaching type inference as the solution of a set of constraints instead of a bottom-up traversal of the program's abstract syntax tree, we make it easier to dynamically reorder work as the need arises. This

framework also helps to manage our extra region, effect and closure information. Once we have a system for expressing and solving general type constraints, the fact that we have constraints of different kinds does not add much complexity to the system. Constraint based systems also naturally support the graphical effect and closure terms discussed in §2.3.8, as well as providing a convenient way to manage the information used to generate type error messages.

Our system has some similarity to the one use by ML^{F} [RY08], though we do not consider higher rank types. We believe our system could be extended to support them, but we have been mainly interested in the region, effect and closure information, and have not investigated it further. We also derive inspiration from Erwig's visual type inference [Erw06], and the graphs used by Duggan [DB96] to track the source of type errors. However, unlike their work we do not draw our type graphs pictorially. We have found that the addition of region, effect and closure information, along with the associated type class constraints, tends to make these two dimensional diagrams into "birds nests" with many crossing edges, which hinders the presentation. Instead, we simply write down the constraints as equations, and try to imagine the graph being separated into several two dimensional layers, one for each kind. Such graphs might make an interesting target for work on computer aided visualisation as we know of no tool to generate a suitably pleasing diagram.

3.2 Source language and constraint slurping

This section presents the formal description of our desugared source language and discusses how to generate type constraints. The typing rules given in this chapter serve as inspiration when generating constraints, but correctness for our overall system relies on the proof of soundness for the core language given in the appendix.

Any errors in constraint generation or type inference will be detected when checking the program once it has been translated to core. We believe this is a fair approach, because new type systems are usually presented for a cut down, desugared language anyway. We view the core type system as the "real" type system for Disciple, with the system presented in this chapter being part of the compiler implementation.

Declarations

$pgm \rightarrow$	$\overline{decl};\ t$		(program)
$decl \rightarrow$	**data** $T :: \% \rightarrow \overline{\kappa} \rightarrow *$ **where** $\overline{K : \varphi}$		(data type declaration)

Programs consist of a list of declarations followed by a term to be evaluated. Data type declarations introduce a new type constructor T, and give its kind. All type constructors T, and data constructors K defined in a program must be distinct. We define the meta-function ctorTypes(T) to get the list of data constructors $\overline{K : \varphi}$ corresponding to a particular type constructor T.

The set of allowable types for data constructors is more restrictive than indicated here. These restrictions are introduced by the typing rules presented in §3.2.8.

Kinds

κ	$\rightarrow$	$(\kappa_1 \rightarrow \kappa_2)$	(kind function)
	$\mid$	$* \mid \% \mid ! \mid \$$	(atomic kind constructors)

Kinds consist of kind functions and the atomic kinds $*$, %, ! and \$ which are the kinds of value types, regions, effects and closures respectively.

Types

$\varphi, \tau, \sigma, \varsigma$

$\rightarrow$	a_κ	(type variables)
$\mid$	$\varphi \rhd \Omega$	(constrained type)
$\mid$	$\forall(a:\kappa).\ \varphi$	(unbounded quantification)
$\mid$	$\varphi_1 \vee \varphi_2$	(least upper bound)
$\mid$	$\top_\kappa \mid \bot_\kappa$	(top and bottom)
$\mid$	$\tau_1 \xrightarrow{\sigma\ \varsigma} \tau_2$	(function type constructor)
$\mid$	$T_\kappa\ r\ \overline{\varphi}$	(data type constructor)
$\mid$	$\mathit{Read}\ r \mid \mathit{ReadH}\ \tau \mid \mathit{Write}\ r$	(effect type constructors)
$\mid$	$x:\varphi$	(closure constructor)

We do not make a rigid syntactic distinction between polytypes and monotypes, or constrained and unconstrained types. For this we are inspired by the pure type system approach of the lambda cube and the Henk intermediate language [PJM97]. We also do not make a syntactic distinction between value types, regions, closures and effects. We have found maintaining these distinctions to be cumbersome in both the presentation and implementation. However, we will hint at the intended kind of a particular type by using the variables φ, τ, σ and ς. We intend τ to be an unquantified value type, σ and ς to be unquantified effect and closure types respectively, and allow φ to be any type. a_k are type variables tagged with their kind, though we tend to elide their kinds in this presentation. As type variables contain their kinds, we can determine the kind of an arbitrary type expression without needing an auxiliary environment. When a specific kind is intended we use s, r, e and c as value type, region, effect and closure variables respectively.

Ω is a set of constraints. The term $\varphi \rhd \Omega$ is a constrained type whose general meaning is similar to $\Omega \Rightarrow \varphi$ in Haskell style systems deriving from type classes [WB89] and Jones's work on general qualified types [Jon92]. The expression $\varphi \rhd \Omega$ is pronounced "φ with Ω". We use the $\varphi \rhd \Omega$ form because we find it easier to read when there are a large number of constraints, and the order of constraints in the source language is irrelevant. Although Ω is a set, we usually write $\varphi \rhd \chi_1, \chi_2$ instead of $\varphi \rhd \{\chi_1, \chi_2\}$. We take $\varphi \rhd \emptyset$ as being equivalent to φ.

We use only unbounded quantification in the source language. In the type $\forall(a:\kappa).\varphi$ we usually elide the kind term when it is obvious from the name of the variable. For example, $\forall r_1.\varphi$ quantifies a region variable and $\forall e_1.\varphi$ quantifies an effect variable. We treat $\forall\overline{a:\kappa}.\ \varphi$ as short for $\overline{\forall a:\kappa}.\ \varphi$. We also treat expressions like $\forall r_{1..3}.\ \varphi$ as short for $\forall r_1\ r_2\ r_3.\ \varphi$. The operator $\rhd$ binds more tightly than $\forall$, so the type $\forall(a:\kappa).\ \varphi \rhd \Omega$ should be read as $\forall(a:\kappa).\ (\varphi \rhd \Omega)$. We do not consider higher ranked types, and assume that all quantified types are in prenex form.

The least upper bound $\varphi_1 \vee \varphi_2$ is defined on effect and closure types only. $\top_\kappa$ and $\bot_\kappa$ include their kinds and $\top$ may only be an effect. In the types presented to the user, $\bot_\kappa$ may be an effect or closure only, but during type inference we abuse the notation and use it as a value type and region type as well. The function type $\tau_1 \xrightarrow{\sigma\ \varsigma} \tau_1$ contains effect and closure annotations, but if these are not present we will assume they are $\bot$. Due to the form of the data type definitions, data type constructors $T_\kappa\ r\ \overline{\varphi}$ always have their primary region

variable as as their first parameter. The κ in the subscript is the constructor's kind.

Read r, *ReadH* τ and *Write* r are our initial effect types, though we will add more later. *ReadH* τ expresses a read on a data constructor's primary region, and we use it when generating type constraints for case expressions. $x : \varphi$ is a closure term tagged with a usefully named value variable. See §2.5 for a discussion of this.

Constraints

χ	$\rightarrow$	$\tau_1 = \tau_2$	(type equality)
	$\mid$	$\varphi_1 \sqsupseteq \varphi_2$	(effect or closure constraint)

Our initial type constraints are $\tau_1 = \tau_2$ and $\varphi_1 \sqsupseteq \varphi_2$, though we will add type class constraints later. Equality constraints like $\tau_1 = \tau_2$ are used to constrain value types and type variables of all kinds. Inequality constraints like $\varphi_1 \sqsupseteq \varphi_2$ are used to constrain effects and closures. When performing type checking we must allow the left of these constraints to be a full type, though in annotations and type schemes it is always a variable. When performing type inference and checking, all effect and closure constraints must be in the weak form discussed in §2.3.6. Types are strengthened only when presenting them to the user or converting them to the core language.

Terms

t	$\rightarrow$	x	(term variable)
	$\mid$	K	(data constructor)
	$\mid$	$\lambda x.\ t$	(term abstraction)
	$\mid$	$t_1\ t_2$	(term application)
	$\mid$	**let** $\overline{x = t}$ **in** t'	(let bindings)
	$\mid$	**case** t **of** $\overline{p \rightarrow t'}$	(case expression)

Patterns

p	$\rightarrow$	_	(wild card)
	$\mid$	$K\ \overline{x}$	(constructor pattern)

Derived Forms

if t_1 **then** t_2 **else** t_3	$\stackrel{\text{def}}{=}$	**case** t_1 **of** $\{\mathit{True} \rightarrow t_2;\ \mathit{False} \rightarrow t_3\}$
do $\overline{\mathit{bindstmt}}$; t	$\stackrel{\text{def}}{=}$	**let** $\overline{\text{mkBind}(\mathit{bindstmt})}$ **in** t
where *bindstmt*	$\rightarrow$	$x = t \mid t$
mkBind$(x = t)$	$\stackrel{\text{def}}{=}$	$x = t$
mkBind(t)	$\stackrel{\text{def}}{=}$	$x = t,\quad x$ fresh

Our term language is standard, with let bindings being mutually recursive. This is only a simple desugared language. Full Disciple is sweeter and includes pattern guards, kind inference, monadic do notation, and other features — but we do not discuss them here.

3.2.1 Normal types

Although the language definition provides a high degree of freedom when writing type expressions, the types presented to the programmer are all *normal*.

Consider the following type:

$$\begin{array}{ll} succ & :: \forall e_1\ r_1\ r_2.\ Int\ \ r_1 \xrightarrow{e_1} Int\ \ r_2 \\ & \rhd \quad e_1 \sqsupseteq Read\ \ r_1 \end{array}$$

We could also write this as:

$$\begin{array}{ll} succ & :: \forall e_1\ r_1\ r_2.\ s_1 \xrightarrow{e_1} s_2 \\ & \rhd \quad s_1 = Int\ \ r_1 \\ & , \quad s_2 = Int\ \ r_2 \\ & , \quad e_1 \sqsupseteq Read\ \ r_1 \end{array}$$

or:

$$\begin{array}{ll} succ & :: \forall e_1\ r_1\ r_2.\ (s_1 \rhd s_1 = Int\ \ r_1) \xrightarrow{e_1} s_2 \\ & \rhd \quad s_2 = Int\ \ r_2 \\ & , \quad e_1 \sqsupseteq Read\ \ r_1 \end{array}$$

All three of these types are equivalent, and perfectly valid in our system, though only the first is normal. The last two can appear as intermediate forms during type inference. Normal types obey the following rules:

1. Normal types are of the form $\forall \overline{a : \kappa}.\ \varphi \rhd \Omega$ where φ is not another constrained type like $\varphi_1 \rhd \Omega$. When this restriction is in place we refer to φ as the *body* of the type.

2. There are no $\tau_1 = \tau_2$ constraints, and the left positions of all $\varphi_1 \sqsupseteq \varphi_2$ constraints are variables.

3. For every constraint set Ω in the type, there is only one constraint per variable. For example, we write $\varphi \rhd e \sqsupseteq \sigma_1 \vee \sigma_2$ instead of $\varphi \rhd e \sqsupseteq \sigma_1,\ e \sqsupseteq \sigma_2$.

4. Normal types do not contain nested closure terms of the form $x_1 : x_2 : \varphi$. The value variables are used for documentation only, so we keep just the first one and write $x_1 : \varphi$ instead.

3.2.2 Free variables of types

The function for computing the free variables of a type is unsurprising:

$$
\begin{array}{ll}
fv(a) & = \{a\} \\
fv(\varphi \rhd \Omega) & = fv(\varphi) \setminus \{a \mid (a = \varphi') \in \Omega\} \\
fv(\forall(a : \kappa).\ \varphi) & = fv(\varphi) \setminus \{a\} \\
fv(\varphi \vee \varphi') & = fv(\varphi) \cup fv(\varphi') \\
fv(\bot) & = \emptyset \\
fv(\top) & = \emptyset \\
fv(\tau \xrightarrow{\sigma\ \varsigma} \tau') & = fv(\tau) \cup fv(\tau') \cup fv(\sigma) \cup fv(\varsigma) \\
fv(T_\kappa\ r\ \overline{\varphi}) & = \{r\} \cup \overline{fv(\varphi)} \\
fv(\mathit{Read}\ \ r) & = \{r\} \\
fv(\mathit{ReadH}\ \ \tau) & = fv(\tau) \\
fv(\mathit{Write}\ \ r) & = \{r\} \\
fv(x : \varphi) & = fv(\varphi)
\end{array}
$$

3.2.3 Dangerous variables

Dangerous variables were discussed in §2.5.1. To compute the dangerous variables of a type we use a domain D, where a member of this domain can be either a pair consisting of a set of constraints and a type expression $\langle\{\chi\}, \varphi\rangle$, or a dotted type variable $a^\bullet$.

$$D = \{\text{Constraint}\} \times \text{Type} \ + \ \text{DotVar}$$

To compute the dangerous variables in a particular type φ, we start with the set $\{\langle\emptyset, \varphi\rangle\}$ then iteratively apply the relation $dv : \{D\} \to \{D\}$ until we reach a fixpoint. In the resulting set, the dotted variables are the ones that are dangerous in the initial type.

The relation dv is as follows:

$$dv : \{D\} \to \{D\}$$
$$dv(\mu) = \mu \ \cup \ \bigcup\{ dv'(x) \mid x \in \mu \}$$

where

$$\begin{aligned}
&dv' : D \to \{D\} \\
&dv'(a^\bullet) &&= \{a^\bullet\} \\
&dv'(\langle \Omega,\ a \rangle) &&= \emptyset \\
&dv'(\langle \Omega,\ \varphi \rhd \Omega' \rangle) &&= \{\langle \Omega \cup \Omega',\ \varphi \rangle\} \\
&dv'(\langle \Omega,\ \varphi \vee \varphi' \rangle) &&= \{\langle \Omega,\ \varphi \rangle,\ \langle \Omega,\ \varphi' \rangle\} \\
&dv'(\langle \Omega,\ \bot \rangle) &&= \emptyset \\
&dv'(\langle \Omega,\ \top \rangle) &&= \emptyset \\
&dv'(\langle \Omega,\ \tau \xrightarrow{\sigma\ c} \tau' \rangle) &&= \{\langle \Omega,\ \varphi \rangle\} \\
&\quad \textbf{where} \quad (c \sqsupseteq \varphi) \in \Omega \\
&dv'(\langle \Omega,\ T_\kappa\ r\ \overline{\varphi} \rangle) \\
&\quad \mid \mathit{Mutable}\ \ r \in \Omega &&= \{a^\bullet \mid a \in \mathit{fv}(\overline{\varphi})\} \\
&\quad \mid \mathit{otherwise} &&= \{\langle \Omega, \tau[r\ \overline{\varphi}/\overline{a}] \rangle \mid \forall \overline{a}.\ \tau \in \mathit{args}(\mathit{ctorTypes}(T))\} \\
&dv'(\langle \Omega,\ \mathit{Read}\ \ r \rangle) &&= \emptyset \\
&dv'(\langle \Omega,\ \mathit{ReadH}\ \ \tau \rangle) &&= \emptyset \\
&dv'(\langle \Omega,\ \mathit{Write}\ \ r \rangle) &&= \emptyset \\
&dv'(\langle \Omega,\ x : \varphi \rangle) &&= \{\langle \Omega,\ \varphi \rangle\}
\end{aligned}$$

The *ctorTypes* function returns the types of the constructors associated with a particular data type constructor T. The *args* function returns the arguments of these constructors, retaining the outer quantifiers. As we only compute the dangerous variables of a type *before* generalising it, there is no need to match on the $\forall(a : \kappa).\varphi$ form. If we take the sets $\{D\}$ to be ordered by set inclusion, the function dv is monotonic by construction, as it always returns its argument as part of the result.

For example, suppose the type *TwoThings* is defined as:

$$\begin{aligned}
&\textbf{data}\ \ \mathit{TwoThings}\ \ r_{1..3}\ a\ b \\
&\quad = \quad T1\ (\mathit{Maybe}\ \ r_2\ a) \\
&\quad \mid \quad T2\ (\mathit{Maybe}\ \ r_3\ b)
\end{aligned}$$

In the desugared language this becomes:

$$\begin{aligned}
&\textbf{data}\ \ \mathit{TwoThings}\ \ r_{1..3}\ a\ b\ \textbf{where} \\
&\quad T1 :: \forall r_{1..3}\ a\ b.\ \ \mathit{Maybe}\ \ r_2\ a \to \mathit{TwoThings}\ \ r_{1..3}\ a\ b \\
&\quad T2 :: \forall r_{1..3}\ a\ b.\ \ \mathit{Maybe}\ \ r_3\ b \to \mathit{TwoThings}\ \ r_{1..3}\ a\ b
\end{aligned}$$

Now, if r_1 was mutable, then we could update both alternatives, so both a and b would be dangerous. However, if r_2 was mutable then a would be dangerous (but not necessarily b), and if r_3 was mutable then b would be dangerous (but not necessarily a).

With this definition, the value of $\mathit{args}(\mathit{ctorTypes}(\mathit{TwoThings}))$ is:

$$\{\forall r_{1..3}\ a\ b.\, \mathit{Maybe}\ \ r_2\ a,\ \forall r_{1..3}\ a\ b.\, \mathit{Maybe}\ \ r_3\ b\ \}$$

Suppose we wish to determine the dangerous variables in the type:

$$\mathit{TwoThings}\ r_4\ r_5\ r_6\ (\mathit{Int}\ r_7)\ (c \to d) \rhd \mathit{Mutable}\ r_5$$

The following is the sequence of states we get when computing the fixpoint. We save space by writing ... in place of the elements of the previous state.

$$\{\langle \emptyset, \mathit{TwoThings}\ r_4\ r_5\ r_6\ (\mathit{Int}\ r_7)\ (c \to d) \rhd \mathit{Mutable}\ r_5 \rangle\}$$

$$\vdash \{..., \langle \{\mathit{Mutable}\ r_5\}, \mathit{TwoThings}\ r_4\ r_5\ r_6\ (\mathit{Int}\ r_7)\ (c \to d)\rangle\}$$

$$\begin{aligned} \vdash \{..., & \langle \{\mathit{Mutable}\ r_5\}, \mathit{Maybe}\ r_5\ (\mathit{Int}\ r_7)\rangle \\ & , \langle \{\mathit{Mutable}\ r_5\}, \mathit{Maybe}\ r_6\ (c \to d)\rangle\ \} \end{aligned}$$

$$\vdash \{..., r_7^{\bullet}, c \to d\}$$

Hence, only r_7 is dangerous. The variables c and d would only be dangerous if r_6 or r_4 were mutable.

Computing the dangerous variables of a type using a fixpoint allows us to deal with recursive types. For example, the list type has the following desugared definition:

$$\begin{aligned} &\textbf{data}\ \mathit{List}\ r_1\ a\ \textbf{where} \\ &\quad \mathit{Nil} \quad :: \quad \forall r_1\ a.\, \mathit{List}\ r_1\ a \\ &\quad \mathit{Cons} \quad :: \quad \forall r_1\ a\ c_1.\, a \to \mathit{List}\ r_1\ a \xrightarrow{c_1} \mathit{List}\ r_1\ a \\ &\qquad\quad \rhd \quad c_1 \sqsupseteq x : a \end{aligned}$$

Here is the sequence of states we get when determining the dangerous variables in the type $\mathit{List}\ r_1\ (\mathit{Maybe}\ r_2\ c) \rhd \mathit{Mutable}\ r_2$

$$\{\langle \emptyset, \mathit{List}\ r_1\ (\mathit{Maybe}\ r_2\ c) \rhd \mathit{Mutable}\ r_2\rangle\}$$

$$\vdash \{..., \langle \{\mathit{Mutable}\ r_2\}, \mathit{List}\ r_1\ (\mathit{Maybe}\ r_2\ c)\rangle\}$$

Note that when dv' is applied to the last element in this set, it yields the following pairs:

$$\begin{aligned} \{ \quad & \langle \{\mathit{Mutable}\ r_2\}, \mathit{List}\ r_1\ (\mathit{Maybe}\ r_2\ c)\rangle \\ , \quad & \langle \{\mathit{Mutable}\ r_2\}, \mathit{Maybe}\ r_2\ c\rangle \qquad\qquad \} \end{aligned}$$

The second element here is new, but the first is was already present in the previous set, and arises due to the recursiveness of the *List* type.

Continuing on with the process, we obtain the following states, with the last one being the fixpoint.

$$\vdash \{..., \langle \{\mathit{Mutable}\ r_2\}, \mathit{Maybe}\ r_2\ c\rangle\}$$

$$\vdash \{..., c^{\bullet}\}$$

This shows us that the type variable c is dangerous in this type, as expected.

Problems with nested data types

Note that a direct implementation of the definition of *dv* will diverge when applied to a nested data type [BM98].[1] For example, consider the following type:

$$\begin{array}{l}\textbf{data}\ \mathit{Nest}\ r_1\ a\ \textbf{where}\\ \quad \mathit{MkNest}\ \ ::\ \ \forall r_1\ a.\, \mathit{Nest}\ r_1\ (\mathit{List}\ r_1\ a) \rightarrow \mathit{Nest}\ r_1\ a\end{array}$$

This type is considered "nested" because the recursive application of *Nest* in the first parameter of *MkNest* does not have the same form as that which is being defined, that is, it's not just another $\mathit{Nest}\ r_1\ a$. Here is what happens if we try to compute the dangerous variables in the type $\mathit{Nest}\ r_1\ a$:

$$\begin{array}{ll} & \{\langle \emptyset,\ \mathit{Nest}\ r_1\ a\rangle\}\\ \vdash & \{..., \langle \emptyset,\ \mathit{Nest}\ r_1\ (\mathit{List}\ r_1\ a\rangle\}\\ \vdash & \{..., \langle \emptyset,\ \mathit{Nest}\ r_1\ (\mathit{List}\ r_1\ (\mathit{List}\ r_1\ a))\rangle\}\\ \vdash & \{..., \langle \emptyset,\ \mathit{Nest}\ r_1\ (\mathit{List}\ r_1\ (\mathit{List}\ r_1\ (\mathit{List}\ r_1\ a)))\rangle\}\\ ... & \end{array}$$

The application of *dv* to each successive state yields a larger state, which causes our computation to diverge. However, in the limit, no dotted variables such as $r_1^\bullet$ will be produced, so there are no dangerous variables in the original type. To say this another way: although *dv* is sufficient to *define* the set of dangerous variables, it is not sufficient to compute it, if applied in a naive way.

As mentioned in [BM98], the use of nested data types in practice is rare. Also, important generic functions that operate on them (such as *fold*) need to be assigned rank-2 types, which we do not support either. We expect that computing the dangerous variables of a nested data type could be done using a reachability analysis instead of direct substitution. However, as checking whether a type is nested or not is straight forward, we have not investigated this further.

3.2.4 Material and immaterial variables

The difference between material and immaterial region variables was discussed in §2.5.3. Recall that material region variables correspond to physical objects in the store, whereas immaterial region variables are used to describe the locations of the parameter and return values of functions. In §2.5.7 we discussed how closure terms that do not contain material region variables can be *trimmed* out. In §2.5.5 we defined *strongly material* region variables to be the ones that appear in material positions, but not in immaterial ones. In §2.6.5 we discussed how the immaterial portions of objects cannot be copied.

The following functions, *mv*, *iv* are used to compute the material and immaterial variables of a type. The strongly material variables are then obtained by subtracting the second from the first. The type is required to be in normal form, which is described in §3.2.1. The *mv* and *iv* functions are defined similarly to the *dv* function from the previous section, and we use the same fixpoint process. Note that we classify isolated value type variables as material because they have the potential to be constrained to a type that contains material region variables, such as $\mathit{Int}\ r_1$.

[1] Thanks to one of my thesis examiners for pointing this out.

Material Variables

$$mv : \{D\} \to \{D\}$$
$$mv(\mu) = \mu \ \cup \ \bigcup\{mv'(x) \mid x \in \mu\}$$

where

$$mv' : D \to \{D\}$$
$$mv'(a^\bullet_\kappa) \qquad = \{a^\bullet_\kappa\}$$
$$mv'(\langle\Omega,\ a_\kappa\rangle)$$
$$\quad \mid \kappa \in \{\%,\ *\} \quad = \{a^\bullet_\kappa\}$$
$$\quad \mid otherwise \quad = \emptyset$$

$$mv'(\langle\Omega,\ T_\kappa\ r\ \overline{\varphi}\rangle)$$
$$\quad = \{r^\bullet\} \cup \{\langle\Omega, \tau[r\ \overline{\varphi}/\overline{a}]\rangle \mid \forall\overline{a}.\ \tau \in args(ctorTypes(T))\}$$

... other cases as per dv'

Immaterial Variables

$$iv : \{D\} \to \{D\}$$
$$iv(\mu) = \mu \ \cup \ \bigcup\{iv'(x) \mid x \in \mu\}$$

where

$$iv' : D \to \{D\}$$
$$iv'(a^\bullet_\kappa) \qquad = \{a^\bullet_\kappa\}$$
$$iv'(\langle\Omega,\ a_\kappa\rangle)$$
$$\quad \mid \kappa \in \{\%,\ *\} \quad = \emptyset$$
$$\quad \mid otherwise \quad = \{a^\bullet_\kappa\}$$

$$iv'(\langle\Omega,\ \tau \xrightarrow{e\ c} \tau'\rangle)$$
$$\quad = \{a^\bullet \mid a \in fv(\tau) \cup fv(\tau') \cup fv(\varphi) \cup \{e,\ c\}\} \ \cup \ \{\langle\Omega,\ \varphi'\rangle\}$$
$$\quad \textbf{where} \quad (e \sqsupseteq \varphi) \in \Omega,\ (c \sqsupseteq \varphi') \in \Omega$$

$$iv'(\langle\Omega,\ T_\kappa\ r\ \overline{\varphi}\rangle)$$
$$\quad = \{\langle\Omega, \tau[r\ \overline{\varphi}/\overline{a}]\rangle \mid \forall\overline{a}.\ \tau \in args(ctorTypes(T))\}$$

$$iv'(\langle\Omega,\ Read\ r\rangle) \qquad = \{r^\bullet\}$$
$$iv'(\langle\Omega,\ ReadH\ \tau\rangle) \qquad = \{a^\bullet \mid a \in fv(\tau)\}$$
$$iv'(\langle\Omega,\ Write\ r\rangle) \qquad = \{r^\bullet\}$$
$$iv'(\langle\Omega,\ x : \varphi\rangle) \qquad = \langle\Omega,\ \varphi\rangle$$

... other cases as per dv'

Examples:

τ	material	immaterial	strongly material
$Int\ r_1$	r_1	$\emptyset$	r_1
$List\ r_1\ a$	$r_1\ a$	$\emptyset$	$r_1\ a$
$a \rightarrow Int\ r_1$	$\emptyset$	$a\ r_1$	$\emptyset$
$() \xrightarrow{e_1\ c_1} ()$ $\rhd e_1 \sqsupseteq Write\ r_3$ $,\ c_1 \sqsupseteq x_1 : Int\ r_3\ \vee\ x_2 : Int\ r_4$	$r_3\ r_4$	$e_1\ c_1\ r_3$	r_4
$Int\ r_1 \xrightarrow{e_1\ c_1} Int\ r_2$ $\rhd e_1 \sqsupseteq Read\ r_1$ $,\ c_1 \sqsupseteq x : Int\ r_2$	r_2	$r_1\ r_2\ e_1\ c_1$	$\emptyset$
$Maybe\ r_1\ (a \xrightarrow{\perp\ c_1} Int\ r_3)$ $\rhd c_1 \sqsupseteq x : Int\ r_3$	$r_1\ r_3$	$a\ c_1\ r_3$	r_1
$List\ r_1\ (Tuple2\ r_2\ (Int\ r_3)\ (a \xrightarrow{e_1\ c_1} b))$ $\rhd e_1 \sqsupseteq Read\ r_3$ $,\ c_1 \sqsupseteq x : Int\ r_3$	$r_1\ r_2\ r_3$	$a\ b\ e_1\ c_1\ r_3$	$r_1\ r_2$

3.2.5 The *map* example

We will use the following program as a running example:

$$
\begin{array}{l}
\textbf{data}\ List :: \% \rightarrow * \rightarrow * \ \textbf{where} \\
\quad Nil \quad :: \forall(r : \%).\forall(a : *).\ List\ r\ a \\
\quad Cons \quad :: \forall(r : \%).\forall(a : *).\forall(c : \$).\ a \rightarrow List\ r\ a \xrightarrow{c} List\ r\ a \\
\qquad\quad \rhd c \sqsupseteq x : a \\
\\
\textbf{let} \quad map = \lambda f.\ \lambda xx\,. \\
\qquad\quad \textbf{case}\ xx\ \textbf{of} \\
\qquad\qquad Nil \qquad\quad \rightarrow Nil \\
\qquad\qquad Cons\ x\ xs \quad \rightarrow Cons\ (f\ x)\ (map\ f\ xs) \\
\textbf{in} \quad map\ succ\ foo
\end{array}
$$

This program defines the familiar *List* data type and *map* function, then applies *succ* to all elements of the list *foo*. We will assume that *succ* and *foo* are defined elsewhere and have the following types:

$$
\begin{array}{lll}
succ & :: & \forall r_{1..2}\ e_1.\ Int\ r_1 \xrightarrow{e_1} Int\ r_2 \\
 & \rhd & e_1 \sqsupseteq Read\, r_1 \\
 & & \\
foo & :: & List\ r_5\ (Int\ r_6)
\end{array}
$$

3.2.6 Annotated source language

The first step in type inference is to annotate the source program with fresh type variables to serve as hooks for the type constraints. Formally, we consider the annotated language to be an extension of the source language from the previous section, with the following additional productions:

Terms

$$
\begin{array}{lcll}
t & \rightarrow & \ldots & \\
 & | & \lambda(x : s_x).\ t & \text{(annotated term abstraction)} \\
 & | & \textbf{let } \overline{(x : s_x) = t} \textbf{ in } t' & \text{(annotated let bindings)}
\end{array}
$$

Patterns

$$
\begin{array}{lcll}
p & \rightarrow & \ldots & \\
 & | & K\ \overline{(x : s_x)} & \text{(annotated constructor pattern)}
\end{array}
$$

Annotations are placed on let and lambda bound variables, as well as variables that are bound by a pattern match. Although the annotated language is conceptually separate from the source language, in our practical implementation we represent them with the same data type.

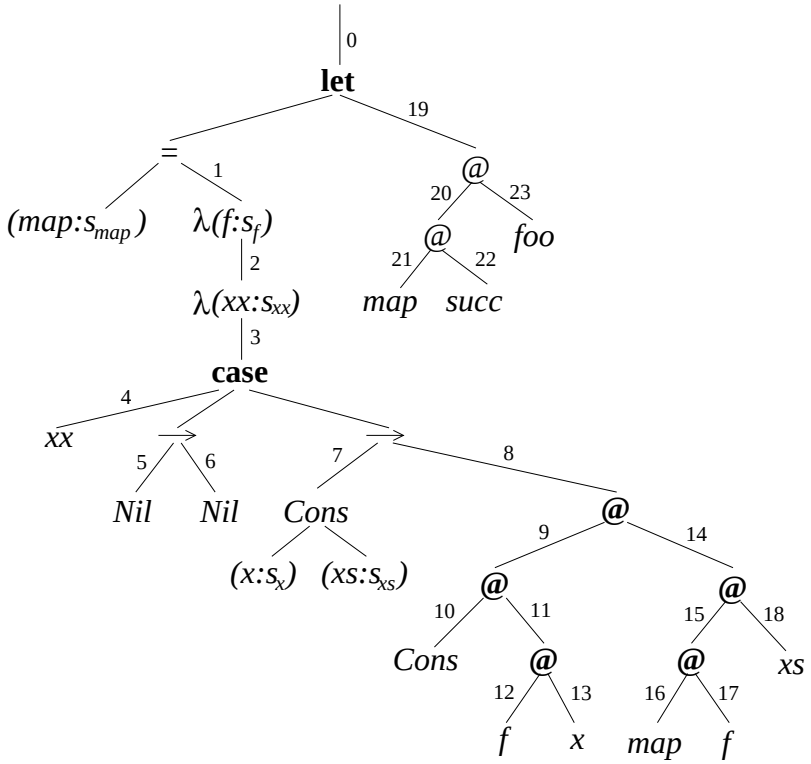
When we add fresh variables, the body of our example program becomes:

$$
\begin{array}{lll}
\textbf{let} & (map : s_{map}) = \lambda(f : s_f).\ \lambda(xx : s_{xx}). & \\
 & \quad \textbf{case } xx \textbf{ of} & \\
 & \quad\quad Nil & \rightarrow Nil \\
 & \quad\quad Cons\ (x : s_x)\ (xs : s_{xs}) & \rightarrow Cons\ (f\ x)\ (map\ \ f\ \ xs) \\
\textbf{in} & \quad map\ \ succ\ \ foo &
\end{array}
$$

Note that we have named the fresh type variables after the value variables they represent. We can imagine that there is a mapping between corresponding value and type variables, and any type variable named after a value variable in the same example is assumed to map to it. We avoid introducing this mapping explicitly to reduce clutter in the presentation. We will also assume that all variables have unique names, so we can easily convert between the two.

When performing inference by hand, we draw the abstract syntax tree for the annotated program. Each of the edges in the tree is given a unique number, and we will use these numbers to name the type variables in the generated constraints. For example, we will name the type of the whole case-expression s_3, and its effect e_3.

The *map* example



let $(map : s_{map}) = \lambda(f : s_f).\ \lambda(xx : s_{xx}).$
  case xx **of**
    Nil $\rightarrow Nil$
    $Cons\ (x : s_x)\ (xs : s_{xs})$ $\rightarrow Cons\ (f\ x)\ (map\ \ f\ \ xs)$
in $map\ \ succ\ \ foo$

3.2.7 Slurping and constraint trees

In DDC we call the process of extracting type constraints from the annotated syntax tree *slurping* the tree. The function **SLURP** takes this syntax tree and produces the corresponding constraint tree:

$$\textbf{SLURP} : \mathit{SyntaxTree} \rightarrow \mathit{ConstraintTree}$$

If the syntax tree has already been annotated with type variables and edge numbers, then the constraints for each node can be produced independently. However, in our implementation we prefer to annotate the tree and generate constraints in a single, bottom-up pass.

The type constraints extracted from the program's syntax tree are represented by another tree that mirrors its overall shape. We use ϕ to represent a branch in this tree, and the branches have the following structure:

ϕ	$\rightarrow$	INST x	(instantiate this var)
	$\mid$	LAMBDA $\overline{x}$ $\overline{\phi}$	(lambda or case bound var)
	$\mid$	LET x $\overline{\phi}$	(let bound var)
	$\mid$	GROUP $\overline{x}$ $\overline{\phi}$	(group of let bindings)
	$\mid$	$a = \varphi$	(type equality)
	$\mid$	$a \sqsupseteq \varphi$	(effect or closure inequality)

INST x corresponds to an occurrence of a bound variable in the program source. When extracting constraints we generate an INST x for every occurrence, irrespective of whether the variable was bound by a let binding, lambda abstraction or pattern match. LAMBDA $\overline{x}$ $\overline{\phi}$ contains constraints arising from a lambda abstraction or pattern match. $\overline{x}$ is the list of bound variables and $\overline{\phi}$ is a list of constraint branches from the body of the abstraction. LET x $\overline{\phi}$ contains constraints arising from a let binding. GROUP $\overline{x}$ $\overline{\phi}$ contains all the constraint branches from a particular mutually recursive let expression. $a = \varphi$ and $a \sqsupseteq \varphi$ are individual constraints on type variables.

3.2.8 Types of programs and declarations

$$\boxed{\Gamma \vdash \mathit{pgm} :: \varphi}$$

$$\frac{\overline{\Gamma \vdash \mathit{decl} :: \Gamma_d} \quad \Gamma \vdash t :: \varphi \quad \Gamma = \Gamma_o, \overline{\Gamma_d}}{\Gamma_o \vdash \overline{\mathit{decl}} \ ; \ t :: \varphi} \qquad \text{(Pgm)}$$

$$\boxed{\Gamma \vdash \mathit{decl} :: \Gamma'}$$

$$\frac{\overline{\text{ValidCtor}(T,\ \% \to \overline{\kappa} \to *,\ \varphi)}}{\Gamma \vdash \textbf{data}\ T : \% \to \overline{\kappa} \to * \ \textbf{where}\ \overline{K : \varphi} \ ::\ (T : \% \to \overline{\kappa} \to *,\ \overline{K : \varphi})} \qquad \text{(DeclData)}$$

$$\begin{array}{l}\text{ValidCtor}(T,\ \% \to \overline{\kappa} \to *,\ \varphi) \\ \quad \text{where} \quad \varphi = \forall (r : \%)\ \overline{a : \kappa}.\ T\ r\ \overline{a}\end{array}$$

$$\begin{array}{l}\text{ValidCtor}(T,\ \% \to \overline{\kappa} \to *,\ \varphi) \\ \quad \text{where} \quad \varphi = \forall (r : \%)\ \overline{a : \kappa}.\ \tau \to T\ r\ \overline{a} \\ \qquad\quad\ \ \mathit{fv}(\tau) \setminus \{r,\ \overline{a}\} \subseteq \emptyset\end{array}$$

$$\begin{array}{l}\text{ValidCtor}(T,\ \% \to \overline{\kappa} \to *,\ \varphi) \\ \quad \text{where} \quad \varphi = \forall (r : \%)\ \overline{a : \kappa}\ (c : \$).\ \tau_1 \to \tau_2 \xrightarrow{c} T\ r\ \overline{a}\ \rhd c \sqsupseteq x_1 : \tau_1 \\ \qquad\quad\ \ (\mathit{fv}(\tau_1) \cup \mathit{fv}(\tau_2)) \setminus \{r,\ \overline{a}\} \subseteq \emptyset\end{array}$$

In (Pgm), we set the overall type of the program to be the type of its final expression. As data type declarations can be mutually recursive, we add the types and kinds generated by each one to the type environment used when checking them.

In (DeclData) the predicate ValidCtor checks that each constructor has a type appropriate to the data type being declared. We have given the first few cases of ValidCtor, and leave the inductive generalisation to the reader. In our implementation we *generate* the types of data constructors from Haskell style algebraic type definitions, instead of requiring the programmer to give them explicitly, but the checking rules are easier to present.

The definition of ValidCtor has several points of note: the type of a constructor cannot have free variables; the type of the return value must have a primary region variable; the types of parameters cannot contain variables that are not present in the return type, and constructors do not have side effects. Also note that the function arrows of constructor types must have appropriate closure annotations, the last case of ValidCtor is an example. This is needed to support the partial application of data constructors.

3.2.9 Kinds of types and constraints

$$\boxed{\varphi :: \kappa}$$

$$a_\kappa :: \kappa \qquad \text{(KiVar)}$$

$$\frac{\varphi :: \kappa \quad \overline{\chi :: \kappa'} \quad \chi \in \Omega}{\varphi \rhd \Omega :: \kappa} \qquad \text{(KiConstr)}$$

$$\frac{\varphi :: \kappa'}{\forall (a_\kappa : \kappa).\ \varphi :: \kappa'} \qquad \text{(KiAll)}$$

$$\frac{\varphi_1 :: \kappa \quad \varphi_2 :: \kappa \quad \kappa \in \{\ !,\ \$\ \}}{\varphi_1 \vee \varphi_2 :: \kappa} \qquad \text{(KiJoin)}$$

$$\frac{\kappa \in \{\ !,\ \$\ \}}{\bot_\kappa :: \kappa} \qquad \text{(KiBot)}$$

$$\top_! :: ! \qquad \text{(KiTop)}$$

$$\frac{\tau_1 :: * \quad \tau_2 :: * \quad \sigma :: ! \quad \varsigma :: \$}{\tau_1 \xrightarrow{\sigma\ \varsigma} \tau_2 :: *} \qquad \text{(KiFun)}$$

$$\frac{r :: \% \quad \overline{\varphi :: \kappa}}{T_{\%\to\overline{\kappa}\to *}\ \ r\ \ \overline{\varphi} :: *} \qquad \text{(KiData)}$$

$$\frac{r :: \%}{\mathit{Read}\ \ r :: !} \qquad \text{(KiRead)}$$

$$\frac{\varphi :: *}{\mathit{ReadH}\ \ \varphi :: !} \qquad \text{(KiReadH)}$$

$$\frac{r :: \%}{\mathit{Write}\ \ r :: !} \qquad \text{(KiWrite)}$$

$$\frac{\varphi :: \kappa \quad \kappa \in \{\ *,\ \$\ \}}{(x : \varphi) :: \$} \qquad \text{(KiClo)}$$

$$\boxed{\chi :: \kappa}$$

$$\frac{\tau_1 :: \kappa \quad \tau_2 :: \kappa}{(\tau_1 = \tau_2) :: \kappa} \qquad \text{(KiCEq)}$$

$$\frac{\varphi_1 :: \kappa \quad \varphi_2 :: \kappa \quad \kappa \in \{\ !,\ \$\ \}}{(\varphi_1 \sqsupseteq \varphi_2) :: \kappa} \qquad \text{(KiCGeq)}$$

Our kinding rules are mostly standard. In (KiConstr) we use the term $\overline{\chi :: \kappa'}$ to require each of the constraints to have a valid kind. The $\chi :: \kappa$ judgement ensures that the types on both sides of a constraint have the same kind.

3.2.10 Types of terms

$$\boxed{\Gamma \vdash t :: \varphi \rhd \Omega \; ; \; \sigma}$$

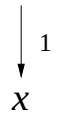
The judgement form $\Gamma \vdash t :: \varphi \rhd \Omega \; ; \sigma$ reads: "with environment Γ the term t has type φ, constraints Ω and effect σ." We will assume that φ contains no further constraint sets, and that the typing rules maintain this property. This is a slight abuse of $\rhd$, but we find it more convenient than introducing another operator. Our handling of constraints is based on Leroy's closure typing system [LW91], so the constraint set Ω is global. When building a type scheme we will include only the constraints reachable from the body of the type. Leroy's approach can be contrasted with Jones's system of qualified types [Jon92] which encodes constraints as bounds on quantifiers, and uses separate rules to move them between local types and the global set. Our core language uses this second system instead, and we convert between the two representations when translating the source program to core.

In our typing rules we make no attempt to keep the constraint set consistent or satisfiable. Inconsistencies such as $\mathit{Int}\ r \rhd \mathit{Mutable}\ r$, $\mathit{Const}\ r$ or $\bot \sqsupseteq \mathit{Console}$ will be discovered when the program is converted to core. The core typing rules ensure that witnesses to the mutability and constancy of a particular region cannot exist in the same program, and effect constraints are checked during type application. Attempting to translate a program that includes inconsistent type constraints to the core language will result in a core type error. However, if these problems are instead detected during type inference, then the compiler would be in a better position to emit a helpful error message. Error handling is discussed in §3.8.

The typing rules are presented in three parts, with the static rule in the center, the associated node of the abstract syntax tree on the left, and the generated type constraints on the right. The combination of node and type constraints inductively defines the **SLURP** function mentioned in §3.2.7.

Var / Ctor

$$\frac{x : \forall \overline{a : \kappa}.\ \varphi \rhd \Omega \in \Gamma}{\Gamma \vdash x :: \varphi[\overline{\varphi'/a}] \rhd \Omega[\overline{\varphi'/a}] \cup \Omega' \ ;\ \bot}$$



$$s_1 \quad = \text{INST}\ s_x$$

We assume that the source program's syntax has already been checked, so x is bound somewhere above its use. The rule for data constructors is identical to the one above, with x replaced by K.

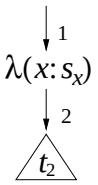
The type for x is required to be in the environment, and this type may include quantifiers $\forall \overline{a : \kappa}$ and more constraints Ω. We instantiate this type scheme by substituting new types $\overline{b}$ for the quantified variables in the body of the type as well as its constraints. The extra constraint term Ω' is needed to match the constraints introduced by other parts of the program, and allows the instantiated type to be weakened and treated as having a larger effect or closure term than it does in the environment. This is required when typing the higher order examples discussed in §2.3.6.

When generating constraints we defer the question of whether the variable was introduced by a let binding, lambda binding, pattern match, or whether it is part of a (mutually) recursive group. If a variable turns out to have been bound by a lambda or pattern match there will be no corresponding generalisation of its type, but we will use INST to instantiate it anyway. This makes the resulting constraints easier to read, and simplifies discussion of how to work out the binding dependency graph in §3.6. During type inference we can think of INST as a function that blocks on the variable s_x, waiting for the type scheme of x to be become available.

Abs

$$\frac{\Gamma,\ x : \tau_1 \rhd \Omega_1 \ \vdash\ t_2 :: \tau_2 \rhd \Omega_2 \ ;\ \sigma_2}{\Gamma \vdash \lambda x.\ t_2 \ ::\ \tau_1 \xrightarrow{e_2\ c_1} \tau_2 \rhd \Omega_1 \cup \Omega_2;\ \bot}$$

where for all $y \in fv(\lambda x.\ t_2)$ we have $(c_1 \sqsupseteq y : \Gamma(y)) \in \Omega_2$
and $e_2 \sqsupseteq \sigma_2 \in \Omega_2$



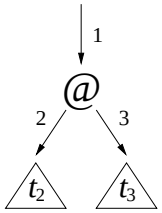
LAMBDA $\{x\}$
$s_1 \quad = s_x \xrightarrow{e_2\ c_1} s_2$
$c_1 \quad \sqsupseteq y_0 : s_{y0} \ \vee\ y_1 : s_{y1} \ \vee \ldots$
where $y_n \leftarrow fv(\lambda x.\ t_2)$
SLURP(t_2)

An abstraction takes a term of type τ_1 and produces a term of type τ_2. When the abstraction is applied it will have the effect σ_2 of its body. In the typing rule we give this effect the name e_2 and bind it to σ_2 in Ω_2. When generating constraints we can simply annotate the function constructor with e_2, and the required effect constraints will be generated when slurping the body. As evaluating the abstraction itself causes no effect, we have $\bot$ in the conclusion of the rule.

The closure of an abstraction contains the types of its free variables. In the typing rule we can read these types directly from the environment using $\Gamma(y)$. When we're generating constraints we won't know what these types are yet, so we use the the place holder variables $s_{y0}, s_{y1}...$ instead. These variables will be bound to their real types during inference.

App

$$\frac{\Gamma \vdash t_2 :: \tau_3 \xrightarrow{\sigma_4\ \varsigma_4} \tau_1 \rhd \Omega;\ \sigma_2 \qquad \Gamma \vdash t_3 :: \tau_3 \rhd \Omega\ ;\ \sigma_3}{\Gamma \vdash t_2\ t_3 :: \tau_1 \rhd \Omega\ ;\ \sigma_2 \vee \sigma_3 \vee \sigma_4}$$



$$\begin{aligned} s_2 &= s_3 \xrightarrow{e_4\ c_4} s_1 \\ e_1 &\sqsupseteq e_2 \vee e_3 \vee e_4 \end{aligned}$$

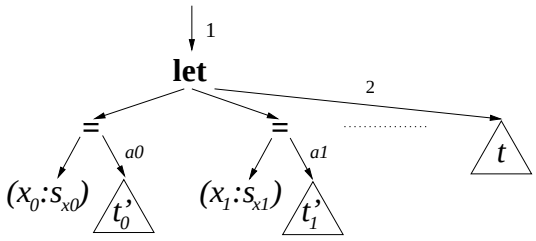
$e_4,\ c_4$ fresh
SLURP(t_2)
SLURP(t_3)

An application node applies a function of type $\tau_3 \xrightarrow{\sigma_4\ \varsigma_4} \tau_1$ to its argument of type τ_3, yielding a result of type τ_1. The act of applying the function has an effect σ_4. The effect of evaluating the entire expression consists of the effect of evaluating the function value, of evaluating the argument, and of applying the function. In the terminology of [LG88], σ_4 is the *intrinsic* effect of the application and $\sigma_2 \vee \sigma_3$ is the *inherited* effect. The closure of the function is of no consequence when typing an application, so ς_4 is only mentioned once in the rule.

When generating type constraints we will not yet know what the effect of the function will be. In our constraints we use e_4 and c_4 as local names for the function's effect and closure. These will be bound to the actual effect and closure of the function during type inference.

Let-Poly

$$\frac{\begin{array}{ll} \Gamma,\ \overline{x_n : \varphi_n} \vdash t :: \tau \rhd \Omega\ ;\ \sigma & \\ \Gamma,\ \overline{x_n : \tau_n \rhd \Omega} \vdash t'_0 :: \tau'_0 \rhd \Omega\ ;\ \sigma'_0 & \varphi_0 = \mathrm{Gen}(\Gamma,\ \tau'_0 \rhd \Omega) \\ \Gamma,\ \overline{x_n : \tau_n \rhd \Omega} \vdash t'_1 :: \tau'_1 \rhd \Omega\ ;\ \sigma'_1 & \varphi_1 = \mathrm{Gen}(\Gamma,\ \tau'_1 \rhd \Omega) \\ \vdots & \vdots \end{array}}{\Gamma \vdash \textbf{let}\ \overline{x_n = t'_n}\ \textbf{in}\ t :: \tau \rhd \Omega\ ;\ \sigma \vee \sigma'_0 \vee \sigma'_1 \vee \ldots}$$



GROUP $\{x_0,\ x_1,\ \ldots\}$
$s_1 = s_2$
$e_1 \sqsupseteq e_{a0} \vee e_{a1} \vee \cdots \vee e_2$
LET x_0
$s_{x0} = s_{a0}$
SLURP(t'_0)
LET x_1
$s_{x1} = s_{a1}$
SLURP(t'_1)
$\vdots$
SLURP(t)

The function Gen generalises the types of each binding. This process is discussed in §3.4. Note that in the expression:

$$\varphi = \mathrm{Gen}(\Gamma,\ \tau\ \rhd \Omega)$$

The resulting type φ contains only the constraints from Ω that are reachable from τ. The conclusion of (Let-Poly) includes the constraint set Ω, and the same set is used in each of the premises. This means that constraints that are conceptually local to a particular binding will "leak" into the global set. For example:

$$\frac{\begin{array}{l} \Gamma,\ \mathit{succL} : \forall r_1\ r_2.\ \mathit{Int}\ r_1 \rightarrow \mathit{Int}\ r_2 \rhd \mathit{Const}\ r_1 \\ \qquad \vdash \mathit{succL}\ 3 \\ \qquad :: \mathit{Int}\ r_3\ \rhd\ \mathit{Const}\ r_3,\ \mathit{Const}\ r_4\ ;\ \bot \\ \Gamma,\ \mathit{succL} : \mathit{Int}\ r_4 \rightarrow \mathit{Int}\ r_5 \rhd \mathit{Const}\ r_3,\ \mathit{Const}\ r_4 \\ \qquad \vdash \lambda x.\ \mathit{suspend1}\ \ \mathit{succ}\ x \\ \qquad :: \mathit{Int}\ r_4 \rightarrow \mathit{Int}\ r_5 \rhd \mathit{Const}\ r_3,\ \mathit{Const}\ r_4\ ;\ \bot \end{array}}{\begin{array}{l} \Gamma \vdash \textbf{let}\ \mathit{succL} = \lambda x.\ \mathit{suspend1}\ \ \mathit{succ}\ x\ \textbf{in}\ \mathit{succL}\ 3 \\ \quad :: \mathit{Int}\ r_3 \rhd \mathit{Const}\ r_3, \mathit{Const}\ r_4\ ;\ \bot \end{array}}$$

The function *succL* is a lazy version of *succ* that reads its argument only when the result is demanded. The general type of *succL* is:

$$\mathit{succL} :: \forall r_1\ r_2.\ \mathit{Int}\ r_1 \rightarrow \mathit{Int}\ r_2 \rhd \mathit{Const}\ r_1$$

The constraint $\mathit{Const}\ r_1$ arises from the use of *suspend1* in the definition of *succL*.

When checking this definition we give *succL* the monotype:

$$Int\ r_4 \rightarrow Int\ r_5 \rhd Const\ r_4$$

Due to the formulation of the (Let-Poly) rule, the constraint $Const\ r_4$ is actually present in *both* premises, as well as the conclusion. Also, in the body of the let-expression, the application of *succL* to the constant 3 requires that constant to be (really) constant, hence the constraint $Const\ r_3$. Although this constraint only concerns the body of the let-expression, it is also present in the set used when typing the bindings.

This behavior is unlike that of the (Let) rule presented by Leroy in [LW91]. Leroy's rule uses Gen to split the constraint set arising from a let-binding into two subsets: those that are reachable from the body of the type being generalised, and those that aren't. If we were to use Leroy's approach, the first premise and conclusion of our example would not contain $Const\ r_4$, and the second premise would not contain $Const\ r_3$. Leroy's rule is "nicer" when drawing proof trees, but we stick to the leaky version because it mirrors what happens during type inference. Our inference algorithm adds all the type constraints extracted from the program into a global graph, solves them, then returns the whole graph. It does not section the graph into portions relating to individual bindings, and it only removes constraints from the graph when dealing with the type classes discussed in §3.7. Retaining information from all bindings also makes it easy for the implementation to add type annotations to the desugared program when converting it to core.

As we have not implemented polymorphic recursion [Myc84], we check the right of each binding using the ungeneralised types for each let-bound variable. Due to this, many useful programs are not directly typeable with this (Let-Poly) rule. Consider this example from [Myc84]:

$$\begin{array}{ll}
\textbf{let} & map = \lambda f.\ \lambda xx\ . \\
& \quad \textbf{case}\ xx\ \textbf{of} \\
& \quad\quad Nil \rightarrow Nil \\
& \quad\quad Cons\ x\ xs \rightarrow Cons\ (f\ x)\ (map\ f\ xs) \\
& squarelist = \lambda l.\ map\ (\lambda x.\ x * x)\ l \\
& complement = \lambda l.\ map\ (\lambda x.\ not\ x)\ l \\
\textbf{in} & \ldots
\end{array}$$

This program will not be accepted as it stands. We need to use the generalised, polymorphic type of map when applying it to $(\lambda x.\ x * x)$ and $(\lambda x.\ not\ x)$ because these expressions have different types. If we use the ungeneralised, monomorphic type then we will get an error.

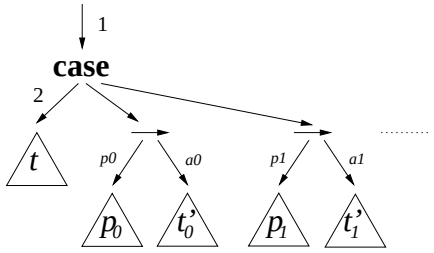
This highlights the fact that our source typing rules are only a guide for generating type constraints, and that we cannot use them to check the source program directly. We must first perform type inference by extracting type constraints and then solving them. As discussed in §3.6, our algorithm for solving type constraints also builds a graph that records what bindings are mutually recursive. Once we have this graph we can use it to split out the definition of *map* from the above example, and convert the program to:

$$
\begin{array}{ll}
\textbf{let} & map = \lambda f.\ \lambda xx\,. \\
& \quad \textbf{case}\ xx\ \textbf{of} \\
& \qquad \begin{array}{ll} Nil & \to Nil \\ Cons\ \ x\ \ xs & \to Cons\ (f\ x)\ (map\ \ f\ \ xs) \end{array} \\
\textbf{in} & \\
\textbf{let} & squarelist \quad = \lambda l.\, map\ (\lambda x.\ x * x)\ l \\
& complement = \lambda l.\, map\ (\lambda x.\, not\, x)\ l \\
\textbf{in} & \ldots
\end{array}
$$

For this version, (Let-Poly) allows us to use the generalised type of *map* when checking the body of the second let-expression. This new program will be accepted without error.

Case

$$
\frac{\begin{array}{r} \Gamma \vdash_{\mathrm{p}} p_0 \to t_0' :: T\ r\ \overline{\varphi} \to \tau \rhd \Omega\ ;\ \sigma_0' \\ \Gamma \vdash_{\mathrm{p}} p_1 \to t_1' :: T\ r\ \overline{\varphi} \to \tau \rhd \Omega\ ;\ \sigma_1' \\ \vdots \\ \Gamma \vdash t :: T\ r\ \overline{\varphi} \rhd \Omega\ ;\ \sigma \end{array}}{\Gamma \vdash \textbf{case}\ t\ \textbf{of}\ \overline{p \to t'} :: \tau' \rhd \Omega\ ;\ Read\ \ r \vee \sigma\ \ \vee \sigma_0' \vee \sigma_1' \vee \ldots}
$$



$$
\begin{array}{ll}
s_2 & = s_{p0} \\
s_2 & = s_{p1} \\
& \vdots \\
s_1 & = s_{a0} \\
s_1 & = s_{a1} \\
& \vdots \\
e_1 & = ReadH\ s_2 \vee e_2 \vee e_{a0} \vee e_{a1} \vee \ldots \\
\multicolumn{2}{l}{\textbf{SLURP}(t)} \\
\multicolumn{2}{l}{\textbf{SLURP}(p_0)} \\
\multicolumn{2}{l}{\textbf{SLURP}(p_1)} \\
& \vdots \\
\multicolumn{2}{l}{\textbf{SLURP}(t_0)} \\
\multicolumn{2}{l}{\textbf{SLURP}(t_1)} \\
& \vdots
\end{array}
$$

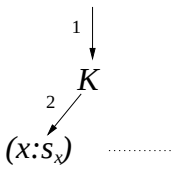
A case expression requires the discriminant t to have the same type as the patterns being matched against. For all alternatives, the types of the patterns must be identical, and so must the types of the expressions. The type of the entire case expression is the type of the right of the alternatives.

The effect of a case expression includes the effect of evaluating the discriminant and examining it, as well as evaluating the alternatives. When type checking a program in a bottom-up manner, when it's time to apply the (Case) rule we will already know the type of the discriminant. In this situation we can use $Read\ r$ as the effect of examining it. On the other hand, when generating constraints we will not yet know the type of the discriminant. We instead use $ReadH\ s_2$, which represents a read effect on the primary region of the (currently unknown) type s_2. During inference, the type of s_2 will resolve to the real type of the discriminant. After this is done, $ReadH\ s_2$ can be reduced to a *Read* effect on the primary region of this new type.

$$\boxed{\Gamma \vdash_{\mathrm{p}} p \to t :: \tau_1 \to \tau_2 \rhd \Omega}$$

$$\frac{\Gamma \vdash t :: \tau_2 \rhd \Omega \ ; \ \sigma}{\Gamma \vdash_{\mathrm{p}} \_ \to t :: \tau_1 \to \tau_2 \rhd \Omega \ ; \ \sigma} \qquad \text{(Pat-Wildcard)}$$

$$\frac{\begin{array}{l} T : \% \to \overline{\kappa} \to * \in \Gamma \\ K : \forall (r : \%)\ \overline{a : \kappa}\ \overline{c : \$}.\ \overline{\tau} \xrightarrow{c'} T\ r\ \overline{a}\ \rhd \Omega \in \Gamma \qquad \theta = [\ r'/r\ \ \overline{\varphi/a}\ ] \\ \Gamma,\ \overline{x_n : \theta(\tau_n)\ \rhd\ \theta(\Omega)}^{\,n} \vdash t :: \tau' \rhd \Omega' \ ; \ \sigma \end{array}}{\Gamma \vdash_{\mathrm{p}} K\ \overline{x} \to t :: T\ r'\ \overline{\varphi} \to \tau' \rhd \Omega' \ ; \sigma} \qquad \text{(Pat-Constructor)}$$



$$\begin{aligned} s_1 &= T\ r'\ \overline{a'} \\ s_x &= \tau' \\ &\vdots \\ &\text{where } \tau' \to \cdots \to T\ r'\ \overline{a'} \\ &\qquad = \mathrm{Inst}(\forall (r : \%)\ \overline{a : \kappa}\ \overline{c : \$}.\ \overline{\tau} \xrightarrow{c'} T\ r\ \overline{a}) \end{aligned}$$

The judgement form $\Gamma \vdash_{\mathrm{p}} p \to t :: \tau_1 \to \tau_2 \rhd \Omega$ reads: "with environment Γ an alternative matching a pattern p and producing a term t has type τ_1 to τ_2 with constraints Ω."

Matching against a wildcard produces no constraints.

In (Pat-Constructor) we lookup the type of the constructor K from the environment. The (DeclData) rule from §3.2.8 introduces these types into the environment and ensures that they have the particular form shown here.

The variables bound by the pattern are named $\overline{x}$, and the types of these variables must have the same form as the types of the arguments of the constructor. If the constructor produces a type containing variables $\overline{a}$ then all occurrences of $\overline{x}$ must agree on the particular types used for $\overline{a}$. For example with the constructor:

$$\mathit{Cons} :: \forall r\ a\ c.\ a \to \mathit{List}\ r\ a \xrightarrow{c} \mathit{List}\ r\ a\ \rhd\ c \sqsupseteq x : a$$

Consider the alternative in:

case ... **of**
  Cons *x* *xs* → ...

We cannot, say, use x at type $\mathit{Int}\ r_1$, but xs at type $\mathit{Cons}\ r_2\ (\mathit{Bool}\ r_1)$ because $\mathit{Int}\ r_1 \neq \mathit{Bool}\ r_1$. This restriction is achieved by requiring the types of each of the pattern bound variables to be related by the substitution θ.

The constraint generation rules given here only concern the pattern in a particular alternative. The job of matching up the types of all alternatives in a case-expression is handled by the constraints for the (Case) rule on the previous page.

When generating constraints for a pattern, we first take a fresh instance of the constructor's type scheme. The result type is the type of the overall pattern, and the argument types are assigned to the variables bound by the pattern.

3.2.11 Example: type constraints

The following contraint tree is for our *map* example:

GROUP $\{map\}$

$s_0 = s_{19}$

$e_0 \sqsupseteq e_1 \vee e_{19}$

LET *map*

$s_{map} = s_1$

LAMBDA $\{f\}$

$s_1 = s_f \xrightarrow{e_2\ c_1} s_2$

$c_1 \sqsupseteq map : s_{map}$

LAMBDA $\{xx\}$

$s_2 = s_{xx} \xrightarrow{e_3\ c_2} s_3$ $\quad c_2 \sqsupseteq map : s_{map} \vee f : s_f$

$s_3 = s_6$ $\quad e_3 \sqsupseteq ReadH\ s_4 \vee e_6 \vee e_7$

$s_3 = s_8$

$s_4 = s_5$

$s_4 = s_7$

$s_5 = List\ r_5\ a_5$

$s_7 = List\ r_7\ a_7$

$s_x = a_7$

$s_{xs} = List\ r_7\ a_7$

$s_4 = \text{INST}\ s_{xx}$

LAMBDA $\emptyset$

$s_6 = \text{INST}\ s_{Nil}$

LAMBDA $\{x,\ xs\}$

$s_9 = s_{14} \xrightarrow{e_{8a}\ c_8} s_8$ $\quad e_8 \sqsupseteq e_9 \vee e_{14} \vee e_{8a}$

$s_{10} = s_{11} \xrightarrow{e_{9a}\ c_9} s_9$ $\quad e_9 \sqsupseteq e_{10} \vee e_{11} \vee e_{9a}$

$s_{10} = \text{INST}\ s_{Cons}$

$s_{12} = s_{13} \xrightarrow{e_{11a}\ c_{11}} s_{11}$ $\quad e_{11} \sqsupseteq e_{12} \vee e_{13} \vee e_{11a}$

$s_{12} = \text{INST}\ s_f$

$s_{13} = \text{INST}\ s_x$

$s_{15} = s_{18} \xrightarrow{e_{14a}\ c_{14}} s_{14}$ $\quad e_{14} \sqsupseteq e_{15} \vee e_{18} \vee e_{14a}$

$s_{16} = s_{17} \xrightarrow{e_{15a}\ c_{15}} s_{15}$ $\quad e_{15} \sqsupseteq e_{16} \vee e_{17} \vee e_{15a}$

$s_{16} = \text{INST}\ s_{map}$

$s_{17} = \text{INST}\ s_f$

$s_{18} = \text{INST}\ s_{xs}$

$s_{20} = s_{23} \xrightarrow{e_{19a}\ c_{19}} s_{19}$ $\quad e_{19} = e_{20} \vee e_{23} \vee e_{19a}$

$s_{21} = s_{22} \xrightarrow{e_{20a}\ c_{20}} s_{20}$ $\quad e_{20} = e_{21} \vee e_{22} \vee e_{20a}$

$s_{21} = \text{INST}\ s_{map}$

$s_{22} = \text{INST}\ s_{double}$

$s_{23} = \text{INST}\ s_{foo}$

The constraint tree echos the abstract syntax tree. We have retained the overall structure of the program, while dispensing with details such as distinction between case alternatives and lambda abstractions, and the order of function applications. Once constraints have been extracted, the inference algorithm can ignore the source program entirely. In our real implementation we use a source language that has more sugar than the one presented here, but the constraint language is the same.

Before discussing how to actually solve the constraints, note that there are several degrees of freedom in their ordering. Within a particular LAMBDA, LET or GROUP block it is always safe to move $=$ or $\sqsupseteq$ constraints earlier in the block. It is also safe to move these constraints up and earlier in the tree, such as moving $s_{map} = s_1$ so it appears directly after $e_0 \sqsupseteq e_1 \vee e_{19}$. Moving these constraints higher up in the tree means they will be considered earlier, and such modifications will not degrade the final constraint solution. It is also safe to change the order of LAMBDA or LET blocks at the same level. This simply corresponds to changing the order of let-bindings or case-alternatives in the original program. On the other hand, in general it is *not* safe to move INST constraints as they control the order in which types are instantiated.

Although we will discuss recursion more fully in §3.6, the structure of the constraints reveals that *map* is recursively defined. This is evident from the fact that $s_{16} = \text{INST } s_{map}$, present in the lower quarter of the list, appears inside the LET *map* block. This constraint corresponds to a recursive use of *map*. This is in contrast to $s_{21} = \text{INST } s_{map}$ which appears outside the block and corresponds to a non-recursive use. Without polymorphic recursion, all recursive uses of let-bound variables should be at identical types, so we will change $s_{16} = \text{INST } s_{map}$ to $s_{16} = s_{map}$.

From the structure of tree we see that the INST constraints for s_4, s_{12}, s_{13}, s_{17} and s_{18} all correspond to uses of lambda or pattern bound variables. This is clear because they appear inside a LAMBDA block corresponding to the variable they instantiate. For this example we will also simplify these constraints by identifying the variables on the left and right, giving: $s_4 = s_{xx}$, $s_{12} = s_f$, $s_{13} = s_x$ and so on.

We will assume that the types of *Nil* and *Cons* are known. This allows us to replace the constraints for s_6 and s_{10} with fresh instantiations of their type schemes. Although the type of *Cons* includes a closure term, we store the closure constraint in the graph instead of directly in its type. The form of our reduction rules require that all constraints involve a single constructor only.

$$
\begin{array}{ll}
s_6 & = \mathit{List}\ r_6\ a_6 \\
s_{10} & = a_{10} \xrightarrow{\perp\ \perp} s_{10a} \\
s_{10a} & = s_{10b} \xrightarrow{\perp\ c_{10}} s_{10c} \\
s_{10b} & = \mathit{List}\ r_{10}\ a_{10} \\
s_{10c} & = \mathit{List}\ r_{10}\ a_{10} \\
c_{10} & = x : a_{10}
\end{array}
$$

3.2.12 Constraint sets and equivalence classes

Now we can start building the type graph. In the graph our constraints are organised into a set of *equivalence classes*. Each equivalence class contains a set of types that must be equal, or constrained by an inequality. Equivalence classes

can be in one of three forms, depending on the kind of the types contained. Value equivalence classes have the form $\kappa n \sim \overline{s} = \overline{\tau}$, where κ is the kind of the class, n is a unique integer identifying it, $\overline{s}$ is a list of type variables, and $\overline{\tau}$ is a set of non-variable types. The intention is for the variables on the left of the $=$ to be identified with the types on the right. Effect and closure classes use $\sqsupseteq$ instead of $=$, so we write them as $\kappa n \sim \overline{a} \sqsupseteq \overline{\tau}$. As there are no constructors of region kind, we write region equivalence classes as $\kappa n \sim \overline{s}$.

For example, when we construct an equivalence class from the following constraint set:

$$\left\{ \begin{array}{lll} s_1 = s_f \stackrel{e_2\ c_1}{\longrightarrow} s_2, & s_{16} = s_{17} \stackrel{e_{15a}\ c_{15}}{\longrightarrow} s_{15}, & \\ s_{16} = s_{map}, & s_1 = s_{map} & \end{array} \right\}$$

we get:

$$*0 \quad \sim \quad s_{map}, \quad s_1, \quad s_{16} \qquad = \quad s_f \stackrel{e_2\ c_1}{\longrightarrow} s_2, \quad s_{17} \stackrel{e_{15a}\ c_{15}}{\longrightarrow} s_{15}$$

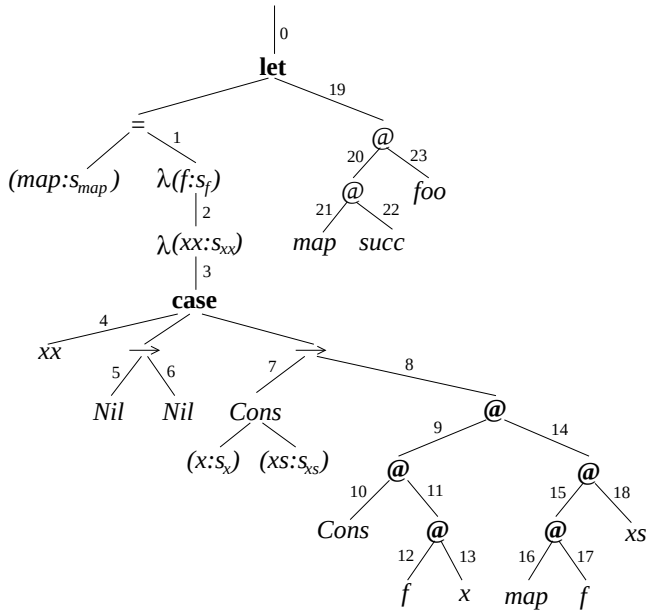
This class has the kind of value types and is identified as class 0. The variable that comes first in the list is the *canonical name* for the class. The canonical name is the variable that we choose to represent the class, and when doing inference by hand we choose the name that is most "interesting". In this case we have chosen s_{map} as more interesting than s_1 or s_{16}, but this choice will not affect the substance of our constraint solution. When a new type variable is added to an equivalence class we substitute the canonical name for occurrences of this variable in the graph. The two types on the right of the $=$ come from the $s_1 = s_f \stackrel{e_2\ c_1}{\longrightarrow} s_2$ and $s_{16} = s_{17} \stackrel{e_{15a}\ c_{15}}{\longrightarrow} s_{15}$ constraints.

We refer to the set of equivalence classes as "the type graph" to distinguish it from the set of constraints which it is built from. Constraint sets and equivalence classes express similar information, but are not completely interconvertible. An equivalence class contains variables and type constructors from the constraint set, but no information about how to match them up. For example, from the equivalence class above we cannot tell if the original constraint set contained $s_1 = s_{map}$, $s_1 = s_{16}$, or both. An equivalence class records a set of types that are all equal, but not exactly why. The fact that this information is is lost will not matter until we discuss error reporting in §3.8. Until then we will use equivalence classes, as this notation is more compact.

3.2.13 Example: type graph

Continuing on with our *map* example, we will add all constraints up to the end of the LET *map* block to the type graph. The result is shown on the next page. Note that we have elided classes that contain only a single variable, such as for r_4 and e_1.

The form of the type graph already suggests how we should proceed from here. Note that the class for s_{map} (*1) contains two type constructors $s_f \stackrel{e_{15a}\ c_{15}}{\longrightarrow} s_{15}$ and $s_f \stackrel{e_2\ c_1}{\longrightarrow} s_2$. These represent the use and definition of *map* respectively. Unifying these two types implies that the classes for s_{15} (*13) and s_2 (*5) should be merged. This induces the unification of s_{xx} and s_{xs}, which implies that the input list must have the same type as its tail, as expected.



*0	$\sim$	$s_0,$	s_{19}	$=$	$\emptyset$	
*1	$\sim$	$s_{map},$	$s_1,\ s_{16}$	$=$	$s_f \xrightarrow{e_{15a}\ c_{15}} s_{15},$	$s_f \xrightarrow{e_2\ c_1} s_2$
*2	$\sim$	$s_f,$	$s_{17},\ s_{12}$	$=$	$s_x \xrightarrow{e_{11a}\ c_{11}} s_{11}$	
*3	$\sim$	$s_x,$	$s_{13},\ a_7$	$=$	$\emptyset$	
*4	$\sim$	$s_{xs},$	s_{18}	$=$	$List\ r_7\ a_7$	
*5	$\sim$	s_2		$=$	$s_{xx} \xrightarrow{e_3\ c_2} s_3$	
*6	$\sim$	$s_3,$	$s_6,\ s_8$	$=$	$List\ r_6\ a_6$	
*7	$\sim$	$s_{xx},$	$s_4,\ s_5,\ s_7$	$=$	$List\ r_5\ a_5,$	$List\ r_7\ a_7$
*8	$\sim$	s_9		$=$	$s_{14} \xrightarrow{e_{8a}\ c_3} s_3$	
*9	$\sim$	s_{10}		$=$	$s_{11} \xrightarrow{e_{9a}\ c_9} s_9,$	$a_{10} \xrightarrow{\perp\ \perp} s_{10a}$
*10	$\sim$	s_{10a}		$=$	$s_{10b} \xrightarrow{\perp\ c_{10}} s_{10c}$	
*11	$\sim$	s_{10b}		$=$	$List\ r_{10}\ a_{10}$	
*12	$\sim$	s_{10c}		$=$	$List\ r_{10}\ a_{10}$	
*13	$\sim$	s_{15}		$=$	$s_{xs} \xrightarrow{e_{14a}\ c_{14}} s_{14}$	
!0	$\sim$	e_0		$\sqsupseteq$	$e_1 \vee e_{19}$	
!1	$\sim$	e_3		$\sqsupseteq$	$ReadH\ \ s_{xx} \vee e_6 \vee e_8$	
!2	$\sim$	e_8		$\sqsupseteq$	$e_9 \vee e_{14} \vee e_{8a}$	
!3	$\sim$	e_9		$\sqsupseteq$	$e_{10} \vee e_{11} \vee e_{9a}$	
!4	$\sim$	e_{11}		$\sqsupseteq$	$e_{12} \vee e_{13} \vee e_{11a}$	
!5	$\sim$	e_{14}		$\sqsupseteq$	$e_{15} \vee e_{18} \vee e_{14a}$	
!6	$\sim$	e_{15}		$\sqsupseteq$	$e_{16} \vee e_{17} \vee e_{15a}$	
$1	$\sim$	c_1		$\sqsupseteq$	$map : s_{map}$	
$2	$\sim$	c_2		$\sqsupseteq$	$map : s_{map} \vee f : s_f$	
$3	$\sim$	c_{10}		$\sqsupseteq$	$x : a_{10}$	

3.3 Constraint reduction

Our immediate goal is to determine a type for *map*. However, the equivalence class corresponding to s_{map} currently contains two different type expressions. Although union typing systems [DP03] provide a join operator on value types, we do not consider these systems here and will instead take the standard approach of unifying all the types in a particular equivalence class. Unifying all types corresponds to a ML style system, which is usually expressive enough for our needs. We will consider union typing again in §5.2.2.

The unification of types may generate more constraints. These new constraints must be added back to the graph, possibly resulting in more unifications, which may generate more constraints, and so on. In DDC we call this process *grinding* the graph, and we take this to include performing unifications as well as reducing type class constraints and compound effects.

3.3.1 Constraint entailment

We use entailment rules to describe operations on constraint sets. Entailment rules have the form $P \Vdash Q$, where P and Q are both sets. $P \Vdash Q$ can be read: "P entails Q", or perhaps "P produces Q". If we have a constraint set R and a rule $P \Vdash Q$ where $P \subseteq R$, then we can replace the constraints P in R by the new constraints Q. Any variables present in P match the corresponding types in R. For example, we could apply transitivity rule:

$$\begin{array}{rl} \text{(trans)} & \{s_1 = s_2,\ s_2 = s_3\} \\ \Vdash & \{s_1 = s_2,\ s_2 = s_3,\ s_1 = s_3\} \end{array}$$

To the following constraint set:

$$\{s_a = s_b,\ s_b = Int\ \ r_1\}$$

to get:

$$\{s_a = s_b,\ s_b = Int\ \ r_1,\ s_a = Int\ \ r_1\}$$

When we apply an entailment rule $P \Vdash Q$, we take any variables present in Q but not P or R to be fresh.

Note that our entailment rules are expressed as operations on constraint sets, not on type graphs. To apply a rule to the type graph we must imagine it being converted to a constraint set and back again. As discussed in §3.2.12 the two forms are not totally equivalent, but the fact that we lose information when converting a constraint set to a type graph will only matter when we come come to discuss type error messages in §3.8. We use the graph representation until then as it is more compact and simplifies the presentation.

3.3.2 Unification

The entailment rules for unification are:

$$\begin{array}{ll} \text{(unify fun)} & \{\ s = a_1 \xrightarrow{e_1\ c_1} b_1,\ s = a_2 \xrightarrow{e_2\ c_2} b_2\ \} \\ \vdash & \{\ s = a_1 \xrightarrow{e_1\ c_1} b_1,\ a_1 = a_2,\ b_1 = b_2,\ e_1 = e_2,\ c_1 = c_2\ \} \end{array}$$

$$\begin{array}{ll} \text{(unify data)} & \{\ s = T_1\ \overline{a},\ s = T_1\ \overline{b}\ \} \\ \vdash & \{\ s = T_1\ \overline{a},\ \overline{a = b}\ \} \end{array}$$

The first rule is applicable when there are two function type constraints on a particular variable s. Applying the rule causes the second constraint to be discarded, while generating four new ones. These new constraints equate the type variables for the argument, return value, effect and closure of each function. The second rule is similar.

When we come to add a constraint like $a_1 = a_2$ to our type graph, if the two variables are already in the same equivalence class then we just ignore the constraint. If they are in separate classes then we add all the variables and types in the first one to the second, and delete the first (or *vice-versa*). For example, our graph for *map* includes the following:

$$\begin{array}{llllll} *1 & \sim & s_{map},\ s_1,\ s_{16} & = & s_f \xrightarrow{e_{15a}\ c_{15}} s_{15}, & s_f \xrightarrow{e_2\ c_1} s_2 \\ *2 & \sim & s_f,\ s_{17},\ s_{12} & = & s_x \xrightarrow{e_{11a}\ c_{11}} s_{11} & \\ *5 & \sim & s_2 & = & s_{xx} \xrightarrow{e_3\ c_2} s_3 & \\ *13 & \sim & s_{15} & = & s_{xs} \xrightarrow{e_{14a}\ c_{14}} s_{14} & \\ !7 & \sim & e_{15a} & \sqsupseteq & \bot & \\ !8 & \sim & e_2 & \sqsupseteq & \bot & \\ \$1 & \sim & c_1 & \sqsupseteq & map : s_{map} & \\ \$4 & \sim & c_{15} & \sqsupseteq & \bot & \end{array}$$

Applying the unification rule to *1 allows us to delete the first function type, while generating the new constraints $s_f = s_f$, $s_{15} = s_2$, $e_{15a} = e_2$ and $c_{15} = c_1$. We can safely discard the trivial identity $s_f = s_f$.

To add $s_{15} = s_2$ back to the graph we add the elements of *5 to *13 and discard *5. Likewise, to add $e_{15a} = e_2$ we will add the elements of !7 to !8 and delete !7. This yields:

$$\begin{array}{llllll} *1 & \sim & s_{map},\ s_1,\ s_{16} & = & s_f \xrightarrow{e_2\ c_1} s_2 & \\ *2 & \sim & s_f,\ s_{17},\ s_{12} & = & s_x \xrightarrow{e_{11a}\ c_{11}} s_{11} & \\ *13 & \sim & s_{15},\ s_2 & = & s_{xs} \xrightarrow{e_{14a}\ c_{14}} s_{14},\ s_{xx} \xrightarrow{e_3\ c_2} s_3 & \\ !8 & \sim & e_2,\ e_{15a} & \sqsupseteq & \bot \vee \bot & \\ \$1 & \sim & c_1,\ c_{15} & \sqsupseteq & map : s_{map} \vee \bot & \end{array}$$

Note that when there are multiple types in a value type equivalence class we separate them by a comma. On the other hand, multiple types in an effect or closure equivalence class are separated by $\vee$. This follows naturally from the fact that constraints on value types are always expressed with $=$, but constraints

on effects and closures are always expressed with $\sqsupseteq$. Using the definition of $\vee$ we can then go on to simplify $\bot \vee \bot$ to just $\bot$. Note that in the constraint set representation this simplification isn't needed. If we have both $e_1 \sqsupseteq \bot$ and $e_1 \sqsupseteq \bot$, then putting these constraints in a set automatically 'merges' them.

The application of (unify fun) to *1 has caused a new type to be added to *1. We keep applying our unification rules until no further progress can be made, and when this is done we have:

*0	$\sim$	s_0,	s_{19}	$=$	$\bot$
*1	$\sim$	s_{map},	s_1, s_{16}	$=$	$s_f \xrightarrow{e_2\ c_1} s_{15}$
*2	$\sim$	s_f,	s_{17}, s_{12}	$=$	$s_x \xrightarrow{e_{11a}\ c_{11}} s_{11}$
*3	$\sim$	s_x,	s_{13}, a_7, a_5	$=$	$\bot$
*6	$\sim$	s_3,	s_6, $s_8, s_{14}, s_{10b}, s_{10c}$	$=$	$List\ r_6\ s_{11}$
*7	$\sim$	s_{xx},	s_4, s_5, s_7, s_{xs}, s_{18}	$=$	$List\ r_5\ s_x$
*9	$\sim$	s_{10}		$=$	$s_{11} \xrightarrow{e_{9a}\ c_9} s_{10a}$
*10	$\sim$	s_{10a}	s_9	$=$	$s_3 \xrightarrow{e_{8a}\ c_3} s_3$
*13	$\sim$	s_{15},	s_2	$=$	$s_{xx} \xrightarrow{e_3\ c_2} s_3$
*14	$\sim$	s_{11},	a_{10}, a_6	$\sqsupseteq$	$\bot$
%0	$\sim$	r_5,	r_7		
%1	$\sim$	r_6,	r_{10}		
!0	$\sim$	e_0		$\sqsupseteq$	$e_1 \vee e_{19}$
!1	$\sim$	e_3,	e_{14a}	$\sqsupseteq$	$ReadH\ s_{xx} \vee e_6 \vee e_8$
!2	$\sim$	e_8		$\sqsupseteq$	$e_9 \vee e_{14} \vee e_{8a}$
!3	$\sim$	e_9		$\sqsupseteq$	$e_{10} \vee e_{11} \vee e_{9a}$
!4	$\sim$	e_{11}		$\sqsupseteq$	$e_{12} \vee e_{13} \vee e_{11a}$
!5	$\sim$	e_{14}		$\sqsupseteq$	$e_{15} \vee e_{18} \vee e_3$
!6	$\sim$	e_{15}		$\sqsupseteq$	$e_{16} \vee e_{17} \vee e_2$
!7	$\sim$	e_2,	e_{15a}	$\sqsupseteq$	$\bot$
$1	$\sim$	c_1		$\sqsupseteq$	$map : s_{map}$
$2	$\sim$	c_2		$\sqsupseteq$	$map : s_{map} \vee f : s_f$
$3	$\sim$	c_{10}		$\sqsupseteq$	$x : a_{10}$

3.3.3 Head read

When unification is complete we have a constraint $s_{xx} = List\ r_5\ s_x$ in *7. This allows us to reduce the $ReadH\ s_{xx}$ effect in !1 that was generated by the case-expression in the original program. In DDC we call the process of reducing effect types or type class constraints *crushing*.

The entailment rule for $ReadH$ is :

$$\begin{array}{ll} \text{(read head)} & \{\ e \sqsupseteq ReadH\ \ s,\ \ s = T\ r\ \overline{s}\ \} \\ \vdash & \{\ e \sqsupseteq Read\ \ r,\ \ \ \ s = T\ r\ \overline{s}\ \} \end{array}$$

This says that the effect of reading the primary region of a data type can be reduced to a simple read of that region once the region is known. This lets us reduce:

$$\begin{array}{rcll} *7 & \sim & s_{xx}, \quad s_4,\ s_5,\ s_7,\ s_{xs},\ s_{18} & = \ \mathit{List}\ r_5\ s_x \\ !1 & \sim & e_3, \quad e_{14a} & \sqsupseteq \ \mathit{ReadH}\ s_{xx} \vee e_6 \vee e_8 \end{array}$$

to:

$$\begin{array}{rcll} *7 & \sim & s_{xx}, \quad s_4,\ s_5,\ s_7,\ s_{xs},\ s_{18} & = \ \mathit{List}\ r_5\ s_x \\ !1 & \sim & e_3, \quad e_{14a} & \sqsupseteq \ \mathit{Read}\ r_5 \vee e_6 \vee e_8 \end{array}$$

Recall that when applying an entailment rule, we must imagine the type graph being converted to a constraint set and back. The two equivalence classes *7 and !1 correspond to the set:

$$\begin{array}{lllll} \{ & s_{xx} = s_4, & s_{xx} = s_5, & s_{xx} = s_7, & s_{xx} = s_{xs}, \\ & s_{xx} = s_{18}, & s_{xx} = \mathit{List}\ r_5\ s_x, & & \\ & e_3 = e_{14a}, & e_3 \sqsupseteq \mathit{ReadH}\ s_{xx}, & e_3 \sqsupseteq e_6, & e_3 \sqsupseteq e_8 \quad \} \end{array}$$

Expressing the graph in this form separates out the constraints $e_3 \sqsupseteq \mathit{ReadH}\ s_{xx}$ and $s_{xx} = \mathit{List}\ r_5\ s_x$, which match the premise of the (read head) rule.

3.4 Generalisation

When no more types can be unified, and no more effects or type class constraints can be reduced, the graph is said to be in *normal form*. We can now extract the type for *map* from our graph and generalise it into a type scheme.

In DDC we refer to the complete process of building a type scheme from the information in the type graph as "generalisation". This process is broken down into several stages, summarised below. Note that in a concrete implementation, several of these stages can performed at the same time. For example, checking for loops through the constraints can be done while tracing them, as tracing corresponds to a simple reachability analysis.

1.	Tracing:	isolate the type information of interest.
2.	Loop checking:	check for infinite value type errors.
3.	Packing:	pack the graphical constraints into "flat" form.
4.	Loop breaking:	break loops through effect and closure constraints.
5.	Cleaning:	discard non-interesting effect and closure variables.
6.	Quantification:	introduce type quantifiers.

3.4.1 Tracing

The first step is to trace out the section of graph that is reachable from the type variable we're interested in. For this example we're interested in *map* and all equivalence classes except *0 are reachable from s_{map}. This process makes a copy of the information present in the "global" graph, and the operations described in the rest of this section are performed on the copy.

3.4.2 Loop checking

We now check for loops through the value type portion of the copied sub-graph. A classic example of a program with a looping type is:

$$x \quad = \mathit{Cons}\ \ x\ \mathit{Nil}$$

where *Cons* and *Nil* have the following types:

$$\begin{array}{ll} \mathit{Nil} & :: \forall r\ a.\ \mathit{List}\ \ r\ a \\ \mathit{Cons} & :: \forall r\ a\ c.\ a \rightarrow \mathit{List}\ \ r\ a \xrightarrow{c} \mathit{List}\ \ r\ a \\ & \quad \rhd c \sqsupseteq x : a \end{array}$$

After slurping and grinding constraints, this program has the following graph:

$$\begin{array}{llll} *1 \sim s_x, & s_1,\ s_4,\ a_3,\ a_5 & = & \mathit{List}\ \ r_3\ s_x \\ *2 \sim s_2, & & = & s_5 \xrightarrow{e_1\ c_1} s_x \\ *3 \sim s_3, & s_{\mathit{Cons}} & = & s_4 \xrightarrow{e_2\ c_2} s_2 \\ *4 \sim s_5, & & = & \mathit{List}\ \ r_3\ s_x \\ \$1 \sim c_3, & c_1 & \sqsupseteq & x : s_x \\ \%1 \sim r_3, & r_5 & & \end{array}$$

The loop is through equivalence class *1. We cannot produce a flat, non-graphical type for s_x because if we tried to solve its constraint our algorithm would diverge:

$$
\begin{array}{lll}
 & s_x & = List\ r_3\ s_x \\
\equiv & s_x & = List\ r_3\ (List\ r_3\ s_x) \\
\equiv & s_x & = List\ r_3\ (List\ r_3\ (List\ r_3\ s_x)) \\
\equiv & \ldots &
\end{array}
$$

Unlike [CC91] we do not attempt to handle recursive value types, so we flag them as errors instead. This is common to other compilers such as GHC, which would emit a message "cannot construct infinite type". Note that loops through the effect or closure portions of a type graph are *not* counted as errors. We deal with these in §3.4.4.

3.4.3 Packing

Packing is the process of converting a set of individual type constraints into the normalised form of 3.2.1. When we pack the constraints from our *map* example into this form we have:

$$
\begin{array}{lllll}
s_{map} & = (s_x \xrightarrow{e_{11a}\ c_{11}} s_{11}) \xrightarrow{e_2\ c_1} List\ r_5\ s_x \xrightarrow{e_3\ c_2} List\ r_6\ s_{11} & & & \\
\rhd & e_3 & \sqsupseteq & Read\, r_5 \vee e_6 \vee e_8 \\
, & e_8 & \sqsupseteq & e_9 \vee e_{14} \vee e_{8a} \\
, & e_9 & \sqsupseteq & e_{10} \vee e_{11} \vee e_{9a} \\
, & e_{11} & \sqsupseteq & e_{12} \vee e_{13} \vee e_{11a} \\
, & e_{14} & \sqsupseteq & e_{15} \vee e_{18} \vee e_3 \\
, & e_{15} & \sqsupseteq & e_{16} \vee e_{17} \vee e_2 \\
, & c_1 & \sqsupseteq & map : (s_x \xrightarrow{e_{11a}\ c_{11}} s_{11}) \xrightarrow{e_2\ c_1} List\ r_5\ s_x \xrightarrow{e_3\ c_2} List\ r_6\ s_{11} \\
, & c_2 & \sqsupseteq & map : (s_x \xrightarrow{e_{11a}\ c_{11}} s_{11}) \xrightarrow{e_2\ c_1} List\ r_5\ s_x \xrightarrow{e_3\ c_2} List\ r_6\ s_{11} \\
 & & & \quad \vee\ f : s_x \xrightarrow{e_{11a}\ c_{11}} s_{11}
\end{array}
$$

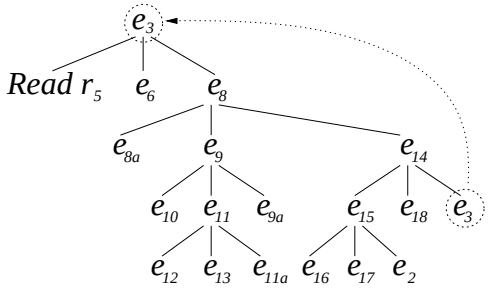
Although the body of this type is now in the familiar form, there is still a hash of effect and closure constraints. However, notice that there is only one use each of the variables e_8, e_9, e_{11}, e_{14} and e_{15}, and they are not mentioned in the body of this type. From a compiler optimisation point of view, the only effect information that we need to preserve is the manifest effect of each function arrow. The fact that, say, e_3 includes e_8 and e_8 includes e_9 does not matter, as long as all of the appropriate variables are reachable from e_3.

This means that we can inline the constraints on e_8, e_9 and so on into the constraint for e_3. We can also use the closure trimming process discussed in §2.5.7 to simplify the constraints for c_1 and c_2. This produces:

$$
\begin{array}{lllll}
s_{map} & = (s_x \xrightarrow{e_{11a}\ c_{11}} s_{11}) \xrightarrow{e_2\ c_1} List\ r_5\ s_x \xrightarrow{e_3\ c_2} List\ r_6\ s_{11} & & & \\
\rhd & e_3 & \sqsupseteq & Read\, r_5 \vee e_6 \vee e_8 \vee e_9 \vee e_{14} \vee e_{8a} \\
 & & & \quad \vee\ e_{10} \vee e_{11} \vee e_{9a} \vee e_{12} \vee e_{13} \vee e_{11a} \\
 & & & \quad \vee\ e_{15} \vee e_{18} \vee e_3 \vee e_{16} \vee e_{17} \vee e_2 \\
, & c_1 & \sqsupseteq & map : c_1 \\
, & c_2 & \sqsupseteq & map : c_1 \vee\ f : c_{11}
\end{array}
$$

3.4.4 Loop breaking

As discussed in §2.3.8, we can use the lattice structure of effects and closures to break loops through effect and closure constraints. If the type of a particular value variable contains looping constraints, then breaking these loops makes the type simpler, and allows it to be exported to the core language. The following diagram shows a loop through the effect portion of the type for *map*, as it was after the first stage of packing:



Notice how the structure of this effect graph echos the original abstract syntax tree for *map*. In the effect graph we can see two branches, headed by e_9 and e_{14}, corresponding to the two alternatives of the original case expression. We can also see the recursive call via e_3. *map* applies itself, so the effect of *map* includes the effect of *map*. Similarly, *map* references itself, so the closure of *map* includes the closure of *map*.

However, although recursion in the function's *definition* serves a useful purpose, the fact that its effect and closure is also recursive is not exploited by our analysis. We need to retain the set of effects caused by an application of *map*, that is, the effects reachable from e_3, but this information would not be lost if we substituted $\bot$ for its recursive occurrence. Finally, the constraints $e_3 \sqsupseteq e_3$ and $c_1 \sqsupseteq \mathit{map} : c_1$ are trivially satisfied, so can be discarded. This leaves us with:

$$
\begin{aligned}
s_{map} \quad &= (s_x \xrightarrow{e_{11a}\ c_{11}} s_{11}) \xrightarrow{e_2\ c_1} \mathit{List}\ r_5\ s_x \xrightarrow{e_3\ c_2} \mathit{List}\ r_6\ s_{11} \\
&\rhd \quad e_3 \quad \sqsupseteq \quad \mathit{Read}\, r_5 \vee e_6 \vee e_8 \vee e_9 \vee e_{14} \vee e_{8a} \\
&\qquad\qquad\qquad\qquad \vee e_{10} \vee e_{11} \vee e_{9a} \vee e_{12} \vee e_{13} \vee e_{11a} \\
&\qquad\qquad\qquad\qquad \vee e_{15} \vee e_{18} \vee \bot \vee e_{16} \vee e_{17} \vee e_2 \\
&\quad , \quad c_2 \quad \sqsupseteq \quad \mathit{map} : c_1 \vee \ f : c_{11}
\end{aligned}
$$

3.4.5 Cleaning

There are still a large number of effect and closure variables in our type that don't provide any useful information. For example, e_6 is the effect of evaluating the variable *Nil*, but evaluating a variable doesn't have a visible effect. Also consider e_9, the effect of evaluating *Cons* $(f\ x)$. The evaluation of $(f\ x)$ itself has the effect e_{11}, which is interesting because it depends on the function argument passed to map, but the partial application of *Cons* to the result does not have an effect, so is boring. In any event, the constraint on e_3 already contains e_{11}, so it doesn't need to include e_9 as well.

We define boring effect and closure variables to be the ones that are completely unconstrained. Such variables are not mentioned in the type environment, in a

parameter of the type being generalised, or on the left of an (in)equality. If a variable is in the type environment then it depends on a definition in a higher scope of the program. In this case, a more interesting type may be unified into the variable later in the inference process. If it is mentioned in a parameter type then it may be different for each application of the function. If it is on the left of an (in)equality then it depends on other types. For our example, e_{11a}, c_{11}, e_3 and c_2 are interesting, and the rest are boring. We substitute $\bot$ for boring variables, then use the definition of $\vee$:

$$\begin{array}{lll} s_{map} & = (s_x \xrightarrow{e_{11a}\ c_{11}} s_{11}) \longrightarrow \mathit{List}\ r_5\ s_x \xrightarrow{e_3\ c_2} \mathit{List}\ r_6\ s_{11} \\ & \rhd \quad e_3 & \sqsupseteq\ \mathit{Read}\, r_5 \vee e_{11a} \\ & , \quad c_2 & \sqsupseteq\ f : c_{11} \end{array}$$

3.4.6 Quantification

We can now add quantifiers to our type and create a type scheme. There are several restrictions as to what variables can be quantified, which we will recall in a moment, but for this example none apply. After quantification we have:

$$\begin{array}{lll} s_{map} & = \forall s_x\ s_{11}\ r_5\ r_6\ e_{11a}\ c_{11}\ e_3\ c_2 \\ & .\quad (s_x \xrightarrow{e_{11a}\ c_{11}} s_{11}) \longrightarrow \mathit{List}\ r_5\ s_x \xrightarrow{e_3\ c_2} \mathit{List}\ r_6\ s_{11} \\ & \rhd \quad e_3 & \sqsupseteq\ \mathit{Read}\ r_5 \vee e_{11a} \\ & , \quad c_2 & \sqsupseteq\ f : c_{11} \end{array}$$

We will also rewrite the quantified variables to use more familiar names:

$$\begin{array}{lll} s_{map} & = \forall a\ b\ r_1\ r_2\ e_1\ e_2\ c_1\ c_2 \\ & .\quad (a \xrightarrow{e_1\ c_1} b) \longrightarrow \mathit{List}\ r_1\ a \xrightarrow{e_2\ c_2} \mathit{List}\ r_2\ b \\ & \rhd \quad e_2 & \sqsupseteq\ \mathit{Read}\ r_1 \vee e_1 \\ & , \quad c_2 & \sqsupseteq\ f : c_1 \end{array}$$

At this stage we could also apply the the effect masking rules from §2.3.7 and §2.5.2, though none apply in this example. Once generalisation is complete we update the s_{map} equivalence class in the global graph so it contains this new type scheme.

The non-generalisable variables

There are several reasons why a particular type variable may not be permitted to be generalised. All but the first were discussed in the previous chapter.

Don't generalise:

1. Variables free in the type environment. This is the standard restriction for Hindley-Milner type systems [Mil78, DM82]. However, as we're performing type inference instead of type *checking*, the real type environment is not close at hand. We instead use the method discussed in §3.6 to determine the value variables that are free in the binding whose type is being generalised. The types of these variables can be determined from the graph, and we hold their free type variables monomorphic. This achieves the same result.

2. Dangerous type variables, which were discussed in §2.5.1 and §3.2.3. These are variables that appear free under type constructors whose regions are constrained to be mutable. Dangerous type variables must be held monomorphic to avoid the problem with polymorphic update that was discussed in §2.1.

3. Material region variables, which were discussed in §2.5.4. Material regions correspond with objects that are shared between all uses of the variable whose type is being generalised. These must also be held monomorphic.

3.4.7 Late constraints and post-inference checking

The fact that mutability constraints influence what type variables are quantified introduces a slight complication into the inference process. The type inferencer might quantify a type variable while assuming that a particular region is constant, but later discover a mutability constraint that indicates it shouldn't have quantified. We refer to such mutability constraints as *late* constraints. The following example demonstrates the problem:

$$
\begin{array}{l}
\mathit{lateFun}\ () \\
= \mathbf{do}\quad \mathit{ref}\ \ = \mathit{newRef}\ \ \mathit{id} \\
\qquad\quad\ \ f\ \ \ = \lambda().\, \mathit{readRef}\ \ \mathit{ref} \\
\qquad\quad\ \ \mathit{writeRef}\ \ \mathit{ref}\ \ \mathit{succ} \\
\qquad\quad\ \ f\ ()\ \ \text{“}\texttt{oh noes}\text{”}
\end{array}
$$

with

$$
\begin{array}{lll}
\mathit{id} & :: & \forall a.\ a \to a \\
\mathit{succ} & :: & \forall r_1\ r_2\ e_1.\ \mathit{Int}\ \ r_1 \xrightarrow{e_1} \mathit{Int}\ \ r_2 \\
 & \rhd & e_1 \sqsupseteq \mathit{Read}\ \ r_1 \\
\mathit{newRef} & :: & \forall a\ r_1.\ a \to \mathit{Ref}\ \ r_1\ a \\
\mathit{readRef} & :: & \forall a\ r_1\ e_1.\, \mathit{Ref}\ \ r_1\ a \xrightarrow{e_1} a \\
 & \rhd & e_1 \sqsupseteq \mathit{Read}\ \ r_1 \\
\mathit{writeRef} & :: & \forall a\ r_1\ e_1\ c_1.\, \mathit{Ref}\ \ r_1\ a \to a \xrightarrow{e_1\ \ c_1} () \\
 & \rhd & e_1 \sqsupseteq \mathit{Write}\ \ r_1 \\
 & , & c_1 \sqsupseteq x : \mathit{Ref}\ \ r_1\ a \\
 & , & \mathit{Mutable}\ \ r_1
\end{array}
$$

From the types of *newRef*, *readRef* and *writeRef*, we see that an object of type $\mathit{Ref}\ \ r_1\ a$ is only treated as mutable if we actually apply the *writeRef* function to it. If we only ever *read* a reference, then there is no need to mark it as mutable or restrict the type of the contained value. However, with our *lateFun* example, the type inferencer will only discover that *ref* is mutable when it processes the third line of the do-expression. If it considers the bindings in-order, and generalises the type of *ref* while assuming constancy, it will get:

$$\mathit{ref} :: \forall a.\, \mathit{Ref}\ \ r_1\ (a \to a)$$

with this scheme, the type of f becomes:

$$\begin{array}{lll} f & :: & \forall b\ e_1\ c_1.\ () \xrightarrow{e_1\ c_1} b \to b \\ & \rhd & e_1 \sqsupseteq \mathit{Read}\ \ r_1 \\ & , & c_1 \sqsupseteq \mathit{ref} : \mathit{Ref}\ \ r_1\ (b \to b) \end{array}$$

Later, the inferencer will discover the call to *writeRef*, which places a mutability constraint on r_1 and invalidates the previous type scheme for *ref*. Note that it is not safe to simply change the scheme for *ref* to a less general one "on the fly", because the old scheme was instantiated when inferring the type of f, and now that type is wrong as well.

A simple solution is to wait until type inference is complete, re-generalise the types of all let-bound variables, and compare the resulting type schemes against the previously inferred ones. Waiting until inference is complete ensures that all the available mutability constraints have made their way into the type graph. If a newly generalised scheme is different to its original version, then a mutability constraint must have been added to the graph after the original was created. This is reported as an error.

This solution has the disadvantage of requiring the types of polymorphic mutable objects to be given explicitly as signatures. For example, although the following program will not cause a problem at runtime, it will be marked as erroneous:

```
falseLate
 = do       ref    = newRef  id
            writeRef  ref  succ
            (readRef ref) 5
```

Adding a type signature ensures that the inferencer will treat *ref* as being mutable when its type is generalised:

```
fixedLate'
 = do       ref :: Ref  r1 (a → a) ▷ Mutable  r1
            ref    = newRef  id
            writeRef  ref  succ
            (readRef ref) 5
```

An alternate solution would be to re-generalise the type of *ref* at each instantiation point. If the newly generalised scheme was different to the previous one, then we could backtrack to original definition and re-type the subsequent program using the new scheme. We could also perform backtracking in a coarse grained manner, by doing the post-inference check as before, but re-typing the *whole* program if any schemes were different. However, backtracking adds extra complexity, and we expect programs like the above to be rare, so we use the simpler non-backtracking solution in our implementation.

The BitC compiler also checks for the late constraint problem [SSS08]. It does not backtrack, but performs the check during type inference proper. It also uses local heuristics to guess whether a particular variable is mutable, without performing complete inference. Before generalising the type of a variable it inspects how it is used later in the function. If the variable is updated locally, its type is generalised using this information. This is possible in BitC because the assignment operator := is baked into the language, instead of being defined in the standard libraries.

3.5 Projections

This section gives the formal definition of type directed projections, which were introduced in §2.7. Syntactically, projections are a clean extension of the language described in §3.2. Type inference for projections is a mostly-orthogonal extension to the system discussed thus far, though the handling of mutually recursive definitions requires careful ordering of the constraints considered by the inferencer. This section introduces type inference for projections, though we defer further discussion of mutual recursion and constraint ordering to §3.6.

Declarations

$$\begin{array}{lll} \mathit{decl} \rightarrow & \ldots & \\ \quad | & \textbf{project } T \textbf{ with } \overline{l \sim x} & \text{(projection declaration)} \end{array}$$

Terms

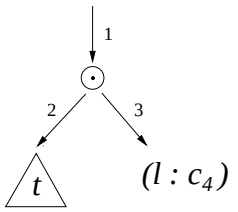
$$\begin{array}{lll} t \rightarrow & \ldots & \\ \quad | & t_1 \odot l & \text{(projection)} \\ \quad | & t_1 \odot (l : c) & \text{(annotated projection)} \end{array}$$

Decl-Proj

$$\Gamma \vdash_{\text{decl}} \textbf{project } T \textbf{ with } \overline{l \sim x} \;::\; \overline{l \overset{T}{\sim} x}$$

Proj

$$\frac{\Gamma \vdash t :: T\ r\ \overline{\tau} \rhd \Omega\ ;\ e_2 \quad \Gamma \vdash x :: T\ r\ \overline{\tau} \xrightarrow{e_3\ c_4} \tau' \rhd \Omega\ ;\ \bot \quad l \overset{T}{\sim} x \in \Gamma}{\Gamma \vdash t \odot l :: \tau' \rhd \Omega\ ;\ e_2 \vee e_3}$$



$$\begin{array}{ll} s_3 & = \text{PROJ } l\ s_2 \\ s_3 & = s_2 \xrightarrow{e_3\ c_4} s_1 \\ e_1 & \sqsupseteq e_2 \vee e_3 \end{array}$$

$$\begin{array}{lll} \text{(projection)} & \{\ s_1 = \text{PROJ } l\ s_2, & s_2 = T\ \overline{s}\ \} \\ \Vdash & \{\ s_1 = \text{INST } s_v, & s_2 = T\ \overline{s}\ \} \\ & \text{where} \quad l \overset{T}{\sim} v & \end{array}$$

A projection dictionary **project** T **with** $\overline{l \sim x}$ is associated with a particular type constructor T. The dictionary lists the names of the instance functions $\overline{x}$ that should be used to implement each of the projections labeled $\overline{l}$. Recall that the projection operator binds more tightly than function application, so $t_1\ t_2 \odot \mathit{field}$ should be read as $t_1\ (t_2 \odot \mathit{field})$ The annotated projection $t_1 \odot (l : c)$ contains a closure variable c, and we will discuss how this is used in a moment.

The (Decl-Proj) rule introduces the bindings $\overline{l \overset{T}{\sim} x}$ from the projection dictionary into the type environment.

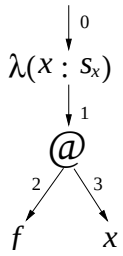
The (Proj) rule says that to assign a type to the projection $t \odot l$, the expression t must have a data type that includes an outer constructor T. There must be a binding $l \overset{T}{\sim} x$ in the type environment showing which instance function x to use to implement the projection. This instance function must take a value of the object type $T\ r\ \overline{\varphi}$ and return a value of the result type τ'. We name the effect and closure of the instance function e_3 and c_4 respectively. Evaluation of the expression $t\ \odot\ l$ causes the instance function to be applied, so we have e_3 in the rule's conclusion.

In the list of constraints, $s_3 = \text{PROJ}\ l\ s_2$ says that we need to wait until the type of s_2 is known before we look up the instance function for l from the corresponding projection dictionary. When this is done we can bind the type of the instance function to s_3. The constraint $s_3 = s_2 \xrightarrow{e_3\ c_4} s_1$ requires the instance function to have an appropriate type. The last constraint gives the effect of the whole node.

We now discuss the meaning of the closure variable on an annotated projection. Firstly, recall that when we generate constraints for λ abstractions, we need to know what free variables lie in their bodies. For example, if we have an annotated expression:

$$\lambda(x : s_x).\ f\ x$$

With labeled syntax tree:



This would lead to the following constraints:

$$\begin{array}{l}\text{LAMBDA}\ \{x\} \\ \quad \begin{array}{ll} s_0 & = s_x \xrightarrow{e_1\ c_0} s_1 \\ c_0 & \sqsupseteq f : s_f \\ s_2 & = s_f \\ s_3 & = s_x \end{array}\end{array}$$

The closure term $f : s_f$ is present because f is free in the body of the abstraction. At runtime f might be represented as a thunk which contains pointers to shared objects, and we use the closure term to account for this.

Now, suppose that the body of the abstraction contained a projection instead:

$$\lambda(x : s_x).\ x\ \odot\ l$$

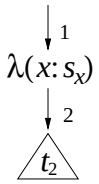
Until we have inferred the type of x we won't know what instance function will be called, or what the closure of the body should be. If, during constraint reduction, we discover that the projection $x \odot l$ resolves to $g\ x$, then this tells us that it is g that will be called at runtime. However, when generating constraints we can't use $c_0 \sqsupseteq g : s_g$ as the closure constraint for the surrounding lambda

abstraction, because we won't know about g until the appropriate constraints are solved. Instead, we annotate the projection label with a fresh variable c_1, and use this as a place holder until the type of x becomes known. Once the type of x is known we can determine what instance function will be called, and then bind its closure to c_1.

$$\lambda(x : s_x).\ x\ \odot\ (l : c_1)$$

The final task is to modify the constraint generation rules for lambda abstraction to take account of these new closure variables:

Abs-Proj



$$\begin{array}{l}\text{LAMBDA}\ \ \{x\}\\ \quad s_1 \quad = s_x \xrightarrow{e_2\ c_1} s_2\\ \quad c_1 \quad \sqsupseteq\ y_0 : s_{y0}\ \vee\ y_1 : s_{y1}\ \vee \ldots\\ \qquad\quad \vee\ l_0\ : c_0\ \ \vee\ l_1\ : c_1\ \ \vee \ldots\\ \qquad\qquad \text{where}\ \underline{y_n} \qquad \leftarrow fv(t_2) \setminus x\\ \qquad\qquad\qquad\ \ \overline{l_n : c_n}\ \ \leftarrow closAnnot(t_2)\end{array}$$

$$\mathbf{SLURP}(t_2)$$

The function $closAnnot(t_2)$ simply collects the $(l : c)$ pairs from all expressions of the form $t \odot (l : c)$ present in term t_2.

3.5.1 Example: vectors

The following program defines data types for vectors of two and three dimensions, along with projections that calculate their magnitudes. We have taken the liberty of using floating point numbers and infix operators without formally introducing them first, and have elided some insignificant details.

$$\begin{array}{l}\textbf{data}\ \ \mathit{Vec2} :: \% \to * \ \textbf{where}\\ \quad \mathit{MkVec2}\ \ :: \forall r_1\ c_1.\ \mathit{Float}\ \ r_1 \to \mathit{Float}\ \ r_1 \xrightarrow{c_1} \mathit{Vec2}\ \ r_1\\ \qquad\qquad \rhd\ c_1 \sqsupseteq x : r_1\\ \textbf{data}\ \ \mathit{Vec3} :: \% \to * \ \textbf{where}\\ \quad \mathit{MkVec3}\ \ :: \ldots\\ \textbf{project}\ \ \mathit{Vec2}\ \ \textbf{with}\ \mathit{magnitude} \sim \mathit{vec2\_magnitude}\\ \textbf{project}\ \ \mathit{Vec3}\ \ \textbf{with}\ \mathit{magnitude} \sim \mathit{vec3\_magnitude}\\ \textbf{let}\ \ \mathit{vec2\_magnitude} = \lambda v.\ \textbf{case}\ v\ \textbf{of}\ \mathit{MkVec2}\ \ x\ y \to \mathit{sqrt}\ (x * x + y * y)\\ \quad \mathit{vec3\_magnitude} = \ldots\\ \quad \mathit{vec}\ \ =\ \ \mathit{MkVec2}\ \ 2.0\ 3.0\\ \textbf{in}\ \ \mathit{vec} \odot \mathit{magnitude}\end{array}$$

Note that our full source language contains sugar that allows us to combine the projection dictionaries with the first two let-bindings. See §2.7 for a description of this.

The point of this example is that the type inferencer will not be able to decide whether to use *vec2_magnitude* or *vec3_magnitude* to implement $\mathit{vec} \odot \mathit{magnitude}$ until it determines whether *vec* is a *Vec2* or a *Vec3*.

Here are the constraints for $vec \odot magnitude$, with s_1 being the type of the whole expression:

$$\begin{array}{ll} s_3 & = \text{PROJ } magnitude\ s_2 \\ s_3 & = s_2 \xrightarrow{e_{1a}\ c_{1a}} s_1 \\ e_1 & = e_2 \vee e_{1a} \\ s_2 & = \text{INST } s_{vec} \end{array}$$

Adding these to the type graph yields the following equivalence classes:

$$\begin{array}{lcll} *1 & \sim & s_3 & = \text{PROJ } magnitude\ s_2, \quad s_2 \xrightarrow{e_{1a}\ c_{1a}} s_1 \\ *2 & \sim & s_2 & = \text{INST } s_{vec} \\ !1 & \sim & e_1 & \sqsupseteq e_2 \vee e_{1a} \end{array}$$

Note the dependency between the constraint on s_3 and on s_2. Before we can crush PROJ $magnitude\ s_2$, we must wait until s_2 has been resolved to a concrete type. This requires that we first infer the type scheme for *vec*. Suppose this works out as:

$$vec\ ::\ Vec2\ r_0$$

Instantiating this scheme (which is a no-op in this case) and adding it to the type graph gives:

$$\begin{array}{lcll} *1 & \sim & s_3 & = \text{PROJ } magnitude\ s_2, \quad s_2 \xrightarrow{e_{1a}\ c_{1a}} s_1 \\ *2 & \sim & s_2 & = Vec2\ r_0 \\ !1 & \sim & e_1 & \sqsupseteq e_2 \vee e_{1a} \end{array}$$

Now that we have a constructor for s_2, we can lookup what projection instance function to use for *magnitude* from the corresponding dictionary:

project *Vec2* **with** $magnitude \sim vec2\_magnitude$

This tells us that *magnitude* for *Vec2* types is implemented by $vec2\_magnitude$, so we can crush the PROJ constraint into an appropriate INST:

$$\begin{array}{lcll} *1 & \sim & s_3 & = \text{INST } vec2\_magnitude, \quad s_2 \xrightarrow{e_{1a}\ c_{1a}} s_1 \\ *2 & \sim & s_2 & = Vec2\ r_0 \\ !1 & \sim & e_1 & \sqsupseteq e_2 \vee e_{1a} \end{array}$$

Now we must wait until we have a type scheme for $vec2\_magnitude$. Suppose this scheme works out to be:

$$\begin{array}{l} vec2\_magnitude \\ \quad :: \forall r_1\ r_2\ e_1.\ Vec2\ r_1 \xrightarrow{e_1} Float\ r_2 \\ \quad \rhd e_1 \sqsupseteq Read\ r_1 \end{array}$$

Instantiating this scheme and adding it to the graph gives:

$$
\begin{array}{llll}
*1 & \sim & s_3 & = s_5 \xrightarrow{e_4} s_6, \quad s_2 \xrightarrow{e_{1a}\ c_{1a}} s_1 \\
*2 & \sim & s_2 & = \mathit{Vec2}\ r_0 \\
*3 & \sim & s_5 & = \mathit{Vec2}\ r_4 \\
*4 & \sim & s_6 & = \mathit{Float}\ r_5 \\
!1 & \sim & e_1 & \sqsupseteq e_2 \vee e_{1a} \\
!2 & \sim & e_4 & \sqsupseteq \mathit{Read}\, r_4
\end{array}
$$

Performing the unification in *1 gives:

$$
\begin{array}{llll}
*1 & \sim & s_3 & = s_2 \xrightarrow{e_4\ c_{1a}} s_1 \\
*2 & \sim & s_2,\ s_5 & = \mathit{Vec2}\ r_0 \\
*4 & \sim & s_1,\ s_6 & = \mathit{Float}\ r_5 \\
\%1 & \sim & r_0,\ r_4 & \\
!1 & \sim & e_1 & \sqsupseteq e_2 \vee e_{1a} \\
!2 & \sim & e_4,\ e_{1a} & \sqsupseteq \mathit{Read}\, r_4
\end{array}
$$

Inspecting the constraint on s_1 shows that the overall type of our program is *Float* r_5.

There are a few things we should note before moving on. Firstly, the type of a projection instance function is not required to be the same as another bound to the same label. For example, we could have defined *vec3_magnitude* to return an *Int* or a *List*, instead of a *Float* like with *vec2_magnitude*. Secondly, in general the inferencer must alternate between instantiating type schemes, performing unifications, and crushing projection constraints. Each PROJ that is crushed results in an INST constraint being added to the graph. This INST constraint may need to be resolved, and some unifications performed, before we have the type that another PROJ constraint is waiting for. This behaviour is common when the program includes chains of projections such as:

$$\mathit{exp} \odot \mathit{field1} \odot \mathit{field2} \odot \mathit{field3}$$

The projections in this expression must be handled from left to right. We first determine the type of *exp*, use that to determine what instance function to use for *field1*, add its type to the graph, and unify the resulting constraints. We can then use the solution to determine which instance function to use for *field2*, add its type to the graph, unify constraints, and so on.

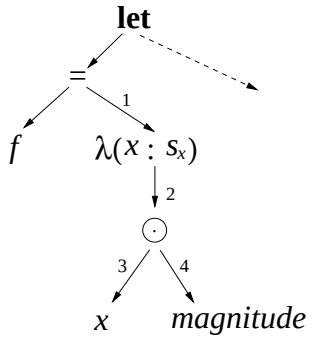
In DDC we implement this behavior by maintaining several work sets of equivalence class identifiers. We have a set for classes that contain types needing to be unified; a set for projection constraints needing to be resolved, and a set for type class constraints needing to be crushed. The inferencer alternates between the various sets, working on one until nothing more can be done, then switching to another. If this process gets stuck, with one of the sets non-empty and no further progress possible, then this is a symptom of having an ambiguous projection constraint in the graph.

3.5.2 Ambiguous projections and type signatures

Ambiguous projections are those that operate on values whose types are not constrained to include an outer constructor. For example, the projection in the following code is ambiguous:

$$\begin{array}{ll}\textbf{let} & f = \lambda x.\ x \odot \mathit{magnitude} \\ \textbf{in} & \ldots\end{array}$$

The abstract syntax tree and constraints for the f binding are:



$$\begin{array}{lll}\text{LET } x & & \\ & s_f & = s_1 \\ & s_1 & = s_x \xrightarrow{e_2\ c_1} s_2 \\ & s_4 & = \text{PROJ } \mathit{magnitude}\ s_3 \\ & s_4 & = s_3 \xrightarrow{e_4\ c_5} s_2 \\ & e_2 & \sqsupseteq e_3 \vee e_4 \\ & s_3 & = s_x\end{array}$$

Adding these constraints to the type graph gives:

$$\begin{array}{rcll} *1 & \sim & s_f,\ s_1 & = s_x \xrightarrow{e_2\ c_1} s_2 \\ *2 & \sim & s_4 & = \text{PROJ } \mathit{magnitude}\ s_x,\ s_x \xrightarrow{e_4\ c_5} s_2 \\ *3 & \sim & s_x,\ s_3 & = \bot \\ !1 & \sim & e_2 & \sqsupseteq e_3 \vee e_4 \end{array}$$

Note that *2 contains two separate type constraints, PROJ $\mathit{magnitude}\ s_x$ and $s_x \xrightarrow{e_5\ c_5} s_2$. As we have no type constructor for s_x we cannot crush the PROJ constraint, and as we cannot represent both the PROJ and function constraints as a normal form type we cannot make a type scheme for f. We can proceed no further, and in this situation our implementation reports an ambiguous projection error.

In future work we plan to investigate the possibility of assigning f a type such as the following:

$$\begin{array}{ll} f & :: \forall a\ b\ e_1\ c_1.\ a \xrightarrow{e_1\ c_1} b \\ & \quad \rhd\ \mathit{HasProj}\ \ \mathit{magnitude}\ \ a\ (a \xrightarrow{e_1\ c_1} b)\end{array}$$

In this type, the constraint $\mathit{HasProj}\ \ \mathit{magnitude}\ \ a\ (a \xrightarrow{e_1\ c_1} b)$ says that a can be any type that supports a projection *magnitude* whose instance function has type $(a \xrightarrow{e_1\ c_1} b)$. Such constraints are discussed in [LL05], and provide some aspects of a true record system [Rem94].

For now, the programmer can fix ambiguous projections by supplying a type signature that constrains the type being projected. Importantly, they only need to supply enough information to allow the projection to be resolved. For example, the programmer could write:

$$\begin{array}{ll}\textbf{let} & f :: \mathit{Vec2} \rightarrow \mathit{Float} \\ & f = \lambda x.\ x \odot \mathit{magnitude} \\ \textbf{in} & \ldots\end{array}$$

This signature does not contain quantifiers, region variables, or effect and closure information. The source desugarer uses the kinds of *Vec2*, *Float*, and the function constructor to determine that this information is missing. It then inserts fresh type variables in the appropriate positions:

$$f :: \mathit{Vec2}\ \ r_6 \xrightarrow{e_7\ c_8} \mathit{Float}\ \ r_9$$

This type is treated as a new constraint, and is added directly to the type graph:

$$\begin{array}{rcll}
*1 & \sim & s_f,\ s_1 & = s_x \xrightarrow{e_2\ c_1} s_2,\ s_5 \xrightarrow{e_7\ c_8} s_6 \\
*2 & \sim & s_4 & = \text{PROJ}\ \mathit{magnitude}\ s_x,\ s_x \xrightarrow{e_5\ c_5} s_2 \\
*3 & \sim & s_x,\ s_3 & = \bot \\
*4 & \sim & s_5 & = \mathit{Vec2}\ \ r_6 \\
*5 & \sim & s_6 & = \mathit{Float}\ \ r_9 \\
!1 & \sim & e_1 & \sqsupseteq e_3 \vee e_4
\end{array}$$

The new constraint gives us enough information to resolve the projection in *2, though we need to perform the unification in *1 to expose it:

$$\begin{array}{rcll}
*1 & \sim & s_f,\ s_1 & = s_x \xrightarrow{e_2\ c_1} s_2 \\
*2 & \sim & s_4 & = \text{PROJ}\ \mathit{magnitude}\ s_x,\ s_x \xrightarrow{e_5\ c_5} s_2 \\
*3 & \sim & s_x,\ s_3,\ s_5 & = \mathit{Vec2}\ \ r_6 \\
*5 & \sim & s_2,\ s_6 & = \mathit{Float}\ \ r_9 \\
!1 & \sim & e_1 & \sqsupseteq e_3 \vee e_4 \\
!2 & \sim & e_2,\ e_7 & \sqsupseteq \bot \\
\%1 & \sim & c_1,\ c_8 & \sqsupseteq \bot
\end{array}$$

Unifying the two function types in *1 has caused s_x and s_5 to be identified. This in turn induces unification of classes *3 and *4. Class *3 now contains *Vec2* r_6, which includes the constructor that the PROJ constraint in *2 was waiting for. We can now lookup the type of the appropriate *magnitude* instance function from the *Vec2* projection dictionary, crush the PROJ constraint to an INST of this type, and complete the inference process as per the example in §3.5.1

3.6 Constraint ordering and mutual recursion

Consider the following program:

$$
\begin{array}{l}
\textbf{project}\ \mathit{Int}\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{even}\ i & = \textbf{if}\ i == 0\ \textbf{then}\ \mathit{True}\ \textbf{else}\ (i-1) \odot \mathit{odd} \\
\mathit{odd}\ i & = \textbf{if}\ i == 0\ \textbf{then}\ \mathit{False}\ \textbf{else}\ (i-1) \odot \mathit{even}
\end{array} \\
\\
\mathit{main}\ () = \mathit{print}\ \ 5 \odot \mathit{odd}
\end{array}
$$

This program defines two projections, then uses the second to determine whether 5 is odd. Now, although we can plainly see that these projection functions are mutually recursive, the inference algorithm does not know this *a priori*. This point should be clearer when we desugar the program into the simplified language described in §3.2

$$
\begin{array}{lll}
\textbf{project} & \mathit{Int} :: \% \to * \ \textbf{with} & \\
\quad \mathit{even} & \sim \mathit{int\_even} & \\
\quad \mathit{odd} & \sim \mathit{int\_odd} & \\
\textbf{let} & \mathit{main} & = \lambda x.\ \textbf{case}\ x\ \textbf{of}\ \mathit{Unit} \to \mathit{print}\ \ 5 \odot \mathit{odd} \\
 & \mathit{int\_even} & = \lambda i_1.\ \textbf{if}\ i_1 == 0\ \textbf{then}\ \ \mathit{True}\ \ \textbf{else}\ (i_1 - 1) \odot \mathit{odd} \\
 & \mathit{int\_odd} & = \lambda i_2.\ \textbf{if}\ i_2 == 0\ \textbf{then}\ \ \mathit{False}\ \ \textbf{else}\ (i_2 - 1) \odot \mathit{even} \\
\textbf{in} & \ldots &
\end{array}
$$

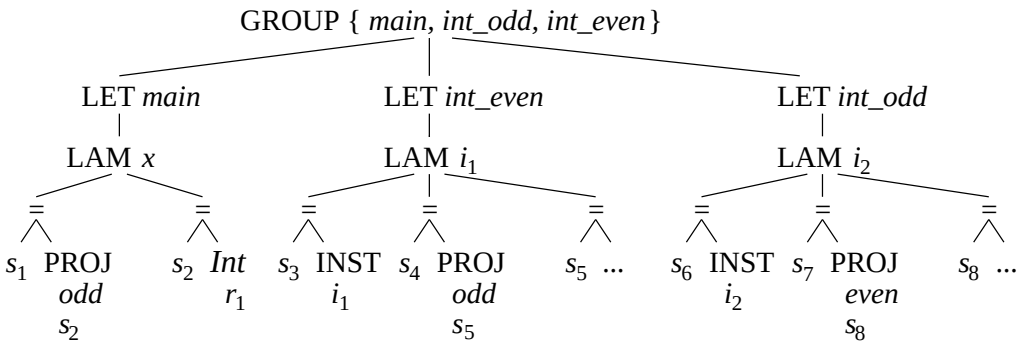
In this program we have introduced new bindings for each of the projection functions, expressed function bindings with λ-abstractions, added the kind signature for *Int*, and renamed the i variables so they have unique names. We have also taken the liberty of moving the binding for *main* to the front of the list, as it will make for a better example. Note that the projection labels *even* and *odd* are not specific to the *int_even* and *int_odd* instance functions. We could have easily reused these labels in projection dictionaries for other types, so the inferencer really does need to infer that $(i_1 - 1)$ is an *Int* before it can decide that $(i_1 - 1) \odot \mathit{even}$ is implemented by *int_even*.

This section gives an overview of how our type inference algorithm handles programs such as this one, that define recursive projections. The main point is that we must compute the binding dependency graph on the fly while we are inferring the types of expressions. We must also reorder bindings on the fly, so that the type of *int_odd* will be known when we come to resolve the $5 \odot \mathit{odd}$ projection in the *main* function.

The constraint tree for this program is shown on the following page, leaving out the constraints that aren't important to the discussion.

GROUP $\{main,\ int\_even,\ int\_odd\}$
  LET *main*
    LAMBDA x
      $s_1 = \text{PROJ}\ odd\ s_2$
      $s_2 = Int\ r_1$
      ...
  LET *int_even*
    LAMBDA i_1
      $s_3 = \text{INST}\ i_1$
      ...
      $s_4 = \text{PROJ}\ odd\ s_5$
      $s_5 = \ldots$
  LET *int_odd*
    LAMBDA i_2
      $s_6 = \text{INST}\ i_2$
      ...
      $s_7 = \text{PROJ}\ even\ s_8$
      $s_8 = \ldots$

For the remainder of this section we will draw our constraint trees graphically, as it makes the presentation clearer. The above tree becomes:



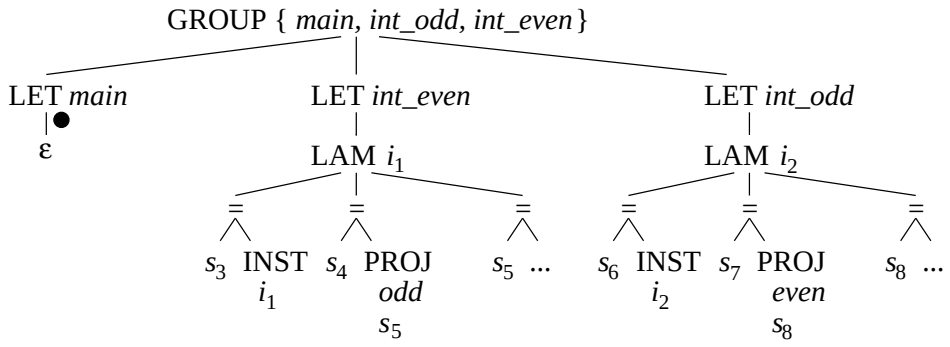
During type inference we perform a left to right, depth first traversal over the constraint tree. As we do this we delete constraints from the tree and add them to the type graph. We start with a full tree and an empty type graph, and finish with a empty tree and a full graph. Internal nodes such as GROUP, LET and LAM organise the type constraints and represent the structure of the original program. We refer to the sub-tree headed by a GROUP, LET or LAM node as a GROUP, LET or LAM *branch*. Once we have added all the constraints in a particular branch to the type graph we can delete the head-node as well. Deleting a LET node also triggers type generalisation, which we will discuss in a moment. Firstly, note that we when we arrive at an INST node, we can determine how the contained variable was bound by examining its parents. In the above example, we see that i_1 is lambda bound. In our practical implementation we maintain a stack of internal nodes for this purpose, pushing them onto the stack as we enter a branch, and popping as we leave.

As mentioned earlier, deletion of a LET node invokes generalisation of the type of the contained variable. However, recall from §3.4 that before we generalise a type from the graph we must pack it into flat form, and this is only possible when the graph is in normal form. The graph is in normal form when no

further reduction rules apply, and when it contains no unresolved PROJ or INST constraints. These two requirements ensure that we have solved all the constraints from a particular binding before we generalise its type.

When performing type inference on a program whose bindings are not in dependency order, or whose bindings are mutually recursive, there will be situations when we wish to leave a LET branch but the graph is not in normal form. This will be because we have no type scheme to satisfy an INST node, or no type constructor to guide the reduction of a PROJ node. In these situations we remove the offending node from the graph and place it back in the tree, then restructure the tree so that further progress can be made before we need to perform the generalisation. This gives us time to infer the required type scheme, or determine the required type constructor, before we have to generalise the original type.

Both of these situations arise when typing the even/odd example on the previous page, so we will work through it now. We use • to indicate where we are in the traversal, and ε to represent an empty branch. After descending into the right most branch and adding the s_1 and s_2 constraints we arrive at:



The type graph is:

$$\begin{array}{lcll} *1 & \sim & s_1 & = \text{PROJ } \mathit{odd}\ s_2 \\ *2 & \sim & s_2 & = \mathit{Int}\ r_1 \end{array}$$

Now, we would like to leave the current branch and generalise the type of *main*, but before we do that we must reduce the graph to normal form. This requires that we resolve the projection constraint in *1. The projection constraint refers to s_2, which is constrained to be $\mathit{Int}\ r_1$. This means that we can look up the instance function to use from the corresponding dictionary. Here is the *Int* dictionary again:

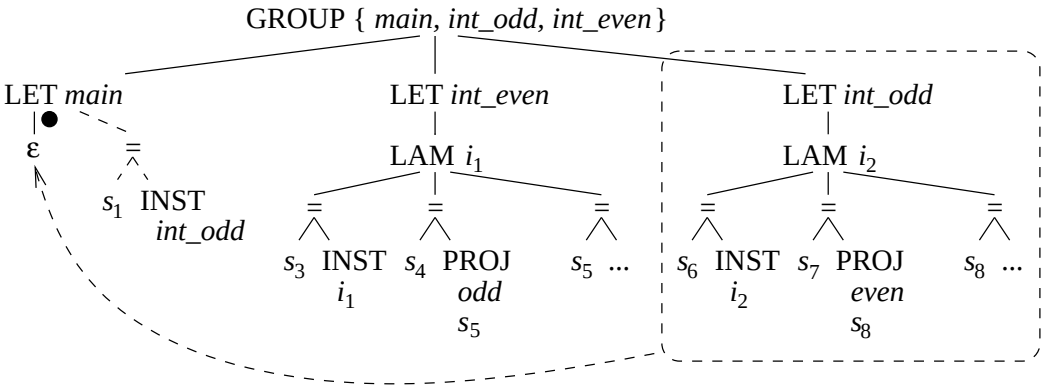
$$\begin{array}{l} \textbf{project}\ \mathit{Int} :: \% \rightarrow * \ \textbf{with} \\ \quad \mathit{even} \sim \mathit{int\_even} \\ \quad \mathit{odd} \sim \mathit{int\_odd} \end{array}$$

The *odd* instance for integers is *int_odd*, so we can crush the PROJ node in the graph into an INST of this function's type. This yields:

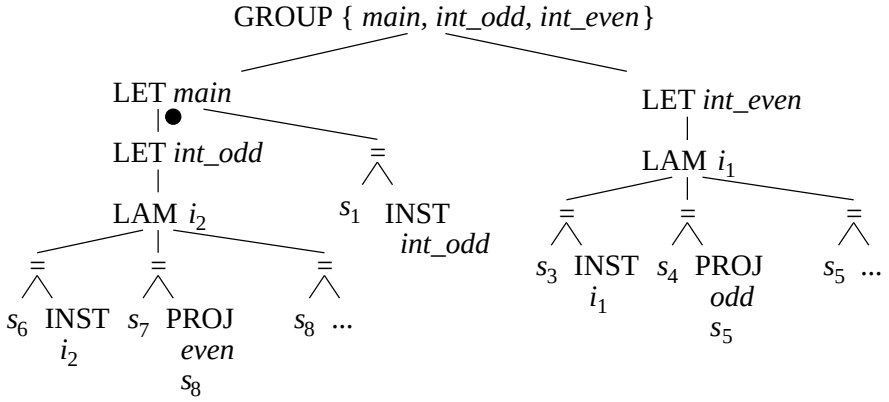
$$\begin{array}{lcll} *1 & \sim & s_1 & = \text{INST } \mathit{int\_odd} \\ *2 & \sim & s_2 & = \mathit{Int}\ r_1 \end{array}$$

Now, we cannot actually instantiate the type for *int_odd* yet because we haven't inferred it. Instead, we will defer further work on the type of *main* and focus

on *int_odd* instead. We do this by removing the $s_1 =$ INST *int_odd* constraint from the graph and placing it back in the tree. We then move the LET *int_odd* branch under LET *main*, so we can work on that before returning to generalise the type of *main*:



Note that the $s_1 =$ INST *int_odd* constraint is placed *after* the LET branch, so that the type for *int_odd* will have been generalised before we need to instantiate it. Completing the move yields:

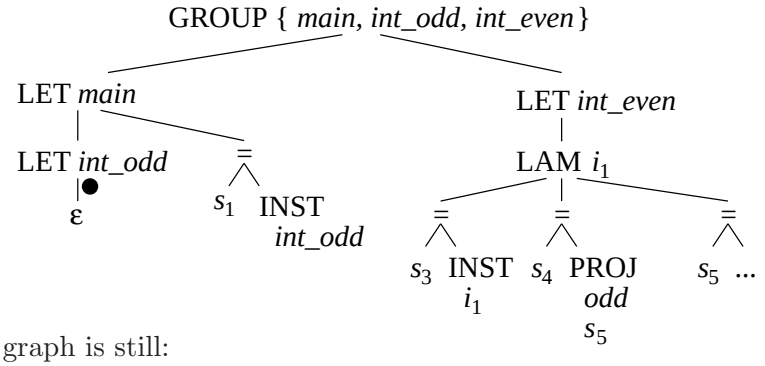


We can now continue our depth first traversal into the *int_odd* branch, adding the constraints for s_6, s_7 and s_8 to the type graph. Assuming s_8 resolves to the type *Int* r_2, we end up with the following graph:

$$
\begin{array}{llll}
*1 & \sim & s_1 & = \bot \\
*2 & \sim & s_2 & = \mathit{Int}\ r_1 \\
*3 & \sim & s_6 & = s_{i_2} \\
*4 & \sim & s_7 & = \text{PROJ}\ \mathit{even}\ s_8 \\
*5 & \sim & s_8 & = \mathit{Int}\ r_2
\end{array}
$$

Note that in our constraint tree, the constraint $s_6 =$ INST i_2 appears under the LAM i_2 node, which tells us that i_2 is lambda bound. As we do not support higher rank types, lambda bound variables do not have polytypes. This means that we do not have to instantiate them, and we can simplify the s_6 constraint to $s_6 = s_{i_2}$.

After the constraints from the *int_odd* branch are added, our constraint tree is:



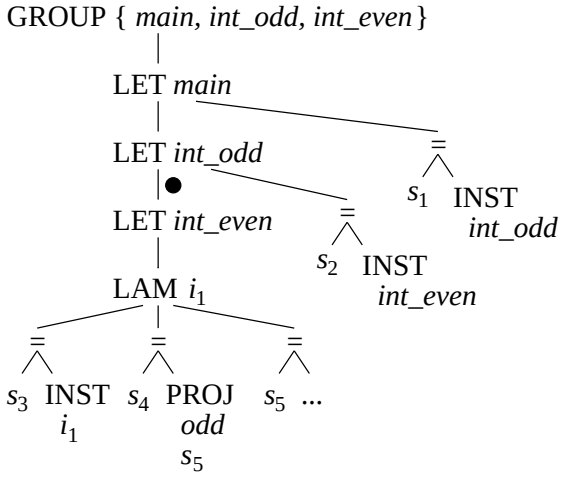
And the graph is still:

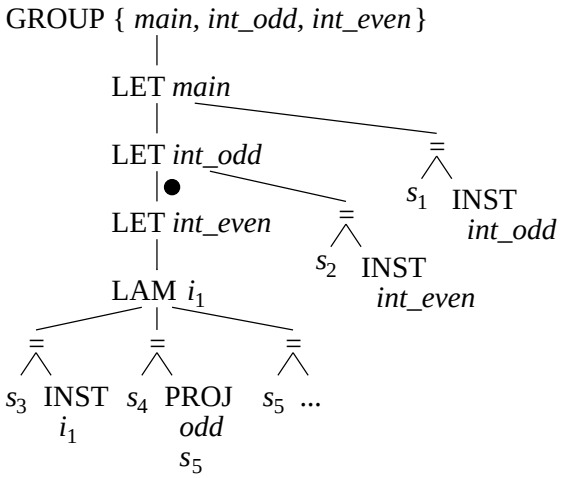
$$
\begin{array}{llll}
*1 & \sim & s_1 & = \bot \\
*2 & \sim & s_2 & = \mathit{Int}\ r_1 \\
*3 & \sim & s_6 & = s_{i_2} \\
*4 & \sim & s_7 & = \text{PROJ}\ \mathit{even}\ s_8 \\
*5 & \sim & s_8 & = \mathit{Int}\ r_2
\end{array}
$$

Now, the • shows that we are still inside the LET *int_odd* branch. However, we cannot leave it yet and generalise the type of *int_odd* because the graph contains an unresolved PROJ constraint, so is not in normal form. As before, this constraint refers to s_8 which an *Int*, so we can lookup the projection instance function from the corresponding dictionary and crush PROJ *even* s_8 to INST *int_even*. Note that crushing a PROJ constraint in this way corresponds to discovering part of the program's call tree, because we now know that *int_odd* calls *int_even*. The new graph is:

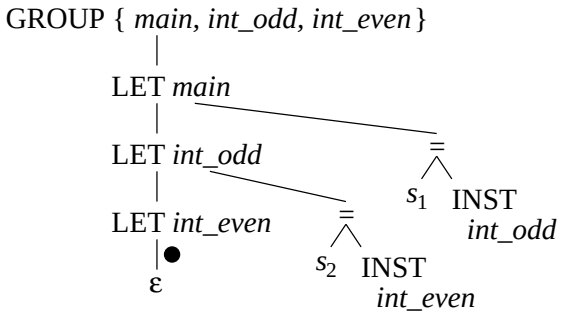
$$
\begin{array}{llll}
*1 & \sim & s_1 & = \bot \\
*2 & \sim & s_2 & = \mathit{Int}\ r_1 \\
*3 & \sim & s_6 & = s_{i_2} \\
*4 & \sim & s_7 & = \text{INST}\ \mathit{int\_even} \\
*5 & \sim & s_8 & = \mathit{Int}\ r_2
\end{array}
$$

We still cannot generalise the type of *int_odd* since it is not in normal form. As before, we will remove the offending $s_7 =$ INST *int_even* constraint from the graph and place it back in the tree, then reorganise the tree so we can make further progress. This gives:





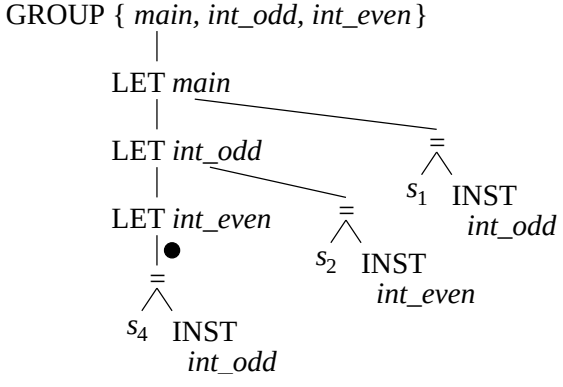
As before, we can continue our traversal by descending into the left of the tree, and adding the constraints for s_3, s_4 and s_5 to the graph. Once this is done we have:



After crushing the $s_4 = \text{PROJ}\ \ odd\ \ s_5$ constraint to $s_4 = \text{INST}\ \ int\_odd$ the type graph becomes:

$$
\begin{array}{llll}
*1 & \sim & s_1 & = \bot \\
*2 & \sim & s_2 & = Int\ \ r_1 \\
*3 & \sim & s_6 & = s_{i_2} \\
*4 & \sim & s_7 & = \bot \\
*5 & \sim & s_8 & = Int\ \ r_8 \\
*6 & \sim & s_3 & = s_{i_1} \\
*7 & \sim & s_4 & = \text{INST}\ \ int\_odd \\
*8 & \sim & s_5 & = Int\ \ r_5
\end{array}
$$

Note that at this point, we have discovered that *int_odd* and *int_even* are mutually recursive. This becomes clear when we place $s_4 =$ INST *int_odd* back in the tree:



In this tree, the fact that the INST *int_odd* constraint appears under LET *int_even* tells us that the binding for *int_even* references *int_odd*, likewise, *int_odd* references *int_even*. As with Haskell [Jon99], in the absence of polymorphic recursion we type mutually recursive bindings using monotypes for the bound variables. This allows us to rewrite the $s_4 = \text{INST } \mathit{int\_odd}$ and $s_2 = \text{INST } \mathit{int\_even}$ constraints to $s_4 = s_{\mathit{int\_odd}}$ and $s_2 = s_{\mathit{int\_even}}$. These can be added directly to the graph without needing to instantiate type schemes for *int_odd* or *int_even*. Note that in this example we have elided the majority of the constraints from the program. In practice, after adding the two constraints $s_4 = s_{\mathit{int\_odd}}$ and $s_2 = s_{\mathit{int\_even}}$ we will need to perform unifications and other reductions to return it to normal form.

Finally, when leaving the LET *int_even* and LET *int_odd* branches, we must wait until we are outside all the branches of a binding group before we generalise their types. This ensures that all constraints from all bindings in the group have been processed, and that we treat the group as a single unit.

3.6.1 Comparison with Helium. 2002 Heeren, Hage, Swierstra.

There are many degrees of freedom to the order in which constraints are processed. If a program contains multiple type errors, then solving constraints in a different order affects which errors are encountered first. For all other programs, changing the order should not affect the substance of the solution.

However, there *is* one overriding restriction. Before the type of a let-bound variable can be generalised into a type scheme, all the constraints from the right of the binding must have been added to the graph. If we fail to do this then the resulting scheme may be instantiated at several different types before we encounter a constraint that would have prevented part of it from being generalised.

We handle this restriction by expressing the type constraints as a tree. Each let-binding corresponds to a branch in the tree, and during our traversal we use the structure of the tree to ensure that constraints from sub-branches are added to the graph before the type of the binding is generalised. In contrast, in the Helium [HHS03, HHS02] compiler, the type constraints fed to the solver have a flat structure, and the requirement to process constraints from the right of a let-binding before generalising the type of the bound variable is handled in a different way.

Helium uses a constraint of the following form:

$$\tau_1 \leq_M \tau_2$$

This constraint says that τ_1 is obtained by first generalising τ_2 with respect to the set of monomorphic variables M, and then instantiating the resulting type scheme. The restriction is enforced by requiring that all constraints involving type variables that are present in τ_2, but cannot appear in M, are processed first. This requirement ensures that the form of τ_2 cannot change once we make the type scheme. We feel that this is a more elegant solution than our own method of dynamically reordering constraint trees.

On the other hand we are unsure whether it is possible to adapt Helium's algorithm to deal with the mutually recursive projection definitions considered

in this section. In [HHS03], the rules given to extract type constraints from the source program require the calculation of the binding dependency graph, and we have seen that this is not possible with recursive projections. It would seem that to calculate the binding dependency graph on the fly, we must retain information about which projections appear in which bindings. It may well be possible to construct a hybrid of Helium's system and our own, but we have not looked into it in detail.

We also wonder about the run time cost of determining which of the $\tau_1 \leq_M \tau_2$ constraints in the graph are ready to be processed. Using a flat constraint tree provides more freedom to choose the order in which constraints are considered, but using a hierarchical one provides more direction. Chapter 4 of [Hee05] contains further discussion of the pros and cons of several related approaches.

Besides the restriction outlined above, it is also important that enough type information makes it into the graph before we are forced to resolve projection constraints. In our current implementation, a particular type is only generalised the first time we need to instantiate it. If there are no bound occurrences of a variable in the module being compiled, which is common for library code, then its type is generalised after all other information is added to the graph. We are unsure whether our chosen approach admits edge cases where a particular projection constraint "should have" been resolved, but wasn't. This would create spurious ambiguous projection errors, but we have not noticed any so far.

3.6.2 Comparison with other constraint systems. Pottier, Sulzmann, Odersky, Wehr *et al*

It is folklore that type inference can be treated as a constraint satisfaction problem. In fact, we feel that a two-phase process of extracting type constraints and then solving them is more natural than the "standard" syntax directed algorithms W and M [Mil78, LY98]. These algorithms achieve the same result, but combine the two phases into a single pass over the source code. The pseudo-code for these algorithms is short and dense, but arguably harder to understand if the reader is not already familiar with them.

Although constraint based inference for simple, monomorphic types is straightforward, complications arise when we add other features. In §3.6 we discussed how the addition of type directed projections, combined with polymorphism using type schemes, requires us to be mindful about the order in which the constraints are handled.

In [Pot00] Pottier presents a system that includes subtyping, conditional constraints and row types. In [Pot95] he considers type inference for a language including subtyping and recursively constrained types. Although these systems use the same basic machinery as ours, namely sets of constraints and simplification rules, their main focus is on managing subtyping relationships that we do not have. Also, the system of [Pot95] is able to judge whether the constraint set is *consistent*, meaning the program is well typed, but it does not actually *solve* the constraints. In contrast, for DDC we need the complete solution, as we use this information to annotate the program when converting between the source and core languages.

In [SOW99] Sulzmann, Odersky and Wehr present HM(X), a general framework for Hindley/Milner type systems with constraints. This system abstracts over

the particular constraint domain being used, along with its reduction rules. In [PR05] Pottier and Remy give a formal presentation of type inference for HM(X). In this work the state of the constraint solver is expressed algebraically, and the algorithm is given as a state rewriting system. The DDC type inferencer can be seen as an extended instance of this system, though we only give a semi-formal description of it in this thesis.

3.7 Type Classes

It is straightforward to add basic support for type class constraints to a graph based inference algorithm. This includes support for "baked in" constraints such as *Mutable* and *Pure*, as well as programmer defined constraints such as *Show* and *Eq*.

The additions to the language are:

Declarations

$decl \rightarrow$	$\ldots$		
$\mid$	**class** $C\ \overline{a}$ **where** $\overline{x :: \varphi}$		(type class declaration)
$\mid$	**instance** $C\ \overline{\tau}$ **where** $\overline{x = t}$		(type class instance)

Types

$\varphi, \tau, \sigma, \varsigma$			
$\rightarrow$	$\ldots$		
$\mid$	$ReadT\ \tau$	$\mid\ WriteT\ \tau$	(effect constructors)

Constraints

$\chi \rightarrow$	$\ldots$		
$\mid$	$C\ \overline{\tau}$		(value type classes)
$\mid$	$Shape\ \tau\ \tau'$	$\mid\ LazyH\ \tau$	
$\mid$	$ConstT\ \tau$	$\mid\ MutableT\ \tau$	
$\mid$	$Const\ r$	$\mid\ Mutable\ r$	(region classes)
$\mid$	$Lazy\ r$	$\mid\ Direct\ r$	
$\mid$	$Pure\ \sigma$		(effect class)
$\mid$	$Empty\ \varsigma$		(closure class)

The new declarations behave the same way as their Haskell counterparts. We use C to represent the programmer defined type class constructors such as *Show* and *Eq*. The meanings of *Shape*, *ConstT* and *MutableT* are discussed in §2.6. The meanings of *Lazy*, *LazyH* and *Direct* were discussed in §2.3.12. *Pure* is discussed in §2.3.9 and *Empty* is discussed in §2.5.6.

When performing inference we represent type class constraints as extra nodes in the graph. For example, the type:

$$\begin{aligned} s_{updateInt} \quad & = Int\, r_1 \rightarrow Int\, r_2 \xrightarrow{e_1\ c_1} () \\ & \rhd e_1 \sqsupseteq Read\, r_2 \vee Write\, r_1 \\ & ,\ \ c_1 \sqsupseteq x : r_1 \\ & ,\ \ Mutable\ r_1 \end{aligned}$$

Would be represented as:

$$\begin{array}{rcll} *1 & \sim & s_{updateInt} & = Int\ r_1 \rightarrow Int\ r_2 \xrightarrow{e_1\ c_1} () \\ !1 & \sim & e_1 & \sqsupseteq Read\ r_2 \vee Write\ r_1 \\ \$1 & \sim & c_1 & \sqsupseteq x : r_1 \\ \Diamond 1 & \sim & Mutable\ r_1 & \end{array}$$

In the type graph we use $\Diamond$ as an identifier for type class equivalence classes.[2] Note that in $\Diamond 1$ we have not used an $=$ or $\sqsupseteq$ operator. Both of these operators are binary and infix, but *Mutable* is unary and prefix. We store multi-parameter type class constraints such as *Shape* in the same way.

Following on from the section on generalisation §3.4, when we trace a type from the graph we must also include any type class constraints that reference variables in the body of the type. For example, if we were to re-trace the type of *updateInt* from the graph, we would need to include *Mutable* r_1. In a real implementation it would be a disaster if we had to inspect every equivalence class in the graph to find all the appropriate constraints. In DDC we mitigate this problem associating each value, region, effect and closure equivalence class, with a set of type class equivalence classes that contain references to it.

Although the programmer defined type classes do not have super class constraints, there are implicit super class constraints on some of the built in ones. We need to reduce these constraints when performing type inference, and the rest of this section discusses how this is done.

3.7.1 Deep Read/Write

$$
\begin{array}{rll}
\text{(deep read data)} & \{\ s = T_\kappa\ \overline{a}, & e \sqsupseteq \mathit{ReadT}\ \ s\ \} \\
\vdash & \{\ s = T_\kappa\ \overline{a}, & e \sqsupseteq \overline{\mathit{ReadT}\ \ b} \vee \overline{\mathit{Read}\ \ r}\} \\
\text{where} & \overline{b} \in \{\ b' & \mid b' \leftarrow \overline{a},\quad b' :: *,\quad b' \in \mathit{mv}(\emptyset,\ T_\kappa\ \overline{a})\ \} \\
 & \overline{r} \in \{\ r' & \mid r' \leftarrow \overline{a},\quad r' :: \%,\quad r' \in \mathit{mv}(\emptyset,\ T_\kappa\ \overline{a})\ \}
\end{array}
$$

A deep read effect such as $\mathit{ReadT}\ \ a$ represents a read on any region variable contained within the as-yet unknown type a. The *ReadT* constructor has kind $* \rightarrow !$, and the "T" in *ReadT* stands for "value type". This distinguishes it from the standard *Read* constructor that works on single regions.

When reducing a deep read on a data type $T\ \overline{\varphi}$, we first separate the argument variables a according to their kinds. Reads on region variables are expressed with the *Read* constructor, and reads on value type variables are be expressed with *ReadT*. Reads on effect and closure arguments can be safely discarded, as there is no associated action at runtime.

It is only meaningful to read (or write) material variables, hence the clauses $b' \in \mathit{mv}(\emptyset,\ T_\kappa\ \overline{a})$ and $r' \in \mathit{mv}(\emptyset,\ T_\kappa\ \overline{a})$. In these clauses, it is safe to use $\emptyset$ for the constraint set. As mentioned in §3.2.4, the *mv* function expresses a simple reachability analysis, but so does the (deep read data) reduction rule. The appropriate read effects will be generated when we perform further reductions on the resulting graph.

Deep writes are handled similarly to deep reads. Note that an implementation must be careful about applying these reduction rules when there are loops through the value portion of the type graph. For example, if we had the following constraints:

$$\{\ s = \mathit{List}\ \ r\ s,\ \ e \sqsupseteq \mathit{ReadT}\ s\ \}$$

This graph cannot be reduced to normal form because each application of (deep read data) to $\mathit{ReadT}\ s$ generates $\mathit{Read}\ r$ as well as another $\mathit{ReadT}\ s$ effect.

[2] Perhaps the type theorist union should start a petition to introduce more synonyms for "constraint", and "class."

$$\begin{array}{lll} \text{(deep read fun)} & \{\ s = \tau_1 \xrightarrow{\sigma\ \varsigma} \tau_2, & e \sqsupseteq \mathit{ReadT}\ \ s\ \} \\ \vdash & \{\ s = \tau_1 \xrightarrow{\sigma\ \varsigma} \tau_2, & e \sqsupseteq \bot\ \} \end{array}$$

The rule (deep read fun) shows that deep reads (and writes) on function types can be removed from the graph. Function values do not contain material objects that are capable of being updated.

3.7.2 Deep Mutable/Const

Deep mutability and constancy constraints are reduced in a similar way to deep read and write effects.

$$\begin{array}{lll} \text{(deep mutable)} & \{\ s = T\ \overline{a}, & \mathit{MutableT}\ \ s\ \} \\ \vdash & \{\ s = T\ \overline{a}, & \overline{\mathit{MutableT}\ \ b},\ \overline{\mathit{Mutable}\ \ r}\ \} \end{array}$$

$$\begin{array}{lll} \text{where} & \overline{b} \in \{\ b' & \mid b' \leftarrow \overline{a},\quad b' :: *,\quad b' \in mv(\emptyset,\ T_\kappa\ \overline{a})\ \} \\ & \overline{r} \in \{\ r' & \mid r' \leftarrow \overline{a},\quad r' :: \%,\quad r' \in mv(\emptyset,\ T_\kappa\ \overline{a})\ \} \end{array}$$

3.7.3 Purification

$$\begin{array}{lll} \text{(purify)} & \{\ e \sqsupseteq \mathit{Read}\ \ r, & \mathit{Pure}\ \ e\ \} \\ \vdash & \{\ e \sqsupseteq \mathit{Read}\ \ r, & \mathit{Pure}\ \ e, \quad \mathit{Const}\ \ r\} \end{array}$$

$$\begin{array}{lll} \text{(deep purify)} & \{\ e \sqsupseteq \mathit{ReadT}\ \ s, & \mathit{Pure}\ \ e\ \} \\ \vdash & \{\ e \sqsupseteq \mathit{ReadT}\ \ s, & \mathit{Pure}\ \ e, \quad \mathit{ConstT}\ \ s\} \end{array}$$

$$\begin{array}{lll} \text{(purify trans)} & \{\ e_1 \sqsupseteq e_2, & \mathit{Pure}\ \ e_1\ \} \\ \vdash & \{\ e_1 \sqsupseteq e_2, & \mathit{Pure}\ \ e_1, \quad \mathit{Pure}\ \ e_2\} \end{array}$$

To purify a *Read* effect on a region r, we constrain that region to be constant by adding $\mathit{Const}\ \ r$ to the graph. Purification of deep reads is similar. As discussed in §2.3.10, we choose to leave the original *Read* effect in the graph, though we could equally remove it.

We must leave the $\mathit{Pure}\ \ e$ constraint in the graph. If the equivalence class containing e is unified with another, then these new effects need to be purified as well.

For example, suppose we had the following constraint set:

$$(1) \quad \begin{array}{lll} \{ & s = a \xrightarrow{e_1} b, & e_1 \sqsupseteq \mathit{Read}\ \ r_1, \quad \mathit{Pure}\ \ e_1, \\ & s = a \xrightarrow{e_2} b, & e_2 \sqsupseteq \mathit{Read}\ \ r_2, \quad \mathit{Mutable}\ \ r_2 \quad \} \end{array}$$

If we were to set r_1 constant while removing the $\mathit{Pure}\ \ e_1$ constraint we would get:

$$(2) \quad \begin{array}{lll} \{ & s = a \xrightarrow{e_1} b, & e_1 \sqsupseteq \mathit{Read}\ \ r_1, \\ & s = a \xrightarrow{e_2} b, & e_2 \sqsupseteq \mathit{Read}\ \ r_2, \quad \mathit{Mutable}\ \ r_2, \\ & \mathit{Const}\ \ r_1 & \quad \} \qquad \text{(bad purify, 1)} \end{array}$$

Performing unification on s gives:

$$(3)\quad \{\ \ s\ = a \xrightarrow{e_1} b,\quad e_1 \sqsupseteq \mathit{Read}\ r_1,\quad e_1 \sqsupseteq \mathit{Read}\ r_2,\quad \mathit{Mutable}\ r_2,\quad \mathit{Const}\ r_1,\quad e_1 = e_2\ \ \} \qquad \text{(unify fun, 2)}$$

Now, although we used to have a purity constraint on e_1, this effect now includes a read of the mutable region r_2. In addition, there is nothing left in the constraint set to indicate that such an effect is in any way invalid. On the other hand, if we were to take our original constraint set and perform the unification first, then we would get:

$$(4)\quad \{\ \ s\ = a \xrightarrow{e_1} b,\quad e_1 \sqsupseteq \mathit{Read}\ r_1,\quad \mathit{Pure}\ e_1,\quad e_1 \sqsupseteq \mathit{Read}\ r_2,\quad \mathit{Mutable}\ r_2,\quad e_1 = e_2\ \ \} \qquad \text{(unify fun, 1)}$$

Applying the bad purify rule to this new set yields:

$$(5)\quad \{\ \ s\ = a \xrightarrow{e_1} b,\quad e_1 \sqsupseteq \mathit{Read}\ r_1,\quad e_1 \sqsupseteq \mathit{Read}\ r_2,\quad \mathit{Mutable}\ r_2,\quad e_1 = e_2,\quad \mathit{Const}\ r_1,\quad \mathit{Const}\ r_2\ \ \} \qquad \text{(bad purify, 4)}$$

In this case we have both $\mathit{Mutable}\ r_2$ and $\mathit{Const}\ r_2$ in the final constraint set, which indicates a type error. Removing the purity constraint from the graph has caused our reduction to be non-confluent.

3.7.4 Shape

$$\begin{array}{ll} \text{(shape left)} & \{\ s_1 = T_\kappa\ \overline{a},\quad \mathit{Shape}\ s_1\ s_2\ \} \\ \vdash & \{\ s_1 = T_\kappa\ \overline{a},\quad s_2 = \varphi'\ \} \cup \mathit{addShape}(\emptyset,\ T_\kappa\ \overline{a},\ \varphi') \\ & \text{where}\ \ \varphi' = \mathit{freshen}(\emptyset,\ T_\kappa\ \overline{a}) \end{array}$$

$$\begin{array}{ll} \text{(shape right)} & \{\ s_2 = T_\kappa\ \overline{a},\quad \mathit{Shape}\ s_1\ s_2\ \} \\ \vdash & \{\ s_2 = T_\kappa\ \overline{a},\quad s_1 = \varphi'\ \} \cup \mathit{addShape}(\emptyset,\ T_\kappa\ \overline{a},\ \varphi') \\ & \text{where}\ \ \varphi' = \mathit{freshen}(\emptyset,\ T_\kappa\ \overline{a}) \end{array}$$

$$\begin{array}{ll} \mathit{freshen}(SM,\ a_\kappa) & \\ \quad \mid a_\kappa \in SM\ \text{ and } \kappa \in \{*, \%\} & = a'_\kappa\ \text{fresh} \\ \mathit{freshen}(SM,\ T_\kappa\ \overline{\varphi}) & = T_\kappa\ \overline{\mathit{freshen}(SM \cup \mathit{smv}(\emptyset, T_\kappa\ \overline{\varphi}),\ \varphi)} \\ \mathit{freshen}(SM,\ \varphi) & = \varphi \end{array}$$

$$\begin{array}{ll} \mathit{addShape}(SM,\ a_*,\ a'_*) & \\ \quad \mid a_* \in SM\ \text{ and } a_* \neq a'_* & = \{\ \mathit{Shape2}\ a_*\ a'_*\ \} \\ \mathit{addShape}(SM,\ T_\kappa\ \overline{\varphi},\ T_\kappa\ \overline{\varphi'}) & = \bigcup \overline{\mathit{addShape}(SM \cup \mathit{smv}(\emptyset, T_\kappa\ \overline{\varphi}),\ \varphi,\ \varphi')} \\ \mathit{addShape}(SM,\ \varphi,\ \varphi') & = \emptyset \end{array}$$

When reducing a constraint like *Shape* $s_1\ s_2$, the choice of what rule to use depends on whether we have a type constructor for s_1 or s_2. If we have a constructor for s_1, we can use this as a template to constrain the type of s_2, and *vice versa*. If we have a constructor for both s_1 and s_2, then it does not matter which of the rules we use.

Example

We will use the *FunThing* type as an example. A *FunThing* can contain an integer, a character, a function that takes an integer, or a thing of arbitrary type.

$$\begin{array}{ll} \textbf{data}\ \mathit{FunThing}\ r_1\ r_2\ a_1\ a_2\ e_1\ c_1 & \\ \quad = & \mathit{FInt}\ (\mathit{Int}\ r_1) \\ \quad \mid & \mathit{FChar}\ (\mathit{Char}\ r_2) \\ \quad \mid & \mathit{FFun}\ (\mathit{Int}\ r_2 \xrightarrow{e_1\ c_1} a_1) \\ \quad \mid & \mathit{FThing}\ a_2 \end{array}$$

Suppose we have constraints:

$$\begin{array}{ll} \{ & s_1 = \mathit{FunThing}\ r_1\ r_2\ s_2\ s_3\ e_1\ c_1 \\ & s_2 = \mathit{Int}\ r_3 \\ & s_3 = \mathit{Int}\ r_4 \\ & e_1 \sqsupseteq \mathit{Read}\ r_2 \vee \mathit{Read}\ r_5 \\ & c_1 \sqsupseteq \mathit{Int}\ r_5 \\ & \mathit{Shape}\ s_1\ s_1' \qquad\qquad\qquad\qquad\qquad \} \end{array}$$

As we have a constructor for s_1 we can use the (shape left) rule. The material variables of $\mathit{FunThing}\ r_1\ r_2\ s_2\ s_3\ e_1\ c_1$ are:

$$mv(\emptyset, \mathit{FunThing}\ r_1\ r_2\ s_2\ s_3\ e_1\ c_1) \quad = \{r_1,\ r_2,\ a_2\}$$

Whereas the immaterial variables are:

$$iv(\emptyset, \mathit{FunThing}\ r_1\ r_2\ s_2\ s_3\ e_1\ c_1) \quad = \{r_2,\ e_1,\ c_1, a_1\}$$

This means the strongly material variables are:

$$smv(\emptyset, \mathit{FunThing}\ r_1\ r_2\ s_2\ s_3\ e_1\ c_1) \quad = \{r_1,\ a_2\}$$

Note that when reducing the *Shape* constraint, region variables that are only reachable from the closure of a type, such as r_5, are not counted as material. Similarly to the example given in §2.6.5, the programmer cannot copy objects in such regions, so we do not freshen the associated region variables. This is achieved in part by passing $\emptyset$ as the first argument of *freshen* and *addShape*, instead of the full set of constraints being reduced. This ensures that these functions do not have information about the c_1 constraint.

The alternative would be to freshen c_1 as well, and create a new constraint $c_1' \sqsupseteq \mathit{Int}\ r_5'$. We would also need to create a new version of the constraint on e_1, and ensure that this referred to the copied closure. Of course, actually copying the objects in the closures of functions would require additional support from the runtime system, so we have not considered it further.

Applying the *freshen* function to the type of s_1 gives us the new type:

$$s_1' = \mathit{FunThing}\ r_1'\ r_2\ s_2\ s_3'\ e_1\ c_1 \qquad \text{with } r_1', s_3' \text{ fresh}$$

Applying the *addShape* function provides *Shape* constraints on the components of this type:

$$
\begin{array}{l}
\mathit{addShape}(\emptyset,\ \mathit{FunThing}\ r_1\ r_2\ s_2\ s_3\ e_1\ c_1,\ \mathit{FunThing}\ r_1'\ r_2\ s_2\ s_3'\ e_1\ c_1) \\
\quad \equiv \quad \mathit{addShape}(SM,\ r_1,\ r_1') \quad \cup\ \mathit{addShape}(SM,\ r_2,\ r_2) \\
\quad \cup \quad \mathit{addShape}(SM,\ s_2,\ s_2) \quad \cup\ \mathit{addShape}(SM,\ s_3,\ s_3') \\
\quad \cup \quad \mathit{addShape}(SM,\ e_1,\ e_1) \quad \cup\ \mathit{addShape}(SM,\ c_1,\ c_1) \\
\qquad \textbf{where}\ SM = \{r_1,\ s_3\} \\
\quad \equiv \quad \mathit{Shape}\ s_3\ s_3'
\end{array}
$$

So our final result is:

$$
\begin{array}{ll}
\{ & s_1 = \mathit{FunThing}\ r_1\ r_2\ s_2\ s_3\ e_1\ c_1 \\
& s_2 = \mathit{Int}\ r_3 \\
& s_3 = \mathit{Int}\ r_4 \\
& e_1 \sqsupseteq \mathit{Read}\ r_2 \vee \mathit{Read}\ r_5 \\
& c_1 \sqsupseteq \mathit{Int}\ r_5 \\
& s_1' = \mathit{FunThing}\ r_1'\ r_2\ s_2\ s_3'\ e_1\ c_1 \\
& \mathit{Shape}\ s_3\ s_3' \qquad\qquad \}
\end{array}
$$

Note that in the new type *FunThing* $r_1'\ r_2\ s_2\ s_3'\ e_1\ c_1$, the fresh variables r_1' and s_3', correspond to just the components of the underlying *FunThing* values that can potentially be coppied. r_2 and s_2 are not freshened because these variables are used in the parameter and return types of an embedded function value. Likewise e_1 and c_1 are not freshened because they do not represent data objects.

3.8 Error Reporting

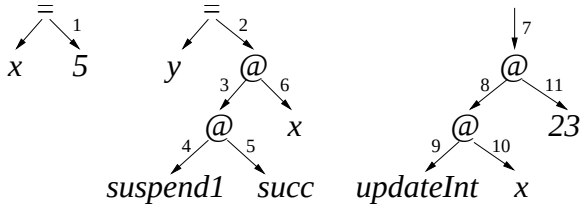
In a constraint based inference algorithm, the natural way to include error reporting is to add justifications to each of the constraints extracted from the source program. These justifications can be maintained as the graph is reduced, and if we encounter an error we can use the justification to determine why the conflicting constraints were added to the graph. This approach is outlined by Wand [Wan86], and elaborated by Duggan [DB96], Stuckey [SSW04] and Heeren [Hee05]. We will give an overview of the general ideas, and focus on how we manage the constraints that are specific to Disciple.

Consider the *succDelay* program from §2.3.9

$$
\begin{array}{l}
\mathit{succDelay}\ () \\
=\ \textbf{do} \quad x = 5 \\
\qquad\quad y = \mathit{succ}\ @\ x \\
\qquad\quad \ldots \\
\qquad\quad x := 23 \\
\qquad\quad \ldots
\end{array}
$$

This program has a purity conflict. The binding for y creates a suspension that will read the value of x, but later in the program this value is updated. This implies that the value of y will depend on when it is forced, which is a program behaviour we take as being invalid.

Here are the syntax trees for the three statements in the do-block, after desugaring:



After extracting type constraints and solving them, we are left with a type graph containing the following equivalence classes:

$$
\begin{array}{lcll}
*1 & \sim & s_x,\ a,\ s_1\ s_6,\ s_{10} & = \mathit{Int}\ r_1 \\
*2 & \sim & s_y,\ b,\ s_2 & = \mathit{Int}\ r_6 \\
*3 & \sim & s_3 & = s_x \xrightarrow{e_2\ c_2} s_y \\
*4 & \sim & s_4 & = s_5 \xrightarrow{e_3\ c_3} s_3 \\
*5 & \sim & s_5 & = s_x \xrightarrow{e_5\ c_5} s_y \\
 & \ldots & & \\
!1 & \sim & e_5,\ e_6 & \sqsupseteq \mathit{Read}\ r_1 \\
!2 & \sim & e_7,\ e_9 & \sqsupseteq \mathit{Read}\ r_{10} \vee \mathit{Read}\ r_1 \\
 & \ldots & & \\
\Diamond 1 & \sim & \mathit{Pure}\ e_5 & \\
\Diamond 2 & \sim & \mathit{Const}\ r_1 & \\
\Diamond 3 & \sim & \mathit{Mutable}\ r_1 &
\end{array}
$$

This graph contains an obvious type error. $\Diamond 2$ contains a constraint that requires r_1 to be constant, while $\Diamond 3$ contains a constraint that requires it to be mutable. As discussed in §4.1, we cannot convert this example to a valid core program because there is no way to construct witnesses for both of these constraints at once. Unfortunately, the *reason* for this error is not so obvious. It would be unhelpful for a compiler to simply report that it "cannot create core witnesses", or that "*Const* r_1 conflicts with *Mutable* r_1". Neither of these messages help us determine what part of the program caused the error, or suggest how we might resolve it.

3.8.1 Constraint justifications

As mentioned previously, we track the source of type errors by attaching justifications to each of the constraints from the program. For example:

$$
\begin{array}{lll}
s_{4u} & = s_5 \xrightarrow{e_3\ c_3} s_3 & \mid i_1 \\
s_{4d} & = \mathrm{INST}\ s_{suspend} & \mid i_2 \\
s_{4u} & = s_{4d} & \mid i_3 \\
s_5 & = \mathrm{INST}\ s_{succ} & \mid i_4 \\
s_9 & = \mathrm{INST}\ s_{updateInt} & \mid i_5 \\
i_1 & = \mathit{IApp}\ 3\ \mathit{suspend}\ \mathit{succ} & \\
i_2 & = \mathit{IVar}\ 3\ \mathit{suspend} & \\
i_3 & = \mathit{IUseDef}\ 3\ \emptyset &
\end{array}
$$

$$
\begin{array}{ll}
i_4 & = IVar\ \ 3\ \ succ \\
i_5 & = IVar\ \ 5\ \ updateInt \\
\ldots &
\end{array}
$$

We now write value type constraints as $s = \tau \mid i$, where s is a type variable, τ is a type, and i is a *source information variable*. In this presentation we will refer to source information as just "information". We extend region, effect, closure and type class constraints in a similar way. Information variables are bound to information expressions in separate constraints. Information expressions are built with *information constructors* such as *IApp* and *IVar*.

We give information constructors informal, descriptive kinds:

$$
\begin{array}{ll}
IApp & :: \text{num} \rightarrow \text{exp} \rightarrow \text{exp} \rightarrow \text{info} \\
IVar & :: \text{num} \rightarrow \text{var} \rightarrow \text{info} \\
IUseDef & :: \text{num} \rightarrow \text{Set info} \rightarrow \text{info}
\end{array}
$$

IApp takes a line number and two expressions. It produces information that says a particular constraint is due to the application of these two expressions, on that line of the source program. *IVar* produces information that says a constraint is due to the use of a particular variable. *IUseDef* produces information that says a constraint is due to the fact that the definition of a variable must match its use. The first argument of *IUseDef* is a line number as before, but the second argument is a set of other information expressions. We will see how this works in a moment.

We now discuss how to record source information in the type graph representation. For constraints involving a constructor, we can take the information variable from the constraint, and use it to annotate the constructor as it is placed into an equivalence class. When performing an instantiation due to an INST constraint, the information variable on the INST constructor is propagated to all the new constructors created during the instantiation. For example, after adding the above constraints to the type graph and performing the instantiations we get:

$$
\begin{array}{llll}
*1 & \sim & s_{4u} & = (s_5 \xrightarrow{e_3\ c_3} s_3)^{i_1} \\
*2 & \sim & s_{4d} & = (s_{41} \xrightarrow{e_{4d}\ c_{4d}} s_{42})^{i_2} \\
*3 & \sim & s_{41} & = (a \xrightarrow{e_{41}\ c_{41}} b)^{i_2} \\
*4 & \sim & s_{42} & = (a \xrightarrow{e_{42}\ c_{42}} b)^{i_2} \\
!1 & \sim & e_5 & \sqsupseteq (Read\ \ r_1)^{i_2} \\
\Diamond 1 & \sim & (Pure\ \ e_{41})^{i_2} & \\
\Diamond 3 & \sim & (Mutable\ \ r_2)^{i_4} & \\
\circ 1 & \sim & i_1 & = IApp\ \ 3\ \ suspend\ \ succ \\
 & \ldots & &
\end{array}
$$

Note that we have used $\circ$ to identify the equivalence classes that hold source information expressions. For a constraint of the form $s_{4u} = s_{4d} \mid i_3$, assuming that s_{4u} and s_{4d} are not already in the same class, adding this constraint to the graph may result in types being unified.

As we wish to track the reason for this unification, we modify the reduction rule for unification so that it combines the information about the types being unified:

$$
\begin{array}{lll}
\text{(unify fun info)} & \{ & s_1 = s_2 \mid i_1, \\
& & s_1 = a_1 \xrightarrow{e_1\ c_1} b_1 \mid i_2, \quad s_2 = a_2 \xrightarrow{e_2\ c_2} b_2 \mid i_3 \\
& & i_1 = \mathit{IUseDef}\ n\ \emptyset \\
& \} & \\
\vdash & \{ & s_1 = s_2 \mid i_1, \\
& & s_1 = a_1 \xrightarrow{e_1\ c_1} b_1 \mid i_4 \\
& & a_1 = a_2 \mid i_4, \qquad\qquad b_1 = b_2 \mid i_4 \\
& & e_1 = e_2 \mid i_4, \qquad\qquad c_1 = c_2 \mid i_4 \\
& & i_1 = \mathit{IUseDef}\ n\ \emptyset \\
& & i_4 = \mathit{IUseDef}\ n\ \{i_2,\ i_3\} \\
& \} &
\end{array}
$$

The rule for unification of data types is similar. The new information constraint $i_4 = \mathit{IUseDef}\ n\ \{i_2,\ i_3\}$ records why the value, effect and closure constraints in the conclusion of the rule were added to the graph. It also contains the information variables from both types that were unified. Returning to our example, we use this new unification rule to add the constraint $s_{4u} = s_{4d} \mid i_3$ to the type graph:

$$
\begin{array}{rcll}
*1 & \sim & s_{4d},\ s_{4u} & = (s_5 \xrightarrow{e_3\ c_3} s_3)^{i_6} \\
*3 & \sim & s_{41},\ s_5 & = (a \xrightarrow{e_{41}\ c_{41}} b)^{i_2} \\
*4 & \sim & s_{42} & = (a \xrightarrow{e_{42}\ c_{42}} b)^{i_2} \\
\circ 1 & \sim & i_1 & = \mathit{IApp}\ 3\ \mathit{suspend}\ \mathit{succ} \\
\circ 2 & \sim & i_2 & = \mathit{IVar}\ 3\ \mathit{suspend} \\
\circ 6 & \sim & i_6 & = \mathit{IUseDef}\ 3\ \{i_1,\ i_2\} \\
& \ldots & &
\end{array}
$$

Unfortunately, although the *set* representation contains full information about why a particular constraint is present, in our type graph representation some of this information is lost. Since we attach source information to constructors only, we have no way of adding a constraint like $s_1 = s_2 \mid i_1$ to an empty graph without losing the information variable i_1. For example, doing so yields:

$$
\begin{array}{rcll}
*1 & \sim & s_1,\ s_2 & = \bot
\end{array}
$$

On the left of the $=$ we have the list of variables in this equivalence class, with the canconical name s_1 appearing first. This list represents a substitution, where all occurrences of s_2 in the graph should be replaced by s_1. We have recorded this bare fact, but have lost the reason *why* such a substitution should take place. For this reason we will stick with the constraint set representation for the rest of this section.

3.8.2 Tracking purity errors

Returning to the *succDelay* example from §3.8, we can track the source of the purity error by modifying the reduction rules for purity constraints:

$$
\begin{array}{lll}
\text{(purify info)} & \{\ e_1 \sqsupseteq \mathit{Read}\ r_1 \mid i_1, & \mathit{Pure}\ e_1 \mid i_2\ \} \\
\vdash & \{\ e_1 \sqsupseteq \mathit{Read}\ r_1 \mid i_1, & \mathit{Pure}\ e_1 \mid i_2, \\
 & \quad i_3 = \mathit{IPurify}\ r_1\ i_1\ i_2, & \mathit{Const}\ r_1 \mid i_3\ \} \\
\\
\text{(purify trans info)} & \{\ e_1 \sqsupseteq e_2 \mid i_1, & \mathit{Pure}\ e_1 \mid i_2\} \\
\vdash & \{\ e_1 \sqsupseteq e_2 \mid i_1, & \mathit{Pure}\ e_1 \mid i_2, \\
 & \quad i_3 = \mathit{IPurifyTrans}\ e_1\ i_2, & \mathit{Pure}\ e_2 \mid i_3\ \}
\end{array}
$$

The (purify info) rule is an extension of (purify) from §3.7. The resulting *Const* r_1 constraint is now tagged with an information variable i_3. The constraint $i_3 = \mathit{IPurify}\ r_1\ i_1\ i_2$ says that *Const* r_1 arose from the purification of a read effect on region r_1. It also includes the information variables from the associated effect and purity constraints. We modify the (deep purify) rule in a similar manner.

The (purify trans info) rule extends (purify trans) from §3.7. The resulting *Pure* e_2 constraint is tagged with a variable i_3, and the information constraint $i_3 = \mathit{IPurifyTrans}\ e_1\ i_2$ records the variables in the premise of the rule. *IPurifyTrans* constraints can be used by the compiler to find the original source of a purity constraint. For example, consider the following set:

$$
\begin{array}{lll}
\{\ e_1 \sqsupseteq e_2 \mid i_1, & & \\
\quad e_2 \sqsupseteq e_3 \mid i_2, & & \\
\quad e_3 \sqsupseteq \mathit{Read}\ r_1 \mid i_3, & i_3 = \mathit{IVar}\ 3\ \mathit{succ}, & \\
\quad \mathit{Pure}\ e_1 \mid i_4, & i_4 = \mathit{IVar}\ 3\ \mathit{suspend}, & \\
\quad \mathit{Mutable}\ r_1 \mid i_5, & i_5 = \mathit{IVar}\ 5\ \mathit{updateInt} & \}
\end{array}
$$

This constraint set contains a latent purity conflict, as the *Pure* constraint on e_1 will lead to a *Const* constraint on r_1 when it is reduced. After reduction we have:

$$
\begin{array}{lll}
\{\ e_1 \sqsupseteq e_2 \mid i_1, & & \\
\quad e_2 \sqsupseteq e_3 \mid i_2, & & \\
\quad e_3 \sqsupseteq \mathit{Read}\ r_1 \mid i_3, & i_3 = \mathit{IVar}\ 3\ \mathit{succ}, & \\
\quad \mathit{Pure}\ e_1 \mid i_4, & i_4 = \mathit{IVar}\ 3\ \mathit{suspend}, & \\
\quad \mathit{Pure}\ e_2 \mid i_6, & i_6 = \mathit{IPurifyTrans}\ e_1\ i_4 & \\
\quad \mathit{Pure}\ e_3 \mid i_7, & i_7 = \mathit{IPurifyTrans}\ e_2\ i_6 & \\
\quad \mathit{Const}\ r_1 \mid i_8, & i_8 = \mathit{IPurify}\ r_1\ i_3\ i_7 & \\
\quad \mathit{Mutable}\ r_1 \mid i_5, & i_5 = \mathit{IVar}\ 5\ \mathit{updateInt} & \}
\end{array}
$$

This set contains an obvious type error. In fact, we have two separate symptoms. The first symptom is that r_1 is constrained to be both mutable and constant. The source of the mutability constraint can be obtained directly from the information constraint on i_5. The source of the constancy constraint can be obtained by tracing up the chain of *IPurifyTrans* constraints to reach i_4,

which tells us that it arose due to the use of *suspend* at line 3 in the program. The second symptom is that e_3 is constrained to be pure, but e_3 includes a read effect on a mutable region, which is not pure. This sort of error arises from a three way interaction between the function reading the region, the function writing to it, and the use of suspend. DDC reports the source location of all three function calls.

The following page shows the error message obtained when compiling *succDelay*. This message gives the exact reason for the error, though does not suggest how to fix it. In future work we plan to adapt the techniques described in [HJSA02] to estimate which expression in the program is most at fault, and suggest a solution. For example, suppose the program contains several expressions that update a particular object, but only one occurrence of a function that reads it being suspended. In this case it is likely that the program is based around destructive update, and that the suspension is "more wrong" than the mutability of the object. The compiler could then suggest that the offending function application is evaluated strictly, instead of being suspended, or that the object is copied beforehand.

```
./test/Error/Purity/PurifyReadWrite1/Main.ds:9:23
    Cannot purify Read effect on Mutable region.
      A purity constraint on a Read effect requires the
      region it acts on to be Const, and it cannot be
      Mutable at the same time.

              the use of: succ
             with effect: !Read %r1
                      at: ./Main.ds:9:18

        is being purified by
              the use of: suspend1
                      at: ./Main.ds:9:23

        which conflicts with
              constraint: Mutable %r1
         from the use of: (:=)
                      at: ./Main.ds:10:16
```

Chapter 4

Core Language

Our core language is based on System-F, and includes a witness passing mechanism similar to one in System-Fc [SCPJD07] which is used in GHC. Our language is typed, and these types are used as both an internal sanity check, and to guide code optimisations. This thesis discusses a few optimisations, though we do not offer any new ones. What we present is a framework whereby optimisations previously reserved for pure languages can be applied to ones that include side effects and mutability polymorphism.

With regard to optimisation, transforms that do not change the order of function applications, and do not modify the sharing properties of data, are equally applicable to both pure and impure languages. For example, the case-elimination transform from [dMS95] is effect agnostic. Inlining function definitions into their call sites also does not present a problem, provided the function arguments are in normal form. This restriction prevents the duplication of computations at runtime, and is also used in pure languages such as GHC [PJS98]. On the other hand, we need effect information to perform the let-floating transform, as it changes the order of bindings. We also need information about the mutability of data to guide optimisations that have the potential to increase data sharing, such as the full laziness transform.

In this chapter we present the main features of our core language, discuss how to use the type information to perform optimisation, then compare our system with other work. We also give highlights of the proof of soundness, though the bulk of the proof is deferred to the appendix.

4.1 Constraints and evidence

4.1.1 Witness passing

Consider the Haskell function *pairEq* which tests if the two elements of a pair are equal:

$$
\begin{aligned}
&\mathit{pairEq} :: \forall a.\ \mathit{Eq}\ a \Rightarrow (a,\ a) \rightarrow \mathit{Bool} \\
&\mathit{pairEq}\ (x,\ y)\ =\ x == y
\end{aligned}
$$

In the type signature, the constraint $\mathit{Eq}\ a$ restricts the types that a can be instantiated with to just those which support equality. This requirement arises because we have used $(==)$ to compare two values of type a.

As well as being a type constraint, a Haskell compiler such as GHC would treat $\mathit{Eq}\ a$ as the type of an extra parameter to *pairEq*. In this case, the parameter will include an appropriate function to compare the two elements of the pair. During compilation, the compiler will detect applications of *pairEq* and add an extra argument appropriate to the type it is called at. For example, a GHC style translation of *pairEq* to its core language [HHPJW96] would yield something similar to:

$$
\begin{aligned}
&\mathit{pairEq} \\
&=\ \ \Lambda\ a \qquad\ : *. \\
&\quad\ \lambda\ \mathit{comp}\ \ : (a \rightarrow a \rightarrow \mathit{Bool}). \\
&\quad\ \lambda\ \mathit{pair}\ \ \ : (a,\ a). \\
&\quad\ \mathbf{case}\ \mathit{pair}\ \mathbf{of} \\
&\qquad (x,\ y) \rightarrow \mathit{comp}\ \ x\ y
\end{aligned}
$$

while an application of this function in the source language:

$$\mathit{pairEq}\ (2, 3)$$

would be translated to:

$$\mathit{pairEq}\ \ \mathit{Int}\ \ \mathit{primIntEq}\ (\mathit{Pair}\ \ \mathit{Int}\ \ 2\ 3)$$

where *primIntEq* is the primitive equality function on integers.

Returning to the translation of *pairEq*, the extra parameter *comp* binds *evidence* [Jon92] that type a really does support the equality operation – and there is no better evidence than the function which performs it.

With this in mind, suppose that we were only interested in the fact that *pairEq* requires a to support equality, rather than how to actually evaluate this function at runtime. In the above translation, we managed our evidence at the value level, by explicitly passing around a comparison function. Alternatively, we could manage it at the type level:

$$
\begin{aligned}
&\mathit{pairEq} \\
&=\ \ \Lambda\ a \qquad\ : *. \\
&\quad\ \Lambda\ w \qquad\ : \mathit{Eq}\ \ a. \\
&\quad\ \lambda\ \mathit{pair}\ \ : (a,\ a). \\
&\quad\ \mathbf{case}\ \mathit{pair}\ \mathbf{of} \\
&\qquad (x,\ y) \rightarrow (==)\ w\ x\ y
\end{aligned}
$$

In this new translation the extra parameter, w, binds a *proof term*. One step removed from value-level evidence, this type-level proof term serves as *witness* that type a really does support equality, and this is recorded in its kind $Eq\ \ a$. Now, the application of $(==)$ to the elements of the pair requires type a to support equality, and we satisfy this requirement by passing it our witness to the fact.

How a particular calling function happens to manufacture its witnesses is of no concern to *pairEq*, though they do need to enter the system somehow. In the general case, a caller has three options: require the witness to be passed in by an outer function, combine two witnesses into a third, or construct one explicitly.

For this example, the third option suffices, and we can translate the call as:

$$pairEq\ \ Int\ (MkEq\ \ Int)\ (Pair\ \ Int\ \ 2\ 3)$$

The type level function *MkEq* is a *witness constructor* which takes a type, and constructs a witness of kind $Eq\ a$. The expression $(MkEq\ \ Int)$ is as an axiom in our proof system, and it is valid to repeat it in the program when required. In contrast, when we discuss witnesses of constancy and mutability in §4.1.3, their construction will be restricted to certain places in the program, to ensure soundness.

With this plumbing in place we can ensure our code is consistent with respect to which types support equality (or mutability), simply by type checking it in the usual way and then inspecting the way witnesses are constructed.

4.1.2 Dependent kinds

Dependent kinds are kinds that contain types, and in DDC we use them to describe witness constructors. Dependent kinds were introduced by the Edinburgh Logical Framework (LF) [AHM89] which uses them to encode logical rules, and aspects of this framework are present in our core language. Types are viewed as assertions about values, and kinds are viewed as assertions about types.

Functions that take types to kinds are expressed with the Π binder, and we apply such a function by substituting its argument for the bound variable, as usual. For example:

$$\frac{\emptyset \vdash\ MkEq :: \Pi(a : *).\ \ Eq\ \ a \qquad \emptyset \vdash\ Int :: *}{\emptyset \vdash\ MkEq\ \ Int :: Eq\ \ Int}$$

Note that $MkEq\ \ Int$ is a type term, and $Eq\ \ Int$ is a kind term. In this chapter we use the convention that type constructors starting with "*Mk*" produce witnesses.

4.1.3 Witnesses of mutability

When optimising programs involving destructive update, it is of crucial importance that we do not lose track of which regions are mutable and which are supposed to be constant. As mentioned earlier, DDC uses the witness passing

mechanism to keep track of this information, both to guide optimisations and as a sanity check on the intermediate code.

Of primary concern are functions that destructively update objects in the store. For example, ignoring effect and closure information, the *updateInt* function from §2.3.2 has type:

$$updateInt :: \forall r_1\ r_2.\ Mutable\ r_1 \Rightarrow Int\ r_1 \rightarrow Int\ r_2 \rightarrow ()$$

Using ideas from §4.1.1, we treat the region constraint $Mutable\ r_1$ as the kind of an extra type parameter to this function. As we are now considering such constraints to also be type *parameters*, we write them in prefix form with $\Rightarrow$ instead of in postfix form with $\triangleright$ as we did in the source language.

When we call *updateInt* we must pass a witness to the fact that r_1 is indeed mutable, and we now consider how these witnesses should be constructed. We could perhaps construct them directly at call-sites as per our *pairEq* example. However, unlike the type class situation, the various region class witnesses are not necessarily compatible. For example, there is nothing wrong with $MkEq\ a$ and $MkShow\ a$ existing in the same program, but if we have both $MkMutable\ r_1$ and $MkConst\ r_1$ then something has gone badly wrong.

If we were to allow region witnesses to be constructed anywhere in the intermediate code, then the compiler would need access to the whole program to ensure that multiple incompatible witnesses are not constructed for the same region. This would be impossible to implement with respect to separate compilation.

Instead, we require that all witnesses involving a particular region variable are constructed at the same place in the code, namely the point where the variable itself is introduced. As in [TBE+06], we use **letregion** to bring region variables into scope. Here is an example program which creates and integer, updates it, and then prints it to the console:

```
printMe :: () → ()
printMe
 =  λ().
    letregion r1 with { w1 = MkMutable r1 }   in
    letregion r2 with { w2 = MkConst r2 }     in
    do
        x  = 5 r1
        updateInt r1 r2 w1 x (23 r2)
        printInt r1 x
```

Note that in the core language, literal values such as '5' act as constructors that take a region variable and allocate a new object. This gives x the type $Int\ r_1$. The only place the constructors *MkMutable* and *MkConst* may be used is in the set of witness bindings associated with a **letregion**. In addition, we may only create witnesses for the region variable being introduced, and we cannot create witnesses for mutability and constancy in the same set. This ensures that conflicting region witnesses cannot be created.

To call the *updateInt* function we *must* have a witness that r_1 is mutable. Trying to pass another witness, like the one bound to w_2, would result in a type error. With this encoding, it is easy to write code transformations that depend on whether a particular region is mutable or constant. Such a transformation can simply collect the set of region witnesses that are in scope while descending into the abstract syntax tree. We will see an example of this in §4.4.5.

4.1.4 Witnesses of purity

When translating a program which uses lazy evaluation to the core language, we must also construct witnesses of purity. Recall from §2.3.9 that the type of *suspend* is:

$$suspend \quad :: \forall a\ b\ e_1.\ Pure\ e_1 \Rightarrow (a \xrightarrow{e_1} b) \rightarrow a \rightarrow b$$

suspend takes a function of type $a \xrightarrow{e_1} b$, its argument of type a and builds a thunk that represents the suspended function application. When the thunk is forced, the function will be applied to its argument yielding a result of type b. The $Pure\ e_1$ constraint ensures that the function application being suspended has no visible side effects, so the value of its result will not depend on when it is forced.

We now consider how witnesses of purity are created in the core language. Consider the following source program:

$$\begin{array}{lll} fun & :: & \forall a\ r_1\ r_2\ e_1 \\ & . & (Int\ r_1 \xrightarrow{e_1} a) \rightarrow Bool\ r_2 \rightarrow a \\ & \triangleright & Pure\ e_1,\ Const\ r_2 \end{array}$$

$$\begin{array}{lll} fun\ f\ b & & \\ = & \textbf{do} & g\ x \quad = \textbf{if}\ x\ \textbf{then}\ f\ 5\ \textbf{else}\ f\ 23 \\ & & suspend\ g\ b \end{array}$$

fun causes its first parameter to be applied to either 5 or 23, depending on whether its second is true or false. This is done by an auxiliary function, g, and the application of this function is suspended. Because the application of g is suspended it must be pure. Note that the purity of g relies on two separate facts: that f is pure, and that x is constant.

Here is *fun* converted to the core language:

$$\begin{array}{lll} fun & & \\ = & \Lambda\ a\ r_1\ r_2\ e_1. & \\ & \Lambda\ (w_1 & : Pure\ e_1). \\ & \Lambda\ (w_2 & : Const\ r_2). \\ & \lambda\ (f & : Int\ r_1 \xrightarrow{e_1} a). \\ & \lambda\ (b & : Bool\ r_2). \\ & \textbf{do} & \\ & \quad g & = \lambda(x : Bool\ r_2).\ \textbf{if}\ x\ \textbf{then}\ f\ (5\ r_1)\ \textbf{else}\ f\ (23\ r_1) \\ & \quad suspend & (MkPureJoin\ e_1\ (Read\ r_2)\ w_1\ (MkPurify\ r_2\ w_2)) \\ & & g\ b \end{array}$$

When we call *suspend*, the term $(MkPureJoin\ e_1\ (Read\ r_2)\ w_1\ (MkPurify\ r_2\ w_2))$ builds a witness to the fact that g is pure. Note that in this chapter we treat *suspend* as a primitive, so we do not need applications for the argument and return types, or the effect of the function. The typing rule for *suspend* takes care of this parameterisation.

The witness to the purity of g is constructed from two simpler witnesses, one showing that e_1 is pure, and another showing that the read from r_2 is pure. The first is given to us by the calling function, and is bound to w_1.

The second is constructed with the *MkPurify* witness constructor which has kind:

$$\mathit{MkPurify} :: \Pi(r : \%).\ \mathit{Const}\ \ r \rightarrow \mathit{Pure}\ (\mathit{Read}\ \ r)$$

This kind encodes the rule that if a region is constant, then any reads from it can be considered to be pure. When we apply *MkPurify* to r_2, this variable is substituted for both occurrences of r yielding:

$$(\mathit{MkPurify}\ \ r_2) :: \mathit{Const}\ \ r_2 \rightarrow \mathit{Pure}\ (\mathit{Read}\ \ r_2)$$

From the Λ-binding at the beginning of the function we have $w_2 :: \mathit{Const}\ \ r_2$, so applying w_2 as the final argument gives:

$$(\mathit{MkPurify}\ \ r_2\ w_2) :: \mathit{Pure}\ (\mathit{Read}\ \ r_2)$$

Which shows that a read from r_2 is indeed pure.

What remains is to join the two simple witnesses together. This is done with the *MkPureJoin* witness constructor which has kind:

$$\begin{array}{l}\mathit{MkPureJoin} \\ \quad :: \quad \Pi(e_1 :: !).\ \Pi(e_2 :: !) \\ \quad .\quad \mathit{Pure}\ \ e_1 \rightarrow \mathit{Pure}\ \ e_2 \rightarrow \mathit{Pure}\ (e_1 \vee e_2)\end{array}$$

Applying the first two arguments gives:

$$\begin{array}{l}(\mathit{MkPureJoin}\ \ e_1\ (\mathit{Read}\ \ r_2)) \\ \quad :: \quad \mathit{Pure}\ \ e_1 \rightarrow \mathit{Pure}\ (\mathit{Read}\ \ r_2) \rightarrow \mathit{Pure}\ (e_1 \vee \mathit{Read}\ \ r_2)\end{array}$$

This says that if we have a witness that the effect e_1 is pure and a witness that the effect $\mathit{Read}\ \ r_2$ is pure, the combination of these two effects is also pure. Our final witness then becomes:

$$\begin{array}{l}(\mathit{MkPureJoin}\ \ e_1\ (\mathit{Read}\ \ r_2)\ w_1\ (\mathit{MkPurify}\ \ r_2\ w_2)) \\ \quad :: \quad \mathit{Pure}\ (e_1 \vee \mathit{Read}\ \ r_2)\end{array}$$

The effect $e_1 \vee \mathit{Read}\ \ r_2$ is exactly the effect of g, so the above witness is sufficient to prove that we can safely suspend a call to it.

Note that we do not need witnesses of *impurity*. The fact that an expression is pure gives us the capability to suspend its evaluation, and by constructing a witness of purity we prove that this capability exists. In contrast, the fact that an expression is impure is not a capability, because it does not allow us to do anything "extra" with that expression.

4.2 Simplified core language

Symbol Classes

a, r, e, w	$\rightarrow$	(type variable)
x	$\rightarrow$	(value variable)
ρ	$\rightarrow$	(region handles)
l	$\rightarrow$	(store locations)

Super-kinds

ω	::=	$\Diamond$	(super-kind of witness kinds)
	$\mid$	$\Box$	(super-kind of non-witness kinds)
	$\mid$	$\kappa \rightarrow \omega$	(super-kind constructor)

Kinds

κ	::=	$\Pi(a : \kappa_1).\ \kappa_2$	(dependent kind abstraction)
	$\mid$	$\kappa\ \varphi$	(dependent kind application)
	$\mid$	$*\ \mid\ \%\ \mid !$	(atomic kinds)
	$\mid$	$\mathit{Const}\ \mid\ \mathit{Mutable}\ \mid\ \mathit{Pure}$	(witness kind constructors)

Types

φ, τ, σ, δ, Δ

	::=	a	(type variables)
	$\mid$	$\forall(a : \kappa).\ \tau$	(unbounded quantification)
	$\mid$	$\varphi_1\ \varphi_2$	(type application)
	$\mid$	$\sigma_1 \vee \sigma_2\ \mid\ \top\ \mid\ \bot$	(least upper bound, top and bottom)
	$\mid$	$()\ \mid (\rightarrow)\ \mid \mathit{Bool}$	(value type constructors)
	$\mid$	$\mathit{Read}\ \mid\ \mathit{Write}$	(effect type constructors)
	$\mid$	$\mathit{MkConst}\ \mid\ \mathit{MkMutable}$	(region witness constructors)
	$\mid$	$\mathit{MkPure}\ \mid\ \mathit{MkPurify}\ \mid\ \mathit{MkPureJoin}$	(effect witness constructors)
	$\mid$	$\underline{\text{mutable}\ \varphi}\ \mid\ \underline{\text{const}\ \varphi}\ \mid\ \underline{\text{pure}\ \sigma}$	(witnesses)
	$\mid$	$\underline{\rho}$	(region witness)

Terms

t	::=	x	(term variable)
	$\mid$	$\Lambda(a : \kappa).\ t$	(type abstraction)
	$\mid$	$t\ \varphi$	(type application)
	$\mid$	$\lambda(x : \tau).\ t$	(term abstraction)
	$\mid$	$t_1\ t_2$	(term application)
	$\mid$	**let** $x = t_1$ **in** t_2	(let binding)
	$\mid$	**letregion** r **with** $\{\overline{w_i = \delta_i}\}$ **in** t	(region introduction)
	$\mid$	**if** t_1 **then** t_2 **else** t_3	(selection)
	$\mid$	$\mathit{True}\ \varphi \mid \mathit{False}\ \varphi$	(boolean constructors)
	$\mid$	$\mathit{update}\ \delta\ t_1\ t_2$	(boolean update)
	$\mid$	$\mathit{suspend}\ \delta\ t_1\ t_2$	(suspension of function application)
	$\mid$	$()$	(unit value)
	$\mid$	$\underline{l}$	(store location)

Derived Forms

$$
\begin{array}{lll}
\kappa_1 \rightarrow \kappa_2 & \stackrel{\text{def}}{=} & \Pi(\_ :: \kappa_1).\ \kappa_2 \\
\kappa\ \Rightarrow \tau & \stackrel{\text{def}}{=} & \forall(\_ :: \kappa).\ \tau \\
\textbf{do}\ \overline{bindstmt}\ ;\ t & \stackrel{\text{def}}{=} & \textbf{let}\ \overline{\text{mkBind}(bindstmt)}\ \textbf{in}\ t \\
\text{where}\ bindstmt & \rightarrow & x = t\ \mid\ t \\
\quad \text{mkBind}(x = t) & \stackrel{\text{def}}{=} & x = t \\
\quad \text{mkBind}(t) & \stackrel{\text{def}}{=} & x = t,\quad x\ \text{fresh}
\end{array}
$$

The language described in this section is a cut-down version of the language used in our real implementation. To simplify the presentation we have omitted witnesses for direct and lazy regions, along with shape constraints. We have also omitted bounded quantification, effect masking, algebraic data types and case expressions.

Witnesses for direct and lazy regions are handled similarly to the ones for mutable and constant regions. Shape constraints are handled similarly to purity constraints. We will discuss bounded quantification in §4.3 and effect masking in §4.3.1. We have included if-expressions as a simpler version of case-expressions, and limit ourselves to booleans as the only updatable type.

Note also that closure information is not used in the core language, though we discuss one of the benefits that would be gained from adding it in §4.3.2.

4.2.1 Symbol Classes

We use a to mean a type variable of arbitrary kind, r to mean a type variable of region kind, e of effect kind, and w of witness kind. The distinction between these symbols is for convenience only. We will use r when only a variable of region kind makes sense, but an implementation must still check that r does indeed have region kind with respect to the typing rules. We use x to mean a value variable. Region handles ρ are terminal symbols that correspond to a set of locations l, in the store.

Note that in a practical implementation it is desirable to attach type and kind information to value and type variables. This allows us to reconstruct the type or kind of an expression locally, without needing access to the information attached to surrounding binders. In [JPJ08] this is known as the *uniqueness of types* property.

4.2.2 Super-kinds

Starting at the top of the strata, we use super-kinds to classify witness kind constructors, and to ensure that they are applied to the right kind of type. For our three baked-in constructors we have the following super-kinds:

$$\begin{array}{lll} \mathit{Const} & :: & \% \to \Diamond \\ \mathit{Mutable} & :: & \% \to \Diamond \\ \mathit{Pure} & :: & ! \to \Diamond \end{array}$$

The first signature says that *Const* may only be applied to a region type, such as with $\mathit{Const}\ r_1$. Applying it to a type of differing kind would not make sense, such as with $\mathit{Const}\ (\mathit{Read}\ r_1)$. We always use $\Diamond$, pronounced "prop", as the result of witness kind constructors. A signature such as $\mathit{Const} :: \% \to \Diamond$ can be read "a witness classified by *Const* guarantees a property of a region". We use $\Box$, pronounced "box", as the super-kind for kinds that do not encode such a property, such as $\%$ and $* \to *$.

4.2.3 Kinds

As our system uses dependent kinds (types in kinds) we have the application and abstraction forms $\Pi(a :: \kappa_1).\ \kappa_2$ and $\kappa\ \varphi$. The function kind $\kappa_1 \to \kappa_2$ is encoded as a sugared form of $\Pi(\_ :: \kappa_1).\ \kappa_2$, where the underscore indicates that the type variable is not present in κ_2 and can be safely ignored. The symbols $*$, $\%$ and $!$ give the kinds of types, regions and effects respectively. The witness kind constructors *Const*, *Mutable* and *Pure* are used to record the particular program property that a type level witness guarantees.

4.2.4 Types

We use φ to range over all type-level information including value types, region variables, effects and witnesses. When we wish to be more specific we use τ, σ and δ to refer to value types, effect types and witness types respectively.

Note that $\top$ and $\bot$ are effect types, and $\vee$ is only applied to effect types.

The types that are underlined, $\underline{\text{mutable}\ \varphi}$, $\underline{\text{const}\ \varphi}$, $\underline{\text{pure}\ \sigma}$ and $\underline{\rho}$ are "operational" witnesses and do not appear in the source program. They are constructed by the evaluation of a witness constructor, and we arrange the typing rules so that their construction requires the heap to possess the associated property. The first three we have seen already, and we will discuss region handles $\underline{\rho}$ in §4.2.7. We use Δ to refer to witnesses.

Note that although our operational semantics manipulates witness terms, they are not actually needed at runtime. We use witnesses to reason about how our system works, and to track information about the program during optimisation, but they can be erased before code generation, along with all other type information.

4.2.5 Terms

The majority of the term language is standard. Our **let** $x = t_1$ **in** t_2 term is not recursive, but we include it to make the examples easier to write. The addition of a **letrec** form can be done via **fix** in the usual way [Pie02, §11.11] but we do not discuss it here. As per §2.3.3 we write our (non-monadic) **do** expressions as a sugared version of **let**.

The term **letregion** r **with** $\{\overline{w_i = \delta_i}\}$ **in** t introduces a new region variable r which is in scope in both the witness bindings $\overline{w_i = \delta_i}$ and the body t. The witness bindings are used to set the properties of the region being created, and introduce witnesses to those properties at the same time. If the region has no specific properties then we include no bindings and write **letregion** r **in** t.

The use of **letregion** imposes some syntactic constraints on the program. These ensure that conflicting region witnesses cannot be created:

Well-formedness of region witnesses: In the list of witness bindings $\overline{w_i = \delta_i}$, each δ_i must be either *MkConst* r or *MkMutable* r, and the list may not mention both. In our full core language we also use the witnesses *MkDirect* r and *MkLazy* r from §2.3.12, and these are also mutually exclusive.

Requiring such witnesses to be mutually exclusive rejects obviously broken terms such as:

$$\textbf{letregion}\ r\ \textbf{with}\ \{w_1 = \mathit{MkConst}\ r,\ w_2 = \mathit{MkMutable}\ r\}\ \textbf{in}\ \ldots$$

Uniqueness of region variables: In all terms **letregion** r **with** $\{\overline{w_i = \delta_i}\}$ **in** t in the initial program, each bound region variable r must be distinct.

This constraint ensures that conflicting witnesses cannot be created in separate **letregion** terms. For example:

$$\begin{aligned}&\textbf{letregion}\ r\ \textbf{with}\ \{w_1 = \mathit{MkConst}\ r\}\ \textbf{in}\\ &\textbf{letregion}\ r\ \textbf{with}\ \{w_2 = \mathit{MkMutable}\ r\}\ \textbf{in}\ \ldots\end{aligned}$$

In an implementation this is easily satisfied by giving variables unique identifiers.

No fabricated region witnesses: Region witnesses constructors may not be used in a type applied directly to a term.

This constraint ensures that conflicting witnesses cannot be created in other parts of the program. For example:

$$\begin{aligned}&\textbf{letregion}\ r\ \textbf{with}\ \{w_1 = \mathit{MkConst}\ r\}\ \textbf{in}\\ &\ldots\ \mathit{update}\ (\mathit{MkMutable}\ r)\ t_1\ t_2\ \ldots\end{aligned}$$

Returning to the list of terms, *True* φ and *False* φ allocate a new boolean value into region φ in the heap. We use the general symbol φ to represent the region as it may be either a region variable r or a region witness $\underline{\rho}$ during evaluation.

The *update* function overwrites its first boolean argument with the value from the second, and requires a witness that its first argument is in a mutable region.

The *suspend* function suspends the application of a function to an argument, and requires a witness that the function is pure.

Store locations, l, are created during the evaluation of a *True* φ or *False* φ term. They can be thought of as the abstract addresses where a particular boolean object lies. When we manipulate store locations in the program we write them as $\underline{l}$, and treat them as (value level) witnesses that a particular store location exists. Akin to region witnesses, store locations are not present in the initial program.

4.2.6 Weak values and lazy evaluation

Values are terms of the form x, $\Lambda(a : \kappa).t$, $\lambda(x : \tau).t$ or $\underline{l}$. We use v to refer to terms that are values.

Additionally, *weak values* are all values, plus terms of the form *suspend* $\delta\ t_1\ t_2$. The latter term is a thunk, which delays the evaluation of the embedded function application. As such, t_1 is the function, t_2 is its argument, and δ is a witness that the application is pure.

When we perform function application the function argument is reduced to a weak value only. When reducing a let-expression, the right of the binding is also reduced to a weak value only. We use v° to refer to weak values, and imagine the circle in the superscript as a bubble that can carry an unapplied function application through the reduction, *sans* evaluation. A weak value is only forced when the surrounding expression demands its (strong) value. This happens when the value is inspected by an if-expression, or is needed by a primitive operator such as *update*.

Here is an example, where the term being reduced at each step is underlined.

$$
\begin{array}{lll}
 & \textbf{let}\ \ z\ = \underline{(\lambda x.\ x)\ \ cat}\ \ \textbf{in} & \\
 & \textbf{let}\ \ f\ = \mathit{suspend}\ (\lambda x.\lambda y.\ x)\ \ (\mathit{suspend}\ (\lambda x.\ x)\ \ \mathit{dog}) & \\
 & \textbf{in}\ \ \ f\ \ z & (1) \\
\longrightarrow & \underline{\textbf{let}}\ \ z\ = \mathit{cat}\ \ \textbf{in} & \\
 & \textbf{let}\ \ f\ = \mathit{suspend}\ (\lambda x.\lambda y.\ x)\ \ (\mathit{suspend}\ (\lambda x.\ x)\ \ \mathit{dog}) & \\
 & \textbf{in}\ \ \ f\ \ z & (2) \\
\longrightarrow & \underline{\textbf{let}}\ \ f\ = \mathit{suspend}\ (\lambda x.\lambda y.\ x)\ \ (\mathit{suspend}\ (\lambda x.\ x)\ \ \mathit{dog}) & \\
 & \textbf{in}\ \ \ f\ \ \mathit{cat} & (3) \\
\longrightarrow & \underline{(\mathit{suspend}\ (\lambda x.\lambda y.\ x)\ \ (\mathit{suspend}\ (\lambda x.\ x)\ \ \mathit{dog}))}\ \ \mathit{cat} & (4) \\
\longrightarrow & \underline{(\lambda y.\ \ \mathit{suspend}\ (\lambda x.\ x)\, \mathit{dog})\ \ \mathit{cat}} & (5) \\
\longrightarrow & \underline{(\mathit{suspend}\ (\lambda x.\ x)\ \ \mathit{dog})} & (6) \\
\longrightarrow & \mathit{dog} & (7)
\end{array}
$$

Note that in the step from (3) to (4), the right of the f binding is a thunk, so it is substituted directly into the body of the let-expression. In the step from (4) to (5), the function application demands the value of the function, so the thunk that represents it is forced.

4.2.7 Stores, machine states and region handles

We model the store as a set of bindings of the form $l \overset{\rho}{\mapsto} V$, where l is a location, V is an atomic value, and ρ is the handle of the region the value is in. As we have limited ourselves to boolean as the only updatable type, only booleans values are present in the store. This means V can be either T or F for true or false values respectively.

Stores may also contain elements of the form (mutable ρ) and (const ρ) which specify the associated property of the region with that handle. Note the distinction between properties and witnesses. Properties exist in the store and are not underlined, whereas witnesses exist in the expression being reduced, and are underlined. To convert a witness to its equivalent property we use the propOf function.

For example:

$$\text{propOf}(\underline{\text{mutable } \rho}) \equiv \text{mutable } \rho$$

A *machine state* is a combination of a store and the term being evaluated. We write machine states as H ; t, where H is the store (also known as the heap) and t is the term. When the program starts evaluating the store is empty, so we use $\emptyset$; t as the *initial state*.

Region witnesses are witnesses to the fact that a particular region is present in the store, and is available to have things allocated into it. Note that the region witnesses in the *store* are written ρ, but when used as a type-level witness they are written with an underline, $\underline{\rho}$. We use **letregion** to create a new region, and the *True* and *False* constructors to allocate values into it. We must pass a region handle to these constructors to prove that the required region exists for them to allocate their value into.

For example, to create a new region and allocate a *True* value into it we could start with the following machine state:

$$\emptyset \ ; \ \textbf{letregion}\ r\ \textbf{with}\ \{w = \mathit{MkConst}\ \ r\}\ \textbf{in}\ \ \mathit{True}\ \ r$$

To reduce the **letregion**, we create fresh region handle ρ, along with its witness $\underline{\rho}$ and substitute the witness for all occurrences of the bound region variable r. If there are witnesses to mutability or constancy attached, then we also construct those and add them to the heap. For example:

$$\begin{array}{lll} & \emptyset & ; \ \textbf{letregion}\ r\ \textbf{with}\ \{w = \mathit{MkConst}\ \ r\}\ \textbf{in}\ \ \mathit{True}\ \ r \\ \longrightarrow & \rho,\ \text{const}\ \rho & ; \ \mathit{True}\ \ \underline{\rho} \end{array}$$

Note that the term *True* $\underline{\rho}$ is closed. If we had not substituted the witness $\underline{\rho}$ for r then r would be free, which would violate the progress theorem we discuss in §4.2.19.

To reduce the application of a data constructor, we create a fresh location in the store and bind the associated value to it:

$$\begin{array}{lll} & \rho,\ \text{const}\ \rho & ;\ \mathit{True}\ \underline{\rho} \\ \longrightarrow & \rho,\ \text{const}\ \rho,\ l \overset{\rho}{\mapsto} T & ;\ \underline{l} \end{array}$$

Note again that the location in the store is written l, but in the term it is written $\underline{l}$. We can think of $\underline{l}$ as a value level witness, or evidence, that there is an associated location in the store. This must be true, because the only way can acquire an $\underline{l}$ is by performing an allocation.

Importantly, in a concrete implementation there is no need to actually record properties like ρ and (const ρ) in the store. A term such as (const ρ) expresses a property which the running program will honor, but we do not need privilege bits, tables, locks, or other low level machinery to achieve this – it's taken care of statically by the type system. The fact a well typed program will not update data in a constant region is part of the guarantee that it will not "go wrong".

On the other hand, we do need to record bindings such as $l \overset{\rho}{\mapsto} T$, because they correspond to physical data in the store.

4.2.8 Region allocation versus lazy evaluation

Note that **letregion** example from the last section would be invalid in systems such as [TB98] which use regions for memory management. Here it is again:

$$\emptyset\ ;\ \textbf{letregion}\ r\ \textbf{with}\ \{w = \mathit{MkConst}\ \ r\}\ \textbf{in}\ \ \mathit{True}\ \ r$$

This expression has type $\mathit{Bool}\ \ r$, which indicates that it returns a boolean value in a region named r. The trouble is that the value will exist in the store after the term has finished evaluating. Systems such as [TB98] use the syntactic scope of the variable bound by a **letregion** to denote the lifetime of the associated region. In these systems, once the body of a **letregion** term has finished evaluating, the region named r, along with all the objects in it, is reclaimed by the storage manager. The type checker ensures that the surrounding program cannot hold references to objects in reclaimed regions, by requiring that the region variable r is not free in the type environment, or the type of the return value. This is an observation criteria similar to the one discussed in §2.4.5.

Unfortunately, this simple criteria only works for strict languages. In Disciple, even though a value may have type $\mathit{Bool}\ \ r$, if it is a lazy value then it may be represented by a thunk. This thunk can hold references to regions that are not visible in its type, and if we were to deallocate those regions before forcing the thunk, then the result would be undefined. This is discussed further in §5.2.4.

As we do not use regions for allocation, we do not enforce the observation criteria mentioned above. However, this requires us to relax our notion of type equality to account for the fact that region handles are substituted for region variables during evaluation. We use the notion of *region similarity*, written $r \sim \rho$ to represent this, and the mechanism is discussed in the coming sections.

4.2.9 Store typings

A *store typing* Σ is a set of elements of the form: $\underline{\rho}$, $\underline{\text{mutable } \rho}$, $\underline{\text{const } \rho}$, $l : \tau$, $r \sim \rho$.

The store typing is an abstract model of the current state of the store, and the properties we require it to have.

A store H is said to be *well typed* with respect to a store typing Σ, written $\Sigma \vdash \text{H}$, if every binding in the store has the type predicted by the store typing. That is:

for all $l \overset{\rho}{\mapsto} V \in \text{H}$ we have
$l : \tau \in \Sigma$ and $\underline{\rho} \in \Sigma$ for some τ, ρ.

for all mutable $\rho \in \text{H}$ we have $\underline{\text{mutable } \rho} \in \Sigma$

for all const $\rho \in \text{H}$ we have $\underline{\text{const } \rho} \in \Sigma$

The dual of well typed is *models*, that is a store typing Σ is said to *model* a store H, written $\Sigma \models \text{H}$, if all members in the store typing correspond to members in the store:

for all $l : \tau \in \Sigma$ we have $l \overset{\rho}{\mapsto} V \in \text{H}$

for all $\underline{\rho} \in \Sigma$ we have $\rho \in \text{H}$

for all $\underline{\text{mutable } \rho} \in \Sigma$ we have mutable $\rho \in \text{H}$

for all $\underline{\text{const } \rho} \in \Sigma$ we have const $\rho \in \text{H}$

4.2.10 Region similarity

The term $r \sim \rho$, pronounced "r is similar to ρ", is used to associate a region handle with a region variable. This notation is used in our proof of soundness to account for the fact that the types of terms change during evaluation.

Our typing rules use the following judgement form:

$$\boxed{\Gamma \mid \Sigma \vdash t :: \tau ; \sigma}$$

This is read: with type environment Γ and store typing Σ the term t has type τ and its evaluation causes an effect σ. The type environment maps value variables to types, and type variables to kinds.

When a **letregion** is reduced, the act of substituting the fresh region handle for the region variable changes the type of the term. We can see this in the following reduction from §4.2.7.

$$\begin{array}{lll} & \emptyset & ; \ \textbf{letregion}\ r\ \textbf{with}\ \{w = \mathit{MkConst}\ \ r\}\ \textbf{in}\ \ \mathit{True}\ \ r \\ \longrightarrow & \rho,\ \text{const}\ \rho & ;\ \mathit{True}\ \ \underline{\rho} \end{array}$$

Writing each position out on a separate line, the type judgement for the initial term is:

$$\begin{array}{ll} & \emptyset \\ \mid & \emptyset \\ \vdash & \textbf{letregion}\ r\ \textbf{with}\ \{w = \mathit{MkConst}\ \ r\}\ \textbf{in}\ \ \mathit{True}\ \ r \\ :: & \mathit{Bool}\ \ r \\ ; & \bot \end{array}$$

Note that the $\mathit{True}\ \ r$ term gives rise to $\mathit{Bool}\ \ r$. However, when we reduce the outer **letregion**, we end up with:

$$\begin{array}{ll} & \emptyset \\ \mid & \underline{\rho},\ \underline{\text{const}\ \rho},\ r \sim \rho \\ \vdash & \mathit{True}\ \ \underline{\rho} \\ :: & \mathit{Bool}\ \ \underline{\rho} \\ ; & \bot \end{array}$$

As the value term is now $\mathit{True}\ \ \underline{\rho}$ instead of $\mathit{True}\ \ r$, its type is $\mathit{Bool}\ \ \underline{\rho}$ instead of $\mathit{Bool}\ \ r$. This is why we introduce the $r \sim \rho$ term into the store typing: it records the mapping between region handles and the variables they were substituted for. In our proof of soundness we require that when we reduce an expression, the result has a type that is *similar to* the initial expression. That is, it is identical up to the renaming of region handles to their associated region variables. Note that the effect term can also change during reduction, with region variables in effects like $\mathit{Read}\ \ r$ being replaced by region handles.

4.2.11 Duplication of region variables during evaluation

Before moving on to discuss the formal typing rules, we point out a final property of the store typing. There can be multiple region handles bound to a particular region variable, that is, we can have both $r \sim \rho_1$ and $r \sim \rho_2$ in the store typing, where ρ_1 and ρ_2 are distinct. This is caused when a term containing a **letregion** is duplicated during function application (or via a let-binding).

For example, starting with the statement:

$$
\begin{array}{lll}
 & f : \ldots & \\
| & \emptyset & \\
\vdash & (\lambda x : () \xrightarrow{\perp} \mathit{Bool}\ r.\ f\ (x\ ())\ (x\ ())) & \\
 & \qquad (\lambda y : ().\ \textbf{letregion}\ r\ \textbf{in}\ \mathit{True}\ r) & \\
:: & \ldots & \\
; & \perp &
\end{array}
$$

We substitute the argument for x, giving:

$$
\begin{array}{lll}
\longrightarrow & f : \ldots & \\
 & | \quad \emptyset & \\
 & \vdash \quad f \quad ((\lambda(y\ :\ ()).\ \textbf{letregion}\ r\ \textbf{in}\ \mathit{True}\ r)\ ()) & \\
 & \qquad\quad ((\lambda(y\ :\ ()).\ \textbf{letregion}\ r\ \textbf{in}\ \mathit{True}\ r)\ ()) & \\
 & :: \quad \ldots & \\
 & ; \quad \perp &
\end{array}
$$

Note the duplication of r in the **letregion** term. When the first copy is reduced it creates its own region handle:

$$
\begin{array}{lll}
\xrightarrow{*} & f : \ldots & \\
 & | \quad \underline{\rho_1},\ r \sim \rho_1,\ l_1 \overset{\rho_1}{\mapsto} \mathit{Bool}\ \underline{\rho_1} & \\
 & \vdash \quad f\ \underline{l_1}\ ((\lambda(y\ :\ ()).\ \textbf{letregion}\ r\ \textbf{in}\ \mathit{True}\ r)\ ()) & \\
 & :: \quad \ldots & \\
 & ; \quad \perp &
\end{array}
$$

Reducing the second copy produces a different one, ρ_2:

$$
\begin{array}{lll}
\xrightarrow{*} & f : \ldots & \\
 & | \quad \underline{\rho_1},\ r \sim \rho_1,\ l_1 \overset{\rho_1}{\mapsto} \mathit{Bool}\ \underline{\rho_1} & \\
 & \qquad \underline{\rho_2},\ r \sim \rho_2,\ l_2 \overset{\rho_2}{\mapsto} \mathit{Bool}\ \underline{\rho_2} & \\
 & \vdash \quad f\ \underline{l_1}\ \underline{l_2} & \\
 & :: \quad \ldots & \\
 & ; \quad \perp &
\end{array}
$$

This illustrates that the mapping between region variables and region handles is not simply one-to-one, a point we must be mindful of in our proof of soundness.

4.2.12 Witness production

$$\boxed{\mathrm{H} \;;\; \delta \rightsquigarrow \delta'}$$

$$\mathrm{H}, \mathrm{const}\ \rho \;;\; \mathit{MkConst}\ \underline{\rho} \rightsquigarrow \underline{\mathrm{const}\ \rho} \qquad \text{(EwConst)}$$

$$\mathrm{H}, \mathrm{mutable}\ \rho \;;\; \mathit{MkMutable}\ \underline{\rho} \rightsquigarrow \underline{\mathrm{mutable}\ \rho} \qquad \text{(EwMutable)}$$

$$\mathrm{H} \;;\; \mathit{MkPure} \rightsquigarrow \underline{\mathrm{pure}\ \bot} \qquad \text{(EwPure)}$$

$$\mathrm{H} \;;\; \mathit{MkPurify}\ \underline{\rho}\ (\underline{\mathrm{const}\ \rho}) \rightsquigarrow \underline{\mathrm{pure}\ (\mathit{Read}\ \rho)} \qquad \text{(EwPurify)}$$

$$\frac{\mathrm{H} \;;\; \delta_1 \rightsquigarrow \delta_1'}{\mathrm{H} \;;\; \mathit{MkPureJoin}\ \sigma_1\ \sigma_2\ \delta_1\ \delta_2 \rightsquigarrow \mathit{MkPureJoin}\ \sigma_1\ \sigma_2\ \delta_1'\ \delta_2} \qquad \text{(EwPureJoin1)}$$

$$\frac{\mathrm{H} \;;\; \delta_2 \rightsquigarrow \delta_2'}{\mathrm{H} \;;\; \mathit{MkPureJoin}\ \sigma_1\ \sigma_2\ \delta_1\ \delta_2 \rightsquigarrow \mathit{MkPureJoin}\ \sigma_1\ \sigma_2\ \delta_1\ \delta_2'} \qquad \text{(EwPureJoin2)}$$

$$\mathrm{H} \;;\; \mathit{MkPureJoin}\ \sigma_1\ \sigma_2\ \underline{\mathrm{pure}\ \sigma_1}\ \underline{\mathrm{pure}\ \sigma_2} \rightsquigarrow \underline{\mathrm{pure}\ (\sigma_1 \vee \sigma_2)} \qquad \text{(EwPureJoin3)}$$

The judgement form $\mathrm{H} \;;\; \delta \rightsquigarrow \delta'$ reads: with store H, type δ produces type δ'.

The first two rules, EwConst and EwMutable are used to sample a particular property of the store. The idea is that a term like *MkMutable* ρ cannot be reduced to a witness to that fact, unless the store really does support the required property. When proving soundness, we show that such a term can only be evaluated in a context that ensures the required property is true, so the evaluation can always progress.

EwPure is a simple axiom allows us to construct a witness that the $\bot$ effect is pure.

EwPurify is used to produce a witness that a read from a region is pure from a witness that the region is constant.

The final three rules, EwPureJoin1, EwPureJoin2 and EwPureJoin are used to join two witnesses, showing the purity of seperate effects, into one that shows the purity of both.

4.2.13 Transitions

$$\boxed{\mathrm{H} \;;\; t \longrightarrow \mathrm{H}' \;;\; t'}$$

$$\frac{\mathrm{H} \;;\; t \longrightarrow \mathrm{H}' \;;\; t'}{\mathrm{H} \;;\; t\ \varphi \longrightarrow \mathrm{H}' \;;\; t'\ \varphi} \qquad \text{(EvTApp1)}$$

$$\mathrm{H} \;;\; (\Lambda(a :: \kappa).\ t)\ \varphi \longrightarrow \mathrm{H} \;;\; t[\varphi/a] \qquad \text{(EvTAppAbs)}$$

$$\frac{\mathrm{H} \;;\; t_1 \longrightarrow \mathrm{H}' \;;\; t_1'}{\mathrm{H} \;;\; t_1\ t_2 \longrightarrow \mathrm{H}' \;;\; t_1'\ t_2} \qquad \text{(EvApp1)}$$

$$\frac{\mathrm{H} \;;\; t \longrightarrow \mathrm{H}' \;;\; t'}{\mathrm{H} \;;\; v\ t \longrightarrow \mathrm{H}' \;;\; v\ t'} \qquad \text{(EvApp2)}$$

$$\mathrm{H} \;;\; (\lambda(x :: \tau).\ t)\ v^\circ \longrightarrow \mathrm{H} \;;\; t[v^\circ/x] \qquad \text{(EvAppAbs)}$$

$$\frac{\mathrm{H} \;;\; t_1 \longrightarrow \mathrm{H}' \;;\; t_1'}{\mathrm{H} \;;\; \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \longrightarrow \mathrm{H}' \;;\; \mathbf{let}\ x = t_1'\ \mathbf{in}\ t_2} \qquad \text{(EvLet1)}$$

$$\mathrm{H} \;;\; \mathbf{let}\ x = v^\circ\ \mathbf{in}\ t \longrightarrow \mathrm{H} \;;\; t[v^\circ/x] \qquad \text{(EvLet)}$$

$$\frac{\mathrm{H},\ \overline{\mathrm{propOf}(\Delta_i)} \;;\; \overline{\delta_i \rightsquigarrow \Delta_i} \qquad \rho \text{ fresh}}{\begin{array}{l}\mathrm{H} \;;\; \mathbf{letregion}\ r\ \{\overline{w_i = \delta_i}\}\ \mathbf{in}\ t \\ \qquad \longrightarrow \mathrm{H},\ \rho,\ \overline{\mathrm{propOf}(\Delta_i)} \;;\; t[\overline{\Delta_i/w_i}][\underline{\rho}/r]\end{array}} \qquad \text{(EvLetRegion)}$$

$$\frac{\mathrm{H} \;;\; t_1 \longrightarrow \mathrm{H}' \;;\; t_1'}{\mathrm{H} \;;\; \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{then}\ t_3 \longrightarrow \mathrm{H}' \;;\; \mathbf{if}\ t_1'\ \mathbf{then}\ t_2\ \mathbf{then}\ t_3} \qquad \text{(EvIf)}$$

$$\mathrm{H},\ l \overset{\rho}{\mapsto} \mathrm{T} \;;\; \mathbf{if}\ \underline{l}\ \mathbf{then}\ t_2\ \mathbf{then}\ t_3 \longrightarrow \mathrm{H},\ l \overset{\rho}{\mapsto} \mathrm{T} \;;\; t_2 \qquad \text{(EvIfThen)}$$

$$\mathrm{H},\ l \overset{\rho}{\mapsto} \mathrm{F} \;;\; \mathbf{if}\ \underline{l}\ \mathbf{then}\ t_2\ \mathbf{then}\ t_3 \longrightarrow \mathrm{H},\ l \overset{\rho}{\mapsto} \mathrm{F} \;;\; t_3 \qquad \text{(EvIfElse)}$$

$$\frac{l \text{ fresh}}{\mathrm{H},\ \rho \;;\; \mathit{True}\ \underline{\rho} \longrightarrow \mathrm{H},\ \rho,\ l \overset{\rho}{\mapsto} \mathrm{T} \;;\; \underline{l}} \qquad \text{(EvTrue)}$$

$$\frac{l \text{ fresh}}{\mathrm{H},\ \rho \;;\; \mathit{False}\ \underline{\rho} \longrightarrow \mathrm{H},\ \rho,\ l \overset{\rho}{\mapsto} \mathrm{F} \;;\; \underline{l}} \qquad \text{(EvFalse)}$$

$$\frac{\mathrm{H} \;;\; t_1 \longrightarrow \mathrm{H}' \;;\; t_1'}{\mathrm{H} \;;\; \mathit{update}\ \Delta\ t_1\ t_2 \longrightarrow \mathrm{H}' \;;\; \mathit{update}\ \Delta\ t_1'\ t_2} \qquad \text{(EvUpdate1)}$$

$$\frac{\mathrm{H} \;;\; t \longrightarrow \mathrm{H}' \;;\; t'}{\mathrm{H} \;;\; \mathit{update}\ \Delta\ v\ t \longrightarrow \mathrm{H}' \;;\; \mathit{update}\ \Delta\ v\ t'} \qquad \text{(EvUpdate2)}$$

$$\begin{array}{l} \mathrm{H},\ \mathrm{mutable}\ \rho_1,\ l_1 \overset{\rho_1}{\mapsto} V_1,\ l_2 \overset{\rho_2}{\mapsto} V_2\ ;\ \mathit{update}\ \underline{\mathrm{mutable}\ \rho_1}\ \underline{l_1}\ \underline{l_2} \\ \longrightarrow \mathrm{H},\ \mathrm{mutable}\ \rho_1,\ l_1 \overset{\rho_1}{\mapsto} V_2,\ l_2 \overset{\rho_2}{\mapsto} V_2\ ;\ () \end{array} \tag{EvUpdate3}$$

$$\frac{\mathrm{H}\ ;\ \delta \rightsquigarrow \delta'}{\mathrm{H}\ ;\ \mathit{suspend}\ \ \delta\ t_1\ t_2 \longrightarrow \mathrm{H}\ ;\ \mathit{suspend}\ \ \delta'\ t_1\ t_2} \tag{EvSuspend1}$$

$$\frac{\mathrm{H}\ ;\ t_1 \longrightarrow \mathrm{H}\ ;\ t_1'}{\mathrm{H}\ ;\ \mathit{suspend}\ \ \Delta\ t_1\ t_2 \longrightarrow \mathrm{H}\ ;\ \mathit{suspend}\ \ \Delta\ t_1'\ t_2} \tag{EvSuspend2}$$

$$\frac{\mathrm{H}\ ;\ t \longrightarrow \mathrm{H}\ ;\ t'}{\mathrm{H}\ ;\ \mathit{suspend}\ \ \Delta\ v\ t \longrightarrow \mathrm{H}\ ;\ \mathit{suspend}\ \ \Delta\ v\ t'} \tag{EvSuspend3}$$

$$\mathrm{H}\ ;\ \mathit{suspend}\ \ \underline{\mathrm{pure}\ \sigma_1}\ (\lambda(x:\tau).\ t)\ v^\circ \longrightarrow \mathrm{H}\ ;\ t[v^\circ/x] \tag{EvSuspend4}$$

Rules EvTApp1 - EvLet are standard.

EvLetRegion creates a new region in the store, and substitutes its region handle into the value term. By inspection of the witness production rules, the statement $H,\ \overline{\mathrm{propOf}(\Delta_i)}\ ;\ \overline{\delta_i \rightsquigarrow \Delta_i}$ is always true. It says that if we place the required properties in the heap, we can then construct witnesses that sample these properties.

EvIf - EvIfElse are standard.

EvTrue - EvFalse show how to allocate new boolean values into the store. Note that to allocate a new value, the region it is to be allocated in must already exist in the store. In the proof of progress we show that if a term contains a region witness $\underline{\rho}$ then the corresponding region will always be present.

EvUpdate1 - EvUpdate3 show how to update a boolean value in the store.

EvSuspend1 - EvSuspend4 handle the suspension of function applications. In EvSuspend1 we include the statement $\mathrm{H}\ ;\ \delta \rightsquigarrow \delta'$ to allow for the evaluation of witness production rules, such as EwPureJoin. Note that in EvSuspend, the effect term σ_1 is only mentioned in the witness term. Our typing rules ensure that σ_1 actually represents the effect of evaluating the term t.

4.2.14 Super-kinds of kinds

$$\boxed{\Gamma \mid \Sigma \vdash_{\mathrm{K}} \kappa \;::\; \omega}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{K}} \kappa_1 \;::\; \omega_1 \qquad \Gamma,\ a : \kappa_1 \mid \Sigma \vdash_{\mathrm{K}} \kappa_2 \;::\; \omega_2}{\Gamma \mid \Sigma \vdash_{\mathrm{K}} \Pi(a : \kappa_1).\ \kappa_2 \;::\; \omega_2} \tag{KsAbs}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{K}} \kappa_1 \;::\; \kappa_{11} \to \omega \qquad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi \;::\; \kappa_{11}}{\Gamma \mid \Sigma \vdash_{\mathrm{K}} \kappa_1\ \varphi \;::\; \omega} \tag{KsApp}$$

$$\frac{\kappa \in \{*,\ \%,\ !\ \}}{\Gamma \mid \Sigma \vdash_{\mathrm{K}} \kappa \;::\; \Box} \tag{KsAtom}$$

$$\begin{array}{lll} \Gamma \mid \Sigma \vdash_{\mathrm{K}} & \mathit{Const} & :: \% \to \Diamond \\ \Gamma \mid \Sigma \vdash_{\mathrm{K}} & \mathit{Mutable} & :: \% \to \Diamond \\ \Gamma \mid \Sigma \vdash_{\mathrm{K}} & \mathit{Pure} & :: ! \to \Diamond \end{array}$$

The judgement form $\Gamma \mid \Sigma \vdash_{\mathrm{K}} \kappa :: \omega$ reads: with environment Γ and store typing Σ, kind κ has super-kind ω.

KsAbs is the rule for the dependent kind abstraction. Note that a kind signature such as $\% \to *$ is also desugared to this form, resulting in $\Pi(\_ : \%).\ *$. Although ω_1 is only mentioned once in this rule, inclusion of the $\Gamma \mid \Sigma \vdash_{\mathrm{K}} \kappa_1 :: \omega_1$ premise ensures that the kind κ_1 is well formed.

KsApp is the application rule for super-kinds. As we (thankfully) do not need higher-order super kinds, the expression on the left of the super-kind arrow can always be a (non-super) kind.

KsAtom says that the super-kind of atomic kinds is always $\Box$.

The last three rules give super-kinds for the witness kind constructors. These allow us to check for malformed kind expressions such as *Pure* (*Bool* a) and *Const Read*.

4.2.15 Kinds of types

$$\boxed{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi \;::\; \kappa}$$

$$\frac{a : \kappa \in \Gamma}{\Gamma \mid \Sigma \vdash_{\mathrm{T}} a \;::\; \kappa} \quad \text{(KiVar)}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{K}} \kappa_1 \;::\; \omega_1 \qquad \Gamma,\ a : \kappa_1 \mid \Sigma \vdash_{\mathrm{T}} \tau_2 \;::\; *}{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \forall (a : \kappa_1).\ \tau_2 \;::\; *} \quad \text{(KiAll)}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_1 \;::\; \Pi (a : \kappa_1).\ \kappa_2 \qquad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_2 \;::\; \kappa_1}{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_1\ \varphi_2 \;::\; \kappa_2[\varphi_2/a]} \quad \text{(KiApp)}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \sigma_1 \;::\; ! \qquad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \sigma_2 \;::\; !}{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \sigma_1 \vee \sigma_2 \;::\; !} \quad \text{(KiJoin)}$$

$$\Gamma \mid \Sigma \vdash_{\mathrm{T}} \top \;::\; ! \quad \text{(KiTop)}$$

$$\Gamma \mid \Sigma \vdash_{\mathrm{T}} \bot \;::\; ! \quad \text{(KiBot)}$$

$$\frac{\underline{\rho} \in \Sigma}{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \underline{\rho} \;::\; \%} \quad \text{(KiHandle)}$$

$$\frac{\underline{\text{mutable}\ \rho} \in \Sigma}{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \underline{\text{mutable}\ \rho} \;::\; \mathit{Mutable}\ \underline{\rho}} \quad \text{(KiMutable)}$$

$$\frac{\underline{\text{const}\ \rho} \in \Sigma}{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \underline{\text{const}\ \rho} \;::\; \mathit{Const}\ \underline{\rho}} \quad \text{(KiConst)}$$

$$\Gamma \mid \Sigma \vdash_{\mathrm{T}} \underline{\text{pure}\ \bot} \;::\; \mathit{Pure}\ \bot \quad \text{(KiPure)}$$

$$\frac{\underline{\text{const}\ \rho} \in \Sigma}{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \underline{\text{pure}\ (\mathit{Read}\ \rho)} \;::\; \mathit{Pure}\ (\mathit{Read}\ \underline{\rho})} \quad \text{(KiPurify)}$$

$$\frac{\begin{array}{c}\Gamma \mid \Sigma \vdash_{\mathrm{T}} \underline{\text{pure}\ \sigma_1} \;::\; \mathit{Pure}\ \sigma_1 \\ \Gamma \mid \Sigma \vdash_{\mathrm{T}} \underline{\text{pure}\ \sigma_2} \;::\; \mathit{Pure}\ \sigma_2\end{array}}{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \underline{\text{pure}\ (\sigma_1 \vee \sigma_2)} \;::\; \mathit{Pure}\ (\sigma_1 \vee \sigma_2)} \quad \text{(KiPureJoin)}$$

$$
\begin{array}{llll}
\Gamma \mid \Sigma \vdash_{\mathrm{T}} & () & :: & * \\
\Gamma \mid \Sigma \vdash_{\mathrm{T}} & (\rightarrow) & :: & * \rightarrow * \rightarrow ! \rightarrow * \\
\Gamma \mid \Sigma \vdash_{\mathrm{T}} & \mathit{Bool} & :: & \% \rightarrow * \\
\Gamma \mid \Sigma \vdash_{\mathrm{T}} & \mathit{Read} & :: & \% \rightarrow ! \\
\Gamma \mid \Sigma \vdash_{\mathrm{T}} & \mathit{Write} & :: & \% \rightarrow ! \\
\Gamma \mid \Sigma \vdash_{\mathrm{T}} & \mathit{MkConst} & :: & \Pi(r : \%).\ \mathit{Const}\ r \\
\Gamma \mid \Sigma \vdash_{\mathrm{T}} & \mathit{MkMutable} & :: & \Pi(r : \%).\ \mathit{Mutable}\ r \\
\Gamma \mid \Sigma \vdash_{\mathrm{T}} & \mathit{MkPure} & :: & \mathit{Pure}\ \bot \\
\Gamma \mid \Sigma \vdash_{\mathrm{T}} & \mathit{MkPurify} & :: & \Pi(r : \%).\ \mathit{Const}\ r \rightarrow\ \mathit{Pure}\ (\mathit{Read}\ r) \\
\Gamma \mid \Sigma \vdash_{\mathrm{T}} & \mathit{MkPureJoin} & :: & \Pi(e_1 : !).\ \Pi(e_2 : !).\ \mathit{Pure}\ e_1 \rightarrow \mathit{Pure}\ e_2 \rightarrow\ \mathit{Pure}\,(e_1 \vee e_2)
\end{array}
$$

The judgement form $\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi :: \kappa$ reads: with environment Γ, and store typing Σ, the type φ has kind κ.

KiVar is standard.

In KiAll is similar to KsAbs, with the premise $\Gamma \mid \Sigma \vdash_{\mathrm{K}} \kappa_1 :: \omega_1$ ensuring that κ_1 is well formed.

KiApp is the rule for type-type application, and the substitution in the conclusion handles our dependent kinds.

KiJoin ensures that both arguments are effects, as the join operator is only defined for types of that kind.

KiTop and KiBot are straightforward.

KiHandle requires all region witnesses present in the term to be present in the store typing. Provided the store typing models the store §4.2.9, this ensures that if a region witness is present in the term, the corresponding region is also present in the store. Likewise, KiMutable and KiConst ensure that the appropriate witnesses are present in the store typing, so the store has the required property.

KiPure and KiPurify relate type-level witnesses of purity with the corresponding kind-level description of that property.

KiPureJoin joins two separate witnesses, each showing the purity of an effect, into a witness of purity of the sum of these effects. KiPureJoin was introduced in §4.1.4.

The remaining rules give the kinds of our type constructors. We could have arranged for these kinds to be present in the initial type environment, but present them as separate rules due to their built-in nature.

4.2.16 Similarity

$$\boxed{\Sigma \vdash_{\mathrm{T}} \kappa \sim \kappa'}$$

$$\Sigma \vdash_{\mathrm{T}} \kappa \sim \kappa \qquad \text{(SkmRefl)}$$

$$\frac{\Sigma \vdash \varphi_1 \sim \varphi_2}{\Sigma \vdash_{\mathrm{T}} \kappa\ \varphi_1 \sim \kappa\ \varphi_2} \qquad \text{(SkmApp)}$$

$$\boxed{\Sigma \vdash \varphi \sim \varphi'}$$

$$\frac{r \sim \rho \in \Sigma}{\Sigma \vdash r \sim \rho} \qquad \text{(SimHandle)}$$

$$\Sigma \vdash \varphi_1 \sim \varphi_2 \qquad \text{(SimRefl)}$$

$$\frac{\Sigma \vdash \varphi_1 \sim \varphi_2 \quad \Sigma \vdash \varphi_2 \sim \varphi_3}{\Sigma \vdash \varphi_1 \sim \varphi_3} \qquad \text{(SimTrans)}$$

$$\frac{\Sigma \vdash \varphi_1 \sim \varphi_2}{\Sigma \vdash \varphi_2 \sim \varphi_1} \qquad \text{(SimCommute)}$$

$$\frac{\Sigma \vdash_{\mathrm{T}} \kappa_1 \sim \kappa_2 \quad \Sigma \vdash \tau_1 \sim \tau_2}{\Sigma \vdash \forall(a : \kappa_1).\, \tau_1 \sim \forall(a : \kappa_2).\, \tau_2} \qquad \text{(SimAll)}$$

$$\frac{\Sigma \vdash \varphi_{11} \sim \varphi_{21} \quad \Sigma \vdash \varphi_{21} \sim \varphi_{22}}{\Sigma \vdash \varphi_{11}\ \varphi_{12} \sim \varphi_{21}\ \varphi_{22}} \qquad \text{(SimApp)}$$

$$\frac{\Sigma \vdash \sigma_{11} \sim \sigma_{21} \quad \Sigma \vdash \sigma_{12} \sim \sigma_{22}}{\Sigma \vdash \sigma_{11} \vee \sigma_{12} \sim \sigma_{21} \vee \sigma_{22}} \qquad \text{(SimJoin)}$$

$$\frac{\Sigma \vdash \varphi_1 \sim \varphi_2}{\Sigma \vdash \underline{\text{mutable}\ \varphi_1} \sim \underline{\text{mutable}\ \varphi_2}} \qquad \text{(SimMutable)}$$

$$\frac{\Sigma \vdash \varphi_1 \sim \varphi_2}{\Sigma \vdash \underline{\text{const}\ \varphi_1} \sim \underline{\text{const}\ \varphi_2}} \qquad \text{(SimConst)}$$

$$\frac{\Sigma \vdash \varphi_1 \sim \varphi_2}{\Sigma \vdash \underline{\text{pure}\ \varphi_1} \sim \underline{\text{pure}\ \varphi_2}} \qquad \text{(SimPure)}$$

The judgement form $\Sigma \vdash_{\mathrm{T}} \kappa \sim \kappa'$ reads: with store typing Σ, kind κ is similar to kind κ'. We also write this as $\kappa \sim_\Sigma \kappa'$. The judgement form $\Sigma \vdash \varphi \sim \varphi'$ is similar.

The rules for similarity should be self explanatory. The only interesting one is SimHandle. This rule says that region variables are similar to their associated region handles, provided the mapping is present in the store typing.

4.2.17 Subsumption

$$\boxed{\Gamma \mid \Sigma \vdash \sigma \sqsubseteq \sigma'}$$

$$\frac{\Sigma \vdash \sigma_1 \sim \sigma_2}{\Gamma \mid \Sigma \vdash \sigma_1 \sqsubseteq \sigma_2} \quad \text{(SubRefl)}$$

$$\frac{\Gamma \mid \Sigma \vdash \sigma_1 \sqsubseteq \sigma_2 \quad \Gamma \mid \Sigma \vdash \sigma_2 \sqsubseteq \sigma_3}{\Gamma \mid \Sigma \vdash \sigma_1 \sqsubseteq \sigma_3} \quad \text{(SubTrans)}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \sigma :: !}{\Gamma \mid \Sigma \vdash \sigma \sqsubseteq \top} \quad \text{(SubTop)}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \sigma :: !}{\Gamma \mid \Sigma \vdash \bot \sqsubseteq \sigma} \quad \text{(SubBot)}$$

$$\frac{\Gamma \mid \Sigma \vdash \sigma_1 \sqsubseteq \sigma_3 \quad \Gamma \mid \Sigma \vdash \sigma_2 \sqsubseteq \sigma_3}{\Gamma \mid \Sigma \vdash \sigma_1 \vee \sigma_2 \sqsubseteq \sigma_3} \quad \text{(SubJoin1)}$$

$$\frac{\Gamma \mid \Sigma \vdash \sigma_1 \sqsubseteq \sigma_2 \quad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \sigma_2 \vee \sigma_3 :: !}{\Gamma \mid \Sigma \vdash \sigma_1 \sqsubseteq \sigma_2 \vee \sigma_3} \quad \text{(SubJoin2)}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \delta :: \mathit{Pure}\ \sigma}{\Gamma \mid \Sigma \vdash \sigma \sqsubseteq \bot} \quad \text{(SubPurify)}$$

The judgement form $\Gamma \vdash \sigma \sqsubseteq \sigma'$ reads: with environment Γ and store typing Σ, effect σ is subsumed by effect σ'.

All but the last of these rules are standard. Note that the type environment Γ is not used in the premises of these rules. We will make use of it when we discuss bounded quantification in §4.3.4.

SubPurify says that if we have a *Pure* σ witness, then we can treat σ as being pure. This rule is the keystone of our system. It allow us to use the information embodied in a witness to reason that the evaluation of an expression with a read effect cannot interfere with others. The rule is used in the TySuspend case when proving preservation of effects under evaluation in Appendix A.

4.2.18 Types of terms

$$\boxed{\Gamma \mid \Sigma \vdash t :: \tau ; \sigma}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \mid \Sigma \vdash x :: \tau ; \bot} \quad \text{(TyVar)}$$

$$\frac{\Gamma,\ a : \kappa \mid \Sigma \vdash t_2 :: \tau_2 ; \sigma_2}{\Gamma \mid \Sigma \vdash \Lambda(a : \kappa).\, t_2 :: \forall(a : \kappa).\, \tau_2 ; \sigma_2} \quad \text{(TyAbsT)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 :: \forall(a : \kappa_{11}).\, \varphi_{12} ; \sigma_1 \quad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_2 :: \kappa_2 \quad \kappa_{11} \sim_\Sigma \kappa_2}{\Gamma \mid \Sigma \vdash t_1\ \varphi_2 :: \varphi_{12}[\varphi_2/a] ; \sigma_1[\varphi_2/a]} \quad \text{(TyAppT)}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \tau_1 :: * \qquad \Gamma,\ x : \tau_1 \mid \Sigma \vdash t :: \tau_2 ; \sigma}{\Gamma \mid \Sigma \vdash \lambda(x : \tau_1).\, t :: \tau_1 \xrightarrow{\sigma} \tau_2 ; \bot} \quad \text{(TyAbs)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_2 :: \tau_2 ; \sigma_2 \quad \Gamma \mid \Sigma \vdash t_1 :: \tau_{11} \xrightarrow{\sigma} \tau_{12} ; \sigma_1 \quad \tau_{11} \sim_\Sigma \tau_2}{\Gamma \mid \Sigma \vdash t_1\ t_2 :: \tau_{12} ; \sigma_1 \vee \sigma_2 \vee \sigma} \quad \text{(TyApp)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 :: \tau_1 ; \sigma_1 \quad \Gamma,\ x : \tau_3 \mid \Sigma \vdash t_2 :: \tau_2 ; \sigma_2 \quad \tau_1 \sim_\Sigma \tau_3}{\Gamma \mid \Sigma \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 :: \tau_2 ; \sigma_1 \vee \sigma_2} \quad \text{(TyLet)}$$

$$\frac{\overline{\delta_i} \text{ well formed} \qquad \Gamma \mid \Sigma \vdash_{\mathrm{K}} \kappa_i :: \Diamond \quad \Gamma,\ r : \%,\ \overline{w_i : \kappa_i} \mid \Sigma \vdash t :: \tau ; \sigma \qquad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \delta_i :: \kappa_i}{\Gamma \mid \Sigma \vdash \mathbf{letregion}\ r\ \mathbf{with}\ \{\overline{w_i = \delta_i}\}\ \mathbf{in}\ t :: \tau ; \sigma} \quad \text{(TyLetRegion)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 :: \mathit{Bool}\ \varphi ; \sigma_1 \quad \Gamma \mid \Sigma \vdash t_2 :: \tau_2 ; \sigma_2 \quad \Gamma \mid \Sigma \vdash t_3 :: \tau_3 ; \sigma_3 \quad \tau_2 \sim_\Sigma \tau_3}{\Gamma \mid \Sigma \vdash \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 :: \tau_2 ; \sigma_1 \vee \sigma_2 \vee \sigma_3 \vee \mathit{Read}\ \varphi} \quad \text{(TyIf)}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi :: \%}{\Gamma \mid \Sigma \vdash \mathit{True}\ \varphi :: \mathit{Bool}\ \varphi ; \bot} \quad \text{(TyTrue)}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi :: \%}{\Gamma \mid \Sigma \vdash \mathit{False}\ \varphi :: \mathit{Bool}\ \varphi ; \bot} \quad \text{(TyFalse)}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \delta :: \mathit{Mutable}\ \varphi_1 \quad \Gamma \mid \Sigma \vdash t_1 :: \mathit{Bool}\ \varphi_1 ; \sigma_1 \quad \Gamma \mid \Sigma \vdash t_2 :: \mathit{Bool}\ \varphi_2 ; \sigma_2}{\Gamma \mid \Sigma \vdash \mathit{update}\ \delta\ t_1\ t_2 :: () ; \sigma_1 \vee \sigma_2 \vee \mathit{Read}\ \varphi_2 \vee \mathit{Write}\ \varphi_1} \quad \text{(TyUpdate)}$$

$$\frac{\tau_{11} \sim_\Sigma \tau_2 \quad \Gamma \mid \Sigma \vdash t_1 :: \tau_{11} \xrightarrow{\sigma} \tau_{12} ; \sigma_1 \qquad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \delta :: \mathit{Pure}\ \sigma \quad \Gamma \mid \Sigma \vdash t_2 :: \tau_2 ; \sigma_2}{\Gamma \mid \Sigma \vdash \mathit{suspend}\ \delta\ t_1\ t_2 :: \tau_{12} ; \sigma_1 \vee \sigma_2} \quad \text{(TySuspend)}$$

$$\Gamma \mid \Sigma \vdash () :: () ; \bot \quad \text{(TyUnit)}$$

$$\frac{l : \tau \in \Sigma}{\Gamma \mid \Sigma \vdash \underline{l} :: \tau ; \bot} \quad \text{(TyLoc)}$$

The judgement form $\Gamma \mid \Sigma \vdash t :: \tau ; \sigma$ reads: with environment Γ and store typing Σ the term t has type τ and effect σ.

Many of these rules are standard, apart from the fact that we must use the similarity judgements $\kappa \sim_\Sigma \kappa'$ and $\varphi \sim_\Sigma \varphi'$ when performing comparisons.

TyAppT handles type application. Note that the type parameter is substituted into the resulting effect $\sigma_1[\varphi_2/a]$ as well as the resulting type $\varphi_{12}[\varphi_2/a]$. This ensures the effect term remains stable during evaluation. For example, if we were to omit this substitution then we could construct the evaluation:

$$\begin{array}{lll} & \emptyset & \\ & \mid\ \emptyset & \\ & \vdash\ \textbf{letregion}\ r_1\ \textbf{in} & \\ & \quad (\Lambda(r_2 : \%).\ \textbf{if}\ \mathit{True}\ r_2\ \textbf{then}\ \ldots\ \textbf{else}\ \ldots)\ r_1 & \\ & ::\ \ldots & \\ & ;\ \mathit{Read}\ r_2 & \\ \\ \longrightarrow & \emptyset & \\ & \mid\ \underline{\rho_1},\ r_1 \sim \rho_1 & \\ & \vdash\ (\Lambda(r_2 : \%).\ \textbf{if}\ \mathit{True}\ r_2\ \textbf{then}\ \ldots\ \textbf{else}\ \ldots)\ \underline{\rho_1} & \\ & ::\ \ldots & \\ & ;\ \mathit{Read}\ r_2 & \\ \\ \longrightarrow & \emptyset & \\ & \mid\ \underline{\rho_1},\ r_1 \sim \rho_1 & \\ & \vdash\ \textbf{if}\ \mathit{True}\ \underline{\rho_1}\ \textbf{then}\ \ldots\ \textbf{else}\ \ldots & \\ & ::\ \ldots & \\ & ;\ \mathit{Read}\ \underline{\rho_1} & \end{array}$$

When the term in the second step is evaluated, its effect changes from Read r_2 to Read $\underline{\rho_1}$. As there is no element in the store typing specifying that r_2 and $\underline{\rho_1}$ are similar, our preservation theorem would be violated.

4.2.19 Soundness of typing rules

Our proof of soundness is split into Progress and Preservation (subject reduction) theorems, in the usual way. The bulk of the proof is relegated to the appendix, but we repeat the main theorems below. Note that this proof addresses the soundness of the type system itself. Proving the validity of code transforms, such as the ones presented in §4.4, is another matter, but we will touch on it in the next section.

Theorem: (Progress)
Suppose we have a state H **;** t with store H and term t. Let Σ be a store typing which models H. If H is well typed with respect to Σ, and t is closed and well typed, and t contains no fabricated region witnesses (discussed in §4.2.5), then either t is a value or H **;** t can transition to the next state.

$$
\begin{array}{ll}
\text{If} & \emptyset \mid \Sigma \vdash t :: \tau \,;\, \sigma \\
\text{and} & \Sigma \models \mathrm{H} \\
\text{and} & \Sigma \vdash \mathrm{H} \\
\text{and} & \mathrm{nofab}(t) \\
\text{then} & t \in \mathrm{Value} \\
\text{or} & \text{for some } \mathrm{H}',\, t' \text{ we have} \\
 & \quad (\mathrm{H} \,;\, t \longrightarrow \mathrm{H}' \,;\, t' \;\text{ and }\; \mathrm{nofab}(t'))
\end{array}
$$

Theorem: (Preservation)
Suppose we have a state H **;** t with store H and term t. Let Σ be a store typing which models H. If H and t are well typed, and H **;** t can transition to a new state H′ **;** t' then for some Σ' which models H′, H′ is well typed, t' has a similar type to t, and the effect σ' of t' is no greater than the effect σ of t.

$$
\begin{array}{ll}
\text{If} & \Gamma \mid \Sigma \vdash t :: \tau \,;\, \sigma \\
\text{and} & \mathrm{H} \,;\, t \longrightarrow \mathrm{H}' \,;\, t' \\
\text{and} & \Sigma \vdash \mathrm{H} \;\text{ and }\; \Sigma \models \mathrm{H} \\
\text{then for some } \Sigma',\, \tau',\, \sigma' \text{ we have} & \\
 & \quad \Gamma \mid \Sigma' \vdash t' :: \tau' \,;\, \sigma' \\
\text{and} & \Sigma' \supseteq \Sigma \;\text{ and }\; \Sigma' \models \mathrm{H}' \;\text{ and } \Sigma' \vdash \mathrm{H}' \\
\text{and} & \tau' \sim_{\Sigma'} \tau \;\text{ and }\; \sigma' \sqsubseteq_{\Sigma'} \sigma
\end{array}
$$

Note that when a term is evaluated, its effect tends to become smaller, which is expressed as the $\sigma' \sqsubseteq_{\Sigma'} \sigma$ clause in the preservation theorem. For example, although *update* $\delta\ x\ y$ has the effect of reading y and writing x, it is reduced to (), which has no intrinsic effect.

Also note that the store typing grows during evaluation, which is expressed as the $\Sigma' \supseteq \Sigma$ clause of the preservation theorem. This means that once a region's constancy is set, it cannot be revoked, or changed during evaluation.

4.2.20 Goodness of typing rules

When combined, the Progress and Preservation theorems outlined in the previous section guarantee that a reduction of a well typed program does not "get stuck", meaning that it can always be reduced to normal form. Although this "does not get stuck" is the classic interpretation of Milner's famous mantra "well typed programs don't go wrong", anyone who has spent time writing programs will appreciate that there are plenty of ways a program can "go wrong" besides failing to reduce to normal form.

The trouble is that "well typed programs" can still contain plenty of bugs. For compiler writers, a freshly compiled program failing to reduce to normal form usually manifests as a runtime crash, or as an exception being thrown. This occurrence, in fact, is often followed by a sigh of relief. It is a relief because a program that crashes at the same point, every time, in a predictable way, is usually straightforward to debug. In contrast, one that runs to completion but gives the wrong answer provides no direct clue as to the location of the problem in the original source code.

With this in mind, the fact that a type system is sound is only the first step along the road to goodness. What is equally important, is that a program that would be considered "buggy" by its creator also has a high likelihood of being mistyped.

This is the primary reason for using a typed core language in a compiler. As a compiler writer, when you make a mistake you want that mistake to manifest itself as soon as possible. Having a compiled program simply give the wrong answer is always the worst result. Performing optimisations on programs that use mutable data structures doesn't require a complicated type system like ours: with its regions, effects, witness types, dependent kinds and so on. Information concerning what side effects a function has, and which objects are mutable, could equally be stored in tables. However, the benefit of using a typed core language is that this information can also be *checked*.

The real purpose of witnesses

We now come to discuss the real purpose of witnesses in our compiler. We view a witness as a token that gives us permission to perform a particular operation. In particular, the operations that we use them for, namely updating data and suspending computations, are ones that frequently result in hard to diagnose bugs if not handled correctly. Updating an object that was supposed to be constant is not "unsound", but it's probably buggy. Likewise, suspending a computation that isn't pure is not "unsound", but it's probably buggy.

For the first case, having a witness of kind *Mutable* r gives us permission to update data in the region named r. When a region is created by reducing a **letregion** expression, whether that region is going to be constant or mutable is decided at that point. This is shown in the EvLetRegion rule from §4.2.13. Now, as discussed in §4.2.7, in our semantics this decision results in either a (const ρ) or (mutable ρ) property being added to the store. At the same moment, we also get a witness as to which option we chose, which serves as a record of the decision.

Later on in the reduction, we may want to update some object in this same region. Of course, this should only be permitted if we decided the region was

going to be mutable in the first place. This is why, in EvUpdate3 from §4.2.13, the *update* operator requires that we supply it with our witnesses of mutability. Also note that in that same rule, there must be a corresponding (mutable ρ) property in the store. Now, from our Progress theorem we know that we can always apply this transition rule. This, in turn, means that if we can provide a witness of mutability for some region, then we know that the corresponding property is in the store, and that means we really did decide it was going to be mutable when it was created.

On the other hand, if we decide that a new region is going to be constant, then we get a (const ρ) property in the store instead of (mutable ρ). We also get a *Const r* witness in the program, which records this decision. Now, unless we enjoy tracking down difficult-to-find bugs, all data read by a suspended function application should be constant. This ensures that we get the same result independent of when the suspension happens to be forced. This is why, in the EvSuspend4 rule from §4.2.13, we must provide the suspend operator with a witness of purity for the application to be suspended. By inspection of the kinds of witness constructors in §4.2.15 the only way such a witness of purity can be produced is by combining witnesses of constancy for the regions that will be (visibly) read. Finally, the fact that we can come up with said witnesses of constancy ensures that witnesses of mutability for those same regions do not exist elsewhere in the program.

A witness guarantees that something will not be done

For another way of thinking about witnesses, note that the utility of a *Const r* witness is not so much that it encodes that a region is constant, rather, it guarantees that it will not be updated. In the dual case, the utility of a *Pure e* witness, is that it guarantees that a function application with effect e will not read from regions that are mutable.

4.3 Extensions to the simplified language

The simplified core language of §4.2 is not *syntactically complete* with respect to the source language. This means that it cannot directly express all the possible well typed source programs. It does not include algebraic data types, case expressions, effect masking or bounded quantification. In addition, the typing rule for if-expressions does not allow us to choose between two functions that have the same value type, but differing effects.

The additions needed for algebraic data types and case expressions are unsurprising, so we will not discuss them further. We discuss the others in turn.

4.3.1 Masking non-observable effects

We mask three sorts of effects: effects of computations that are unobservable, effects on freshly allocated values, and effects that are known to be pure. The following rule handles the first sort:

$$\frac{\Gamma \mid \Sigma \vdash t :: \tau ; \sigma \quad r \notin fv_T(\Gamma) \quad r \notin fv(\tau)}{\Gamma \mid \Sigma \vdash t :: \tau ; \sigma \setminus (\mathit{Read}\ r \vee \mathit{Write}\ r)} \qquad \text{(TyMaskObserve)}$$

This rule encodes the observation criteria discussed in §2.4. It says that if a region variable is not present in the type environment or type of a term, then we can ignore the fact that its evaluation will perform read or write actions on the associated region. As we treat $\vee$ as akin to set union $\cup$ the effect minus operator $\setminus$ is defined in the obvious way.

Note that it is safe to allow kind bindings of the form $r : \%$ to be present in the environment, as long as the region variable is not mentioned in the τ part of any $x : \tau$. This is handled by the $fv_T(\Gamma)$ function which is defined as:

$$fv_T(\Gamma) = \bigcup \{\ fv(\tau) \mid x : \tau \in \Gamma\ \}$$

For a concrete implementation, the trouble with TyMaskObserve is that it is not syntax directed. It is valid to apply this rule to any term, but applying it to *every* term could be too slow at compile time. Usefully, when reconstructing the type of a term we only need to perform this sort of masking on sub-terms that are the bodies of λ-abstractions. This is because the typing rule for abstractions is responsible for moving effect information from the σ in $\Gamma \mid \Sigma \vdash t :: \tau ; \sigma$ into the value type expression.

Instead of using a separate TyMaskObserve rule, we find it convenient to incorporate effect masking directly into the rule for λ-abstractions. This gives:

$$\frac{\begin{array}{r}\Gamma,\ x : \tau_1 \mid \Sigma \vdash t :: \tau_2 ; \sigma \\ r \notin fv_T(\Gamma) \quad r \notin fv(\tau_1) \cup fv(\tau_2) \quad \sigma' = \sigma \setminus (\mathit{Read}\ r \vee \mathit{Write}\ r)\end{array}}{\Gamma \mid \Sigma \vdash \lambda(x : \tau_1).\, t :: \tau_1 \xrightarrow{\sigma'} \tau_2 ; \bot} \qquad \text{(TyAbsObserve)}$$

An alternative would be to combine the effect masking TyLetRegion, but this would require the type checker to inspect the effect term more frequently.

4.3.2 Masking effects on fresh regions

As discussed in §2.3.7, we can mask effects that are only used to compute the result of a function, and are not otherwise visible. Here is the rule to do so:

$$\frac{\Gamma \mid \Sigma \vdash \lambda(x:\tau).\ t \ ::\ \tau \xrightarrow{\sigma} \mathit{Bool}\ \ r\ ;\ \sigma' \qquad \begin{array}{r} r \notin \mathit{fv}_T(\Gamma) \quad r \notin \mathit{fv}(\tau) \\ \sigma'' = \sigma \setminus (\mathit{Read}\ \ r \vee \mathit{Write}\ \ r) \end{array}}{\Gamma \mid \Sigma \vdash \lambda(x:\tau).\ t \ ::\ \tau \xrightarrow{\sigma''} \mathit{Bool}\ \ r\ ;\ \sigma'} \qquad \text{(TyMaskFresh)}$$

As have not included closure typing information in our simplified language, we have to tie TyMaskFresh to the lambda abstraction $\lambda(x:\tau).\ t$. This prevents us from inadvertently masking effects on regions present in the closure of the function. When TyMaskFresh is applied, all the free variables in t (the closure) must be present in the environment Γ, and thus the term $\mathit{fv}_T(\Gamma)$ accounts for them.

Note that in TyMaskFresh we have set the result type of the function to *Bool* as that is the only non-function value type constructor in our simplified language. For the full language, we can replace *Bool* by any type constructor, so long as we only mask effects on region variables that are in strongly material positions. Materiality was discussed in §2.5.3.

Returning to the issue of closure typing, note that the following more general variant of TyMaskFresh is bad.[1]

$$\frac{\Gamma \mid \Sigma \vdash t \ ::\ \tau \xrightarrow{\sigma} \mathit{Bool}\ \ r\ ;\ \sigma' \qquad \begin{array}{r} r \notin \mathit{fv}_T(\Gamma) \quad r \notin \mathit{fv}(\tau) \\ \sigma'' = \sigma \setminus (\mathit{Read}\ \ r \vee \mathit{Write}\ \ r) \end{array}}{\Gamma \mid \Sigma \vdash t \ ::\ \tau \xrightarrow{\sigma''} \mathit{Bool}\ \ r\ ;\ \sigma'} \qquad \text{(BadTyMaskFresh)}$$

We can see why BadTyMaskFresh is bad by considering its (non) applicability in the following program. Note that for a clearer example, we have taken the liberty of using *Int* instead of *Bool*.

$$\begin{array}{lllll} \mathit{makeInc} & & & & \\ \quad = \lambda(). & \textbf{let} & x & = 0\ r_1 & \\ & & f & = \lambda().\ \textbf{do}\ \{\ x := x + 1\ r_1;\ x\ \} & \\ & \textbf{in} & f & & \end{array}$$

This program is similar in spirit to the examples from §2.5.2. It allocates a mutable integer x, then returns a function that updates it and returns its value. Without masking, the type of the inner let-expression is:

$$(\textbf{let}\ x = 0\ ...\ \textbf{in}\ f) \ ::\ () \xrightarrow{\mathit{Read}\, r_1\ \vee\ \mathit{Write}\, r_1} \mathit{Int}\ \ r_1$$

At this point, the type environment only needs to contain the term $r_1 : \%$ and type for $(+)$, which has no free variables. If we were to apply BadTyMaskFresh here, then we would end up with:

$$(\textbf{let}\ x = 0\ ...\ \textbf{in}\ f) \ ::\ () \to \mathit{Int}\ \ r_1$$

[1] We avoid the word *unsound* because its use will not prevent a term from being reduced to normal form. However, if we apply it during type reconstruction and then optimise the program based on this information, then we run the risk of producing a program that gives an unintended answer, hence badness.

This is invalid because the expression will return a different value each time we apply it to (), hence we cannot reorder calls to it.

In future work we plan to extend our core language with closure typing information. This would allow us to use a rule similar to the following:

$$\frac{\Gamma \mid \Sigma \vdash t :: \tau \xrightarrow{\sigma\ \varsigma} Bool\ r\ ;\ \sigma' \qquad \begin{array}{r} r \notin fv_T(\Gamma) \quad r \notin fv(\tau) \quad r \notin fv(\varsigma) \\ \sigma'' = \sigma \setminus (Read\ r \vee Write\ r) \end{array}}{\Gamma \mid \Sigma \vdash t :: \tau \xrightarrow{\sigma''\ \varsigma} Bool\ r\ ;\ \sigma'} \quad \text{(CloTyMaskFresh)}$$

When we include closure typing information in the previous example, the type of the let-expression becomes:

$$(\textbf{let}\ x = 0\ ...\ \textbf{in}\ f) :: () \xrightarrow{(Read\, r_1\ \vee\ Write\, r_1)\ (x{:}Int\ \ r)} Int\ \ r_1$$

This closure term $x : Int\ \ r$ reveals the fact that successive applications of this function will share a value of type $Int\ \ r$. Because of this we cannot guarantee that the returned value is fresh, so we cannot mask effects on it.

4.3.3 Masking pure effects

Recall the *mapL* function from §2.3.10 which performs a spine-lazy map across the elements of a list. We will convert its definition to the core language. Firstly, the source type of *mapL* is:

$$\begin{array}{lll} mapL & :: & \forall\ a\ b\ r_1\ r_2\ e_1 \\ & . & (a \xrightarrow{e_1} b) \rightarrow List\ \ r_1\ a \xrightarrow{e_2} List\ \ r_2\ b \\ & , & e_2 = Read\ \ r_1 \vee e_1 \\ & \rhd & Pure\ \ e_1,\ Const\ \ r_1 \end{array}$$

To convert this type to core, we write the purity and constancy constraints in prefix form, and place the manifest effect term directly on the corresponding function constructor:

$$\begin{array}{lll} mapL & :: & \forall\ a\ b\ r_1\ r_2\ e_1 \\ & . & Pure\ \ e_1 \Rightarrow Const\ \ r_1 \\ & \Rightarrow & (a \xrightarrow{e_1} b) \rightarrow List\ \ r_1\ a \xrightarrow{Read\ \ r_1 \vee e_1} List\ \ r_2\ b \end{array}$$

The desugared version of the function definition follows. We have expanded the pattern matching syntax, the infix use of @, and have introduced a binding for each function argument:

```
mapL
 =  λf. λxx.
    case xx of
      Nil          → Nil
      Cons x xs    →
        do  x'     = f x
            mapL'  = mapL f
            xs'    = suspend1 mapL' xs
            Cons x' xs'
```

From the type of *mapL* we see that the core version of the function should have seven type parameters: five due to the universal quantifier, and two to bind the witnesses for $Pure\ e_1$ and $Const\ r_1$. We will add these type arguments, along with type applications where required:

$$
\begin{array}{l}
mapL \\
= \quad \Lambda\ a\ b\ r_1\ r_2\ e_1. \\
\quad\ \ \Lambda\ w_1 \quad :: Pure\ e_1. \\
\quad\ \ \Lambda\ w_2 \quad :: Const\ r_1. \\
\quad\ \ \lambda\ f \quad\ \ :: a \xrightarrow{e_1} b. \\
\quad\ \ \lambda\ xx \quad :: List\ r_1\ a. \\
\quad\ \ \textbf{case}\ xx\ \textbf{of} \\
\quad\quad Nil \qquad\quad \rightarrow Nil\ a\ r_2 \\
\quad\quad Cons\ x\ xs \quad \rightarrow \\
\quad\quad \textbf{do} \quad x' \qquad = f\ x \\
\quad\quad\quad\ \ mapL' \quad = mapL\ a\ b\ r_1\ r_2\ e_1\ w_1\ w_2\ f \\
\quad\quad\quad\ \ xs' \qquad = suspend1 \\
\quad\quad\quad\quad\quad\quad (List\ r_1\ a)\ (List\ r_2\ b)\ (Read\ r_1 \vee e_1) \\
\quad\quad\quad\quad\quad\quad (MkPureJoin\ (Read\ r_1)\ e_1\ (MkPurify\ r_1\ w_2)\ w_1) \\
\quad\quad\quad\ \ Cons\ b\ r_2\ x'\ xs'
\end{array}
$$

We have elided the kind annotations on the first five type parameters to aid readability. The variables w_1 and w_2 bind witnesses to the facts that e_1 is pure and r_1 is constant. Note that in the recursive call to *mapL* all of its type parameters must be passed back to itself. We also add type applications to satisfy the quantifiers and constraints on *suspend1*, and to satisfy *Nil* and *Cons*. For reference, *Nil* and *Cons* have the following types:

$$
\begin{array}{ll}
Nil & :: \forall a\ r_1.\ List\ r_1\ a \\
Cons & :: \forall a\ r_1.\ a \rightarrow List\ r_1\ a \rightarrow List\ r_1\ a
\end{array}
$$

Note that because the type we used for *mapL* contains the effect term $Read\ r_1 \vee e_1$, when we call *suspend1* we must provide a witness that this effect is pure. This is the reason for the $(MkPureJoin\ (Read\ r_1)\ e_1\ (MkPurify\ r_1\ w_2)\ w_1)$ term. Such witnesses were discussed in §4.1.4.

This is a valid translation, but as mentioned in §2.3.10 it would be "nicer" if we could mask the $Read\ r_1$ and e_1 effects and not have to write them in the type. After all, the point of proving that a particular effect is pure is so we can ignore it from then on. Masking these effects in the type is straightforward, and the core version is:

$$
\begin{array}{lll}
mapL & :: & \forall\ a\ b\ r_1\ r_2\ e_1 \\
 & . & Pure\ e_1 \Rightarrow Const\ r_1 \\
 & \Rightarrow & (a \xrightarrow{e_1} b) \rightarrow List\ r_1\ a \xrightarrow{\bot} List\ r_2\ b
\end{array}
$$

However, using this type requires that we add a mechanism to mask the equivalent effects in the core program. One option is to add a rule similar to TyMaskObserve from §4.3.1:

$$
\frac{\Gamma \mid \Sigma \vdash t :: \tau\ ;\ \sigma \quad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \delta :: Pure\ \sigma'}{\Gamma \mid \Sigma \vdash t :: \tau\ ;\ \sigma \setminus \sigma'} \qquad \text{(TyMaskPure)}
$$

However, as with TyMaskObserve, this rule is not syntax directed. Another option is to introduce an explicit masking keyword, which states the witness being used to mask the effect of a particular expression. For example:

$$\frac{\Gamma \mid \Sigma \vdash t :: \tau \,;\, \sigma \quad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \delta :: \mathit{Pure}\ \sigma'}{\Gamma \mid \Sigma \vdash \mathbf{mask}\ \delta\ \mathbf{in}\ t :: \tau \,;\, \sigma \setminus \sigma'} \qquad \text{(TyMaskPureEx)}$$

The following code is a core version of the *mapL* function that uses the **mask** keyword, and has the "nice" type mentioned above. Note that because the effect of *mapL* is now $\bot$ we use this as the effect argument to *suspend1*.

$$\begin{array}{l}
\mathit{mapL} \\
= \quad \Lambda\ a\ b\ r_1\ r_2\ e_1. \\
\quad\ \ \Lambda\ w_1 \quad :: \mathit{Pure}\ e_1. \\
\quad\ \ \Lambda\ w_2 \quad :: \mathit{Const}\ r_1. \\
\quad\ \ \lambda\ f \quad\ \ :: a \xrightarrow{e_1} b. \\
\quad\ \ \lambda\ xx \quad :: \mathit{List}\ r_1\ a. \\
\quad\ \ \mathbf{mask}\ \mathit{MkPureJoin}\ (\mathit{Read}\ r_1)\ e_1\ (\mathit{MkPurify}\ r_1\ w_2)\ w_1\ \mathbf{in} \\
\quad\ \ \mathbf{case}\ xx\ \mathbf{of} \\
\quad\quad \mathit{Nil} \qquad\quad \rightarrow \mathit{Nil}\ a\ r_2 \\
\quad\quad \mathit{Cons}\ x\ xs \quad \rightarrow \\
\quad\quad\ \mathbf{do} \quad x' \qquad = f\ x \\
\quad\quad\quad\ \ \mathit{mapL}' \quad = \mathit{mapL}\ a\ b\ r_1\ r_2\ e_1\ w_1\ w_2\ f \\
\quad\quad\quad\ \ xs' \qquad = \mathit{suspend1} \\
\quad\quad\quad\quad\quad\quad (\mathit{List}\ r_1\ a)\ (\mathit{List}\ r_2\ b)\ \bot \\
\quad\quad\quad\quad\quad\quad (\mathit{MkPure}\ \bot) \\
\quad\quad\quad\ \ \mathit{Cons}\ b\ r_2\ x'\ xs'
\end{array}$$

4.3.4 Bounded quantification

We add bounded quantification to the core language so we can support the higher order programs discussed in §2.3.6. For example, when converted to core, the third order function *foo* has the following type and definition:

$$\begin{array}{lll}
\mathit{foo} & :: & \forall\ r_1\ r_2\ r_3\ r_4\ (e_1 \sqsupseteq \mathit{Read}\ r_1)\ e_2 \\
& . & ((\mathit{Int}\ r_1 \xrightarrow{e_1} \mathit{Int}\ r_2) \xrightarrow{e_2} \mathit{Int}\, r_3) \xrightarrow{e_2 \vee \mathit{Read}\, r_3} \mathit{Int}\, r_4
\end{array}$$

$$\begin{array}{l}
\mathit{foo} \\
= \quad \Lambda\ r_1\ r_2\ r_3\ r_4\ (e_1 \sqsupseteq \mathit{Read}\ r_1)\ e_2. \\
\quad\ \ \lambda\ f : (\mathit{Int}\ r_1 \xrightarrow{e_1} \mathit{Int}\ r_2) \xrightarrow{e_2} \mathit{Int}\ r_3. \\
\quad\ \ \mathbf{do} \quad x_1 \quad = \mathit{succ}\ r_1\ r_2 \\
\quad\quad\ \ x_2 \quad = f\ x_1 \\
\quad\quad\ \ \mathit{succ}\ r_3\ r_4\ x_2
\end{array}$$

Note that in the application $f\ x_1$ the expected type of the argument is:

$$f \quad :: \mathit{Int}\ r_1 \xrightarrow{e_1} \mathit{Int}\, r_2$$

but x_1 has type:

$$x_1 \quad :: \mathit{Int}\ r_1 \xrightarrow{\mathit{Read}\ r_1} \mathit{Int}\ r_2$$

To support this we modify the rule for application so that the argument may have any type that is subsumed by the type of the function parameter. We

arrange the typing rule for bounded type abstraction to add its constraint to the type environment, and use this to show that applications such as $f\ x_1$ are valid.

The additions to the core language are as follows:

$$\begin{array}{lcll} \varphi & \rightarrow & \ldots & \\ & | & \forall(e \sqsupseteq \sigma).\ \tau & \text{(bounded quantification)} \end{array}$$

$$\begin{array}{lcll} t & \rightarrow & \ldots & \\ & | & \Lambda(e \sqsupseteq \sigma).\ t & \text{(bounded type abstraction)} \end{array}$$

Operationally, bounded type application behaves the same way as the unbounded case:

$$\mathrm{H}\ ;\ (\Lambda(e \sqsupseteq \sigma).\ t)\ \varphi \longrightarrow \mathrm{H}\ ;\ t[\varphi/e] \qquad \text{(EvTAppAbsB)}$$

The new typing rules are:

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \sigma\ ::\ !\quad \Gamma,\ e : ! \mid \Sigma \vdash_{\mathrm{T}} \tau\ ::\ \kappa\quad e \notin fv(\Gamma)}{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \forall(e \sqsupseteq \sigma).\ \tau\ ::\ \kappa_2} \qquad \text{(KiAllB)}$$

$$\frac{\begin{array}{c}\Gamma \mid \Sigma \vdash_{\mathrm{T}} \sigma_1\ ::\ ! \\ \Gamma,\ e : !,\ e \sqsupseteq \sigma_1 \mid \Sigma \vdash\ t_2\ ::\ \tau_2\ ;\ \sigma_2\end{array}}{\Gamma \mid \Sigma \vdash\ \Lambda(e \sqsupseteq \sigma_1).\ t_2\ ::\ \forall(e \sqsupseteq \sigma_1).\ \tau_2\ ;\ \sigma_2} \qquad \text{(TyAbsTB)}$$

$$\frac{\begin{array}{l}\Gamma \mid \Sigma \vdash\ t_1\ ::\ \forall(e \sqsupseteq \sigma_{11}).\ \tau_{12}\ ;\ \sigma_1 \\ \Gamma \mid \Sigma \vdash_{\mathrm{T}} \sigma_2\ ::!\qquad\qquad\qquad\qquad \sigma_{11} \sqsubseteq_\Sigma \sigma_2\end{array}}{\Gamma \mid \Sigma \vdash\ t_1\ \sigma_2\ ::\ \tau_{12}[\sigma_2/e]\ ;\ \sigma_1[\sigma_2/e]} \qquad \text{(TyAppTB)}$$

$$\frac{\begin{array}{l}\Gamma \mid \Sigma \vdash\ t_2\ ::\ \tau_2\ ;\ \sigma_2 \\ \Gamma \mid \Sigma \vdash\ t_1\ ::\ \tau_{11} \xrightarrow{\sigma} \tau_{12}\ ;\ \sigma_1\quad \tau_2 \sqsubseteq_\Sigma \tau_{11}\end{array}}{\Gamma \mid \Sigma \vdash\ t_1\ t_2\ ::\ \tau_{12}\ ;\ \sigma_1 \vee \sigma_2 \vee \sigma} \qquad \text{(TyAppB)}$$

$$\frac{a \sqsupseteq \sigma \in \Gamma}{\Gamma \mid \Sigma \vdash\ \sigma\ \sqsubseteq\ a} \qquad \text{(SubVar)}$$

$$\frac{\begin{array}{l}\Gamma \mid \Sigma \vdash\ \tau_{21}\ \sqsubseteq\ \tau_{11} \\ \Gamma \mid \Sigma \vdash\ \tau_{12}\ \sqsubseteq\ \tau_{22}\quad \Gamma \mid \Sigma \vdash\ \sigma_1\ \sqsubseteq\ \sigma_2\end{array}}{\Gamma \vdash\ \tau_{11} \xrightarrow{\sigma_1} \tau_{12}\ \sqsubseteq\ \tau_{21} \xrightarrow{\sigma_2} \tau_{22}} \qquad \text{(SubFun)}$$

$$\frac{\Sigma \vdash_{\mathrm{T}}\ \kappa_1 \sim \kappa_2\quad \Sigma \vdash\ \sigma_1 \sim \sigma_2\quad \Sigma \vdash\ \tau_1 \sim \tau_2}{\Sigma \vdash\ \forall(a \sqsupseteq \sigma_1).\ \tau_1 \sim \forall(a \sqsupseteq \sigma_2).\ \tau_2} \qquad \text{(SimAllB)}$$

KiAllB and TyAbsTB are similar to their unbounded counterparts. Note that bounded quantification is only defined for effects, so the bounding type has this kind.

In TyAbsTB, the effect bound $e \sqsupseteq \sigma_1$ is added to the type environment, and SubVar is used to retrieve it higher up in the proof tree.

In TyAppTB we use subsumption on effects, $\sigma_{11} \sqsubseteq_\Sigma \sigma_2$, to satisfy the bound on the quantifier.

In TyAppB we use a subsumption judgement on value types, $\tau_2 \sqsubseteq_\Sigma \tau_{11}$, to support applications such as the one described in the *foo* example. Note that although subsumption only has meaning on the effect portion of a type, we still need to define it to work on value types. This is because the effect type information is attached to the value type information.

We don't really need contravariance

In SubFun, although we use contravariant subsumption, $\tau_{21} \sqsubseteq \tau_{11}$, for the function parameter, in practice this contravariance isn't used. We could have equally written $\tau_{11} \sqsubseteq \tau_{21}$. This arises due to the way we strengthen inferred types, discussed in §2.3.6. As we do not strengthen constraints on effect variables that appear in the types of function parameters, the effect annotations on such types will always be variables.

In TyAppB, when we apply a function to its argument, we use the subsumption judgement to invoke the SubVar rule, which accepts examples like *foo*. However, in that example we only applied a second order function to a first order one. Annotations on function arrows of higher order will always be variables, so applications involving them are accepted via SubRefl. The variance of function types does not come into play.

For contrast, the process algebra of [NN93] includes an effect system in which variance *does* matter. However, that work is presented as a stand-alone language, not as a core language embedded in a larger compiler. For DDC, we cannot write a program in the source language that maps onto a core-level program in which variance matters, so we have not invested further effort into supporting it.

This situation is similar to when System-F is used as a basis for the core language of a Haskell 98 compiler. System-F supports higher ranked types [PJVWS07], but Haskell 98 doesn't. Types of rank-2 can be introduced when performing lambda lifting [PJ87], but no terms are produced that have types of rank higher than this. The compiler does not need to support full System-F because only a fragment of that language is reachable from source.

4.3.5 Effect joining in value types

Consider the following source program:

$$
\begin{array}{lll}
\mathit{five} & = 5 & \\
f & = \textbf{if} \ldots & \\
 & \quad \textbf{then} & (\lambda().\ \mathit{succ}\ \mathit{five}) \\
 & \quad \textbf{else} & (\lambda().\ \textbf{do}\ \{\ \mathit{putStr}\ \text{“}\texttt{hello}\text{”};\ \mathit{succ}\ \mathit{five}\})
\end{array}
$$

with

$$
\begin{array}{lll}
\mathit{putStr} & :: & \forall r_1.\, \mathit{String}\ r_1 \xrightarrow{e_1} () \\
 & \rhd & e_1 = \mathit{Read}\ r_1 \vee \mathit{Console}
\end{array}
$$

Note that in the definition of f, the two functions in the right of the if-expression have different effects. The first reads the integer bound to *five*, but the second also prints to the console. If we set *five* to have type $\mathit{Int}\ r_5$, then these two expressions have the following types:

$$
\begin{array}{l}
(\lambda x.\ \mathit{succ}\ \mathit{five}) \\
\quad :: \quad () \xrightarrow{e_1} \mathit{Int}\ r_1 \\
\quad \rhd \quad e_1 = \mathit{Read}\ r_5
\end{array}
$$

$$
\begin{array}{l}
(\lambda().\ \textbf{do}\ \{\ \mathit{putStr}\ \text{“}\texttt{hello}\text{”};\ \mathit{succ}\ \mathit{five}\}) \\
\quad :: \quad () \xrightarrow{e_2} \mathit{Int}\ r_1 \\
\quad \rhd \quad e_2 = \mathit{Read}\ r_5 \vee \mathit{Console}
\end{array}
$$

As it stands, we cannot translate this program directly to our core language, because the typing rule TyIf of §4.2.18 requires both alternatives to have similar types, inclusive of effect information. We support such programs by extending the definition of $\vee$ to join the effects contained within value types, and use this operator to compute the resulting type of if-expressions. This mirrors what happens during type inference.

$$
(\tau_1 \xrightarrow{e_1} \tau_2) \vee (\tau_1 \xrightarrow{e_2} \tau_3) \quad \equiv \quad \tau_1 \xrightarrow{e_1 \vee e_2} (\tau_2 \vee \tau_3)
$$

$$
\frac{\Gamma \mid \Sigma \vdash t_1 :: \mathit{Bool}\ \varphi\ ;\ \sigma_1 \qquad \Gamma \mid \Sigma \vdash t_2 :: \tau_2\ ;\ \sigma_2 \qquad \Gamma \mid \Sigma \vdash t_3 :: \tau_3\ ;\ \sigma_3 \qquad \tau = \tau_2 \vee \tau_3}{\Gamma \mid \Sigma \vdash \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 :: \tau\ ;\ \sigma_1 \vee \sigma_2 \vee \sigma_3 \vee \mathit{Read}\ \varphi} \qquad \text{(TyIfJoin)}
$$

Note that as our type inference algorithm only performs generalisation at let-bindings, the types of alternatives will never contain quantifiers. For this reason we don't need to define $\vee$ over quantified types. Also, as we only strengthen the manifest effects of a function type, the effect annotations on parameters will always be variables. Similarly to §4.3.4, this guarantees that the effect annotations in function parameters are variables, so we don't have to join them.

An alternate method would be to provide an explicit type annotation for the result of the if-expression, and use the subsumption judgement to check that the types of both alternatives are less than the annotation. This approach was taken in [NN93], but we avoid it because it increases the volume of annotation.

4.4 Optimisations

For DDC, the primary purpose of tracking effect information is to support compiler optimisations. With that said, we don't present any new ones, nor do we make substantial improvements over existing ones. What we *do* provide, is the ability to perform the same sort of optimisations previously reserved for purely functional languages such as Haskell, but now in the presence of side effects. The great enablers are the let-floating transforms discussed in [PJPS96]. These allow bindings to be moved from their definition sites to their use sites. This in turn exposes opportunities for other simple, correctness preserving transforms, many of which are described in Santos's thesis [dMS95].

We will discuss how our effect and mutability information can be used to perform these transforms. The reader is advised to consult [PJPS96] or [dMS95] for matters not relating to effects.

We distinguish four kinds of let-floating transforms, and will consider each in turn:

- Local transforms that are part of the language normalisation process.
- Floating bindings to other positions at the same scope level.
- Floating into if/case alternatives.
- Floating outside lambda abstractions.

Although we present our examples in the core language discussed in §4.2, we elide type annotations when they do not contribute to the discussion. We also make use of standard features such as integers, case expressions and unboxed values.

As DDC is still a research prototype, we do not present any concrete speed-up figures for these optimisations. Such figures would be skewed by the naive implementation of our runtime system. See [dMS95] for data relating to a mature compiler. As the saying goes, "what's amazing is not how well the bear dances – what's amazing is that the bear dances at all!"

4.4.1 Local transforms

Here is an example local transform, given in [PJPS96]:

$$(\textbf{let}\ v\ =\ e\ \textbf{in}\ b)\ \ a\ \longrightarrow\ \textbf{let}\ v\ =\ e\ \textbf{in}\ b\ a$$

Although this is a valid transform, its applicability in a concrete implementation depends on whether the core program can ever contain a term of the initial form. There is also the question of whether a term in the resulting form can be directly translated to the back-end intermediate language (or machine code).

In DDC, we keep the core program normalised so that the first term of an application is always a variable. We do this because we compile via C, and this process does not support more general forms of application. In this sense, the above transform is not an optimisation *per se*, because it is part of the normalisation process, and must always be performed.

4.4.2 Floating at the same level

Floating bindings at the same scope level serves to expose opportunities for other transforms. Local unboxing is one such transform, and is a simple, well known technique for eliminating the majority of boxing and unboxing operations in numeric code. We discuss how to perform it in the presence of side effects by using the type information present in the core language. Local unboxing can also be expressed as a backwards dataflow analysis, but we use (forwards) let-floating as the presentation is simpler.

Here is a simple program which takes the successor of an integer, as well as updating it:[2]

$$
\begin{array}{l}
succUpdate\ \ x \\
= \textbf{do} \quad y = succ\ \ x \\
\qquad\quad x := 5 \\
\qquad\quad succ\ \ y
\end{array}
$$

Converting this program to the core language yields the following:

$$
\begin{array}{l}
succUpdate \\
= \quad \Lambda\ r_1\ r_2\ (w_1 : Mutable\ \ r_1). \\
\qquad \lambda\ x \quad : Int\ \ r_1. \\
\qquad \textbf{letregion}\ r_3\ \textbf{with}\ \{w_3 = MkConst\ \ r_3\}\ \textbf{in} \\
\qquad \textbf{letregion}\ r_4\ \textbf{with}\ \{w_4 = MkConst\ \ r_4\}\ \textbf{in} \\
\qquad \textbf{do} \quad y = box\ \ r_3\ (succ_{\#}\ (unbox\ \ r_1\ (force\ \ x))) \\
\qquad\qquad\quad updateInt_{\#}\ \ r_1\ w_1\ (force\ \ x)\ (unbox\ \ r_4\ (box\ \ r_4\ 5_{\#})) \\
\qquad\qquad\quad box\ \ r_1\ (succ_{\#}\ (unbox\ \ r_3\ (force\ \ y)))
\end{array}
$$

In this translation we have expanded the boxed numeric functions *succ* and *updateInt* into a combination of boxing, unboxing, and thunk forcing operators. Here are the types of these new operators:

$$
\begin{array}{lcl}
unbox & :: & \forall r_1.\ \ Int\ \ r_1 \xrightarrow{Read\ r_1} Int_{\#} \\
box & :: & \forall r_1.\ \ Int_{\#} \rightarrow Int\ \ r_1 \\
succ_{\#} & :: & Int_{\#} \rightarrow Int_{\#} \\
updateInt_{\#} & :: & \forall r_1.\, Mutable\ \ r_1 \Rightarrow Int\ \ r_1 \xrightarrow{Write\ r_1} Int_{\#} \rightarrow ()
\end{array}
$$

We use $Int_{\#}$ as the unboxed version of *Int*, and write unboxed literals as $5_{\#}$. A value of type $Int_{\#}$ can be held in a machine register. For the reasons discussed in §2.1, plain unboxed integers are non-updatable and thus do not need a region variable. As an aside, when we wish to store updatable arrays of unboxed integers in the heap, we give the array the type $Ptr_{\#}\ \ r_1\ \ Int_{\#}$, and attach the mutability constraint to the pointer type instead.

Note that $updateInt_{\#}$ uses the value of an unboxed integer to update a boxed one. The boxed integer resides in the heap, not in a register.

In this thesis we treat *force* as a primitive of the core language. *force* tests its argument to see if it is represented by an object with an outermost thunk, and forces that thunk if need be. We treat it as a primitive operator because our

[2] This example has been kept simple to so that the typeset intermediate code is a managable size. Hopefully the reader can appreciate that the techniques also scale to more interesting programs.

type system is not expressive enough to write a sensible type of *force* other than $\forall a.\ a \to a$. We could use this type, but doing so would introduce a large number of superfluous type applications in our example. See §5.2.8 for a discussion of how we might give a better type to *force*.

From the above code, we can already see an obvious optimisation. In the second argument of $updateInt_\#$ we can collapse the term $(unbox\ r_4\ (box\ r_4\ 5_\#))$ into just $5_\#$.

After we have exposed the primitive boxing, unboxing and forcing operators, the next step is to flatten the program so that each binding consists of a single application. This increases the mobility of each binding, that is, the probability that it will be safe to move it. Note that order of evaluation in the program runs left-to right, depth first, so the bindings come out in the following order:

$$
\begin{array}{lll}
succUpdate\ x & & \\
= \quad \Lambda\ r_1\ r_2\ (w_1 : Mutable\ r_1). & & \\
\quad \lambda\ x\ \ : Int\ r_1. & & \\
\quad \textbf{letregion}\ r_3\ \textbf{with}\ \{w_3 = MkConst\ r_3\}\ \textbf{in} & & \\
\quad \textbf{letregion}\ r_4\ \textbf{with}\ \{w_4 = MkConst\ r_4\}\ \textbf{in} & & \\
\quad \textbf{do} \quad x_1 \quad = force\ x & & \bot \\
\qquad\quad x_2 \quad = unbox\ r_1\ x_1 & & Read\ r_1 \\
\qquad\quad y_1 \quad = succ_\#\ x_2 & & \bot \\
\qquad\quad y \quad\ = box\ r_3\ y_1 & & \bot \\
\qquad\quad u_1 \quad = 5_\# & & \bot \\
\qquad\quad u_2 \quad = force\ x & & \bot \\
\qquad\quad updateInt_\#\ r_1\ w_1\ u_2\ u_1 & & Write\ r_1 \\
\qquad\quad z_1 \quad = force\ y & & \bot \\
\qquad\quad z_2 \quad = unbox\ r_3\ z_1 & & Read\ r_3 \\
\qquad\quad z_3 \quad = succ_\#\ z_2 & & \bot \\
\qquad\quad box\ r_2\ z_3 & & \bot
\end{array}
$$

We have recorded the effect of each binding on the right. The majority of these bindings are pure, so their position is constrained only by the data dependencies in the program. A special case is the binding for x_2. From its effect we see that it reads the x value from the region named r_1. This interferes with the update operation, which writes to r_1.

Note that the binding for x_1 is a duplicate of u_2, so we can remove the second and substitute $u_2 = x_1$ into successive bindings. We can then float all bindings except x_1 and x_2 into their use sites. We do not float x_1 as it now has two bound occurrences, and we do not float x_2 as this would require moving it across the interfering update expression:

$$
\begin{array}{l}
succUpdate\ x \\
= \quad \Lambda\ r_1\ r_2\ (w_1 : Mutable\ r_1). \\
\quad \lambda\ x\ \ : Int\ r_1. \\
\quad \textbf{letregion}\ r_3\ \textbf{with}\ \{w_3 = MkConst\ r_3\}\ \textbf{in} \\
\quad \textbf{letregion}\ r_4\ \textbf{with}\ \{w_4 = MkConst\ r_4\}\ \textbf{in} \\
\quad \textbf{do} \quad x_1 \quad = force\ x \\
\qquad\quad x_2 \quad = unbox\ r_1\ x_1 \\
\qquad\quad updateInt_\#\ r_1\ w_1\ x_1\ 5_\# \\
\qquad\quad box\ r_2\ (succ_\#\ (unbox\ r_3\ (force\ (box\ r_3\ (succ_\#\ x_2)))))
\end{array}
$$

Now, in the last statement we can eliminate the use of *force*, because a freshly boxed integer is guaranteed not to be a thunk. We can then eliminate the *unbox box* pair as well. This gives:

$$
\begin{array}{lll}
& \mathit{succUpdate}\ x & \\
= & \Lambda\ r_1\ r_2\ (w_1 : \mathit{Mutable}\ r_1). & \\
& \lambda\ x\ : \mathit{Int}\ r_1. & \\
& \textbf{letregion}\ r_3\ \textbf{with}\ \{w_3 = \mathit{MkConst}\ r_3\}\ \textbf{in} & \\
& \textbf{letregion}\ r_4\ \textbf{with}\ \{w_4 = \mathit{MkConst}\ r_4\}\ \textbf{in} & \\
& \textbf{do} \quad x_1 \ = \mathit{force}\ x & \bot \\
& \qquad x_2 \ = \mathit{unbox}\ r_1\ x_1 & \mathit{Read}\ r_1 \\
& \qquad \mathit{updateInt}_{\#}\ r_1\ w_1\ x_1\ 5_{\#} & \mathit{Write}\ r_1 \\
& \qquad \mathit{box}\ r_2\ (\mathit{succ}_{\#}\ (\mathit{succ}_{\#}\ x_2)) & \bot
\end{array}
$$

Compared to the original version, we have eliminated two uses each of *box*, *unbox* and *force*. If we were to constrain the original type of *succUpdate* so that its parameter was *Direct*, then we could eliminate the remaining use of *force* as well. Note that although the position of the x_2 binding is not constrained by data dependencies, it *is* constrained by the interfering effect of the update statement.

4.4.3 Effects and region aliasing

In the *succUpdate* example of the previous section, we could tell that the unboxing and update operations interfered because their effects mentioned the same region variable. If two atomic effects mention *different* region variables, then we must consider whether the corresponding objects may alias when deciding whether the effects interfere. For example, say we had a function of the following type:

$$
\begin{array}{lll}
\mathit{fun} & :: & \forall\ r_1\ r_2 \\
& . & \mathit{Int}\ r_1 \rightarrow \mathit{Int}\ r_2 \xrightarrow{e_1} \ldots \\
& \rhd & e_1 = \ldots \\
& , & \mathit{Mutable}\ r_1
\end{array}
$$

The first part of its definition in the core language could well be:

$$
\begin{array}{ll}
& \mathit{fun} \\
= & \Lambda\ r_1\ r_2\ (w_1 : \mathit{Mutable}\ r_1). \\
& \lambda\ x : \mathit{Int}\ r_1. \\
& \lambda\ y : \mathit{Int}\ r_2. \\
& \textbf{letregion}\ r_3\ \textbf{with}\ \ldots\ \textbf{in}. \\
& \textbf{letregion}\ r_4\ \textbf{with}\ \ldots\ \textbf{in}. \\
& \mathit{exp}
\end{array}
$$

Assume that *exp* is some interesting expression that reads the values of x and y, and updates x in the process. Now, there is nothing preventing the programmer from calling *fun* with the same object for both arguments:

$$\textbf{let}\ x = 5\ \textbf{in}\ \mathit{fun}\ x\ x$$

Because of this, when transforming *exp*, we cannot assume that two effects $\mathit{Read}\ r_2$ and $\mathit{Write}\ r_1$ do not interfere. On the other hand, effects on r_3 and r_4 cannot interfere because they have been introduced by the function itself, and

are known to be distinct. The type system ensures that objects in the region named r_3 are distinct from those that are in the region named r_4. Likewise, effects on r_1 and r_3, or r_1 and r_4 cannot interfere because an object with type $Int\ r_1$ must have been allocated by the caller, and thus cannot alias with locally allocated objects.

However, suppose r_1 and r_2 were constrained to have differing mutabilities:

$$\begin{array}{lll} fun & :: & \forall r_1\ r_2 \\ & . & Int\ r_1 \rightarrow Int\ r_2 \xrightarrow{e_1} ... \\ & \triangleright & e_1 = ... \\ & , & Mutable\ r_1 \\ & , & Const\ r_2 \end{array}$$

In this case the first part of the function definition could be:

$$\begin{array}{ll} fun & \\ = & \Lambda\ r_1\ r_2\ (w_1 : Mutable\ r_1)\ (w_2 : Const\ r_2). \\ & \lambda\ x : Int\ r_1. \\ & \lambda\ y : Int\ r_2. \\ & \textbf{letregion}\ r_3\ \textbf{with}\ ...\ \textbf{in}. \\ & \textbf{letregion}\ r_4\ \textbf{with}\ ...\ \textbf{in}. \\ & exp \end{array}$$

With this new definition the two effects $Read\ r_2$ and $Write\ r_1$ are guaranteed not to interfere. The caller cannot pass the same object for both parameters because it cannot produce witnesses of mutability and constancy for the same region variable.

In future work we intend to use this line of reasoning to extend the language with *NoAlias* witnesses. This is discussed in §5.2.5.

4.4.4 Floating into alternatives

Consider the following program:

$$\begin{array}{ll} \textbf{do} & y = f\ x \\ & \textbf{if}\ exp\ \textbf{then}\ y\ \textbf{else}\ ... \end{array}$$

As Disciple uses call-by-value evaluation by default, the application $f\ x$ will always be evaluated. Note that in DDC, before we try to float bindings into alternatives we transform the program to administrative normal form. In this form the terms involved in an application are either variables or constants.

In the above example, if $f\ x$ and *exp* are pure, or they only *read* mutable data, then it is safe to move the y binding into the alternative to give:

$$\textbf{if}\ exp\ \textbf{then}\ f\ x\ \textbf{else}\ ...$$

This eliminates the need to evaluate $f\ x$ in the event the second branch of the if-expression is taken. Note that if the first alternative contains other function applications, then we need to consider whether $f\ x$ can interfere with them.

For example:

```
do  (x1 : Int r1) = 5 r1                    ⊥
    (x2 : Int r2) = succ x1                 Read r1
    if ...                                  ⊥
      then do
           x1 := 23 r3                      Write r1 ∨ Read r3
           succ x2                          Read r2
      else 42 r2                            ⊥
```

Recall that in the core language we use literal integers such as 5 as constructors, so $(5\ r_1)$ is equivalent to $box\ r_1\ 5_\#$. We have also added type annotations to the binding occurrences of variables to make the example clearer. Note that we cannot move the x_2 binding into its use site because it interferes with the expression $x_1 := 23\ r_1$. However, we *can* move the x_2 binding into the first alternative of the if-expression, so long as we place it before the update:

```
do  (x1 : Int r1) = 5 r1                    ⊥
    if ...                                  ⊥
      then do
           (x2 : Int r2) = succ x1          Read r1
           x1 := 23 r3                      Write r1 ∨ Read r3
           succ x2                          Read r2
      else 42 r2                            ⊥
```

If a binding causes a top level effect then we cannot move it across another. Likewise, we cannot move such a binding inwards, as that would tend to reduce the number of times it was evaluated. For example:

```
do  x1 = do { putStr "hello"; 5 r1 }        Console
    if ...                                  ⊥
      then succ x1                          Read r1
      else 42 r1                            ⊥
```

4.4.5 Floating outside lambda abstractions

Floating bindings outside of lambda abstractions, also known as the *full laziness transform*, allows us to share the result of a computation between calls to a function. This is similar to the "lifting expressions out of loops" optimisation done in compilers for imperative languages.

For example, consider the following program:

```
Λ r2 r3.
letregion r1 with {w = Const r1} in
do  (xs : List r1 (Int r3)) = ...           ⊥
    f = λ(y : Int r2). do
           (n : Int r3) = length xs         ⊥
           n + y                            Read r2 ∨ Read r3
    z1 = f (5 r2)                           Read r2 ∨ Read r3
    ...
    z2 = f (23 r2)                          Read r2 ∨ Read r3
```

As the value of n does not depend on the bound variable y, we can lift it out of the enclosing λ-abstraction. This eliminates the need to recompute it for each application of f:

$$
\begin{array}{lll}
\Lambda\ r_2\ r_3. & & \\
\textbf{letregion}\ r_1\ \textbf{with}\ \{w = \mathit{Const}\ r_1\}\ \textbf{in} & & \\
\textbf{do} \quad (\mathit{xs} : \mathit{List}\ r_1\ (\mathit{Int}\ r_3)) = ... & & \bot \\
\quad (n : \mathit{Int}\ r_3) = \mathit{length}\ \mathit{xs} & & \bot \\
\quad f = \lambda(y : \mathit{Int}\ r_2).\ n + y & & \mathit{Read}\ r_2 \vee \mathit{Read}\ r_3 \\
\quad z_1 = f\ (5\ r_2) & & \mathit{Read}\ r_2 \vee \mathit{Read}\ r_3 \\
\quad ... & & \\
\quad z_2 = f\ (23\ r_2) & & \mathit{Read}\ r_2 \vee \mathit{Read}\ r_3
\end{array}
$$

Of course, in general this is only valid if the lifted expression is pure. Here, we must guarantee that the length of the list is not destructively changed between each application of f. For this example, the purity of the n binding is guaranteed by the constancy of r_1, which is witnessed by w.

Only lift bindings that produce constant results

As it is only safe to increase the sharing of constant data, we must insure that the results of lifted bindings are constant. Here is an example where a binding is pure, and independent of the λ-bound variable, but it is not safe to float it outwards:

$$
\begin{array}{l}
\Lambda\ r_4\ r_6. \\
\textbf{letregion}\ r_5\ \textbf{with}\ \{w = \mathit{Mutable}\ r_5\}\ \textbf{in} \\
\textbf{do} \quad (\mathit{xs} : \mathit{List}\ ...) \quad = ... \\
\quad (\mathit{ys} : \mathit{List}\ r_4\ (\mathit{Int}\ r_5)) \\
\qquad\qquad = \mathit{map}\ (\lambda\_.\ \textbf{do}\ \{\ (m : \mathit{Int}\ r_5) = \mathit{succ}\ 0;\ m\ \})\ \mathit{xs} \\
\quad \mathit{updateInt}\ r_5\ r_6\ w\ (\mathit{ys}\ !!\ 2)\ (5\ r_6) \\
\quad (\mathit{ys}\ !!\ 3)
\end{array}
$$

Where *updateInt* has type:

$$\mathit{updateInt} :: \forall\ r_1\ r_2.\ \mathit{Mutable}\, r_1 \Rightarrow \mathit{Int}\ r_1 \rightarrow \mathit{Int}\ r_2\ ^{\mathit{Read}\ r_2 \vee \mathit{Write}\ r_1}\!\!\longrightarrow ()$$

The operator !! is used to retrieve a numbered element of the list. This example creates a new list of integers, *ys*, which is the same length as the original list *xs*. It then updates the second element of *ys*, and returns the third. Note that the m binding is pure, but as *succ* allocates its result, each element of the list *ys* will be represented by a different run-time object. Even though we update the second element, the third element will still have the value $\mathit{succ}\ 0 = 1$.

If we were to erroneously lift the m binding out of the lambda abstraction, this would cause the same object to be used for every element of the list:

$\Lambda\ r_4\ r_6.$
letregion r_5 **with** $\{\ w = \mathit{Mutable}\ r_5\ \}$ **in**
do $(\mathit{xs} : \mathit{List}\ \ldots) = \ldots$
$(m : \mathit{Int}\ r_5) = \mathit{succ}\ 0$
$(\mathit{ys} : \mathit{List}\ r_4\ (\mathit{Int}\ r_5))$
$= \mathit{map}\ (\lambda\_.\ m)\ \mathit{xs}$
$\mathit{updateInt}\ r_5\ r_6\ w\ (\mathit{ys}\ !!\ 2)\ (5\ r_6)$
$(\mathit{ys}\ !!\ 3)$

In this case, when we update the second element of the list, this is the same as updating the third element as well, so we have changed the meaning of the program.

Suspend lifted bindings to reduce wasted computation

If we cannot guarantee that a particular λ-abstraction is applied at least once, then we should suspend the evaluation of any bindings that are lifted from it. This guards against the case where the abstraction is never applied, or the evaluation of the binding does not terminate. For example:

$\Lambda\ r_3.$
letregion r_4 **with** $\{\ w = \mathit{Const}\ r_4\ \}$ **in**
do $(\mathit{xs} : \mathit{List}\ r_4\ (\mathit{Int}\ r_3))$
$= \ldots$
$f = \lambda y.$ **do** $\{\ n\ = \mathit{length}\ \mathit{xs};\ n + y\ \}$
$g = \ldots$
if ...
then $f\ 5\ +\ f\ 23$
else $g\ 42$

As $\mathit{length}\ \mathit{xs}$ is pure, we can lift the n binding out of the abstraction. This will save it being re-evaluated for each occurrence of f. However, if the **else** branch of the if-expression is taken, then the value of n won't be needed. Due to this we should suspend the function application that produces it:

$\Lambda\ r_3.$
letregion r_4 **with** $\{\ w = \mathit{Const}\ r_4\ \}$ **in**
do $(\mathit{xs} : \mathit{List}\ r_4\ (\mathit{Int}\ r_3))$
$= \ldots$
$n = \mathit{suspend1}\ (\mathit{MkPurify}\ r_4\ w)\ \mathit{length}\ \mathit{xs}$
$f = \lambda y.\ n + y$
$g = \ldots$
if ...
then $f\ 5\ +\ f\ 23$
else $g\ 42$

In practice, we only want to introduce one thunk per binding. If the right of the binding is something other than a single function application, then we can wrap it in a dummy lambda abstraction and suspend that instead. For example, if the right of the n binding was actually $succ\ (length\ xs)$ then we could translate our original example to:

$$
\begin{array}{l}
\Lambda\ r_3. \\
\textbf{letregion}\ r_4\ \textbf{with}\ \{\ w = Const\ \ r_4\ \}\ \textbf{in} \\
\textbf{do}\quad (xs : List\ \ r_4\ (Int\ \ r_3)) \\
\qquad\quad = ... \\
\qquad n \quad = suspend1 \quad (MkPureJoin\ (MkPurify\ \ r_4\ \ w)\ ...) \\
\qquad\qquad\qquad\quad\ \ (\lambda\_.\ succ\ (length\ \ xs)) \\
\qquad\qquad\qquad\quad\ \ () \\
\qquad f \quad = \lambda y.\ n + y \\
\qquad g \quad = ... \\
\qquad \textbf{if}\ ... \\
\qquad\quad \textbf{then}\quad f\ 5\ +\ f\ 23 \\
\qquad\quad \textbf{else}\quad g\ 42
\end{array}
$$

Note that as we only lift pure bindings, we should always be able to create witnesses of purity for those bindings. This is an example of type information serving as an internal sanity check, rather than being used to guide optimisations. If we cannot create a witness of purity for a lifted binding, then there is a bug in our compiler implementation.

4.5 Comparisons

4.5.1 Monadic intermediate languages. 1998 Tolmach, Benton, Kennedy, Russell.

One of the main inspirations for our work has been to build on the monadic intermediate languages of [Tol98], [BK99] and [PJSLT98]. The system of [Tol98] uses a coarse grained effect analysis to guide the translation of the source program into a core language incorporating a hierarchy of monadic types. The monads are `ID`, `LIFT`, `EXN`, and `ST`. Starting from the bottom of the hierarchy: `ID` describes pure, terminating computations; `LIFT` encapsulates pure but potentially non-terminating computations; `EXN` encapsulates potentially non-terminating computations that may raise uncaught exceptions, and `ST` encapsulates computations that may do everything including talk to the outside world.

The optimisations in [Tol98] are given as transform rules on monadic terms, and less transforms apply to expressions written with the more effectual monads. Limitations of this system include the fact that it lacks effect polymorphism, and the coarseness of the hierarchy. In the last part of [Tol98], Tolmach suggests that it would be natural to extend his system with Hindley-Milner style polymorphism for both types and monads in the Talpin-Jouvelot style. He also suggests that it would extend naturally to a collection of fine-grained monads encapsulating primitive effects, but laments the lack of a generic mechanism for combining such monads.

Monads and effects express equivalent information

In [WT03], Wadler and Thiemann compare the effect typing and monadic systems, and give a translation from the first to the second. For their monadic system, they write the types of computations as $\mathrm{T}^{\sigma}\ a$, where a is the type of the resulting value and σ is a set of store effects. They consider store effects such as *Read* r_1 and *Write* r_2, use $\vee$ to collect atomic effect terms, and include type schemes that quantify over type, region and effect variables. Clearly, their monadic system shares a lot of common ground with an effect system. The main technical difference between the two is that the monadic version of the typing rule for applications is broken into two parts:

Whereas in the effect system we have:

$$\frac{\Gamma \vdash t_1 :: \tau_{11} \xrightarrow{\sigma_3} \tau_{12} \ ; \ \sigma_1 \quad \Gamma \vdash t_2 :: \tau_{11} \ ; \ \sigma_2}{\Gamma \vdash t_1\ t_2 :: \tau_{12} \ ; \ \sigma_1 \vee \sigma_2 \vee \sigma_3} \qquad \text{(EffApp)}$$

In the monadic system we have:

$$\frac{\Gamma \vdash t_1 :: \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash t_2 :: \tau_{11}}{\Gamma \vdash t_1\ t_2 :: \tau_{12}} \qquad \text{(MonApp)}$$

$$\frac{\Gamma \vdash t_1 : \mathrm{T}^{\sigma_1}\ \tau_1 \quad \Gamma,\ x : \tau_1 \vdash t_2 : \mathrm{T}^{\sigma_2}\ \tau_2}{\Gamma \vdash \ \textbf{let}\ x \leftarrow t_1\ \textbf{in}\ t_2 :: \mathrm{T}^{\sigma_1 \vee \sigma_2}\ \tau_2} \qquad \text{(MonBind)}$$

In the effect typing system, effects are caused by the application of functions, as well as by the evaluation of primitive operators such as *readRef* and *writeRef*. In the monadic system, all effects are invoked explicitly with the **let** $x \leftarrow t_1$ **in** t_2 form, which evaluates the computation t_1, and then substitutes the resulting value into t_2. Function application of the form $t_1\ t_2$ is always pure.

Expressing T-monads in Disciple

Note that $\mathrm{T}^{\sigma}\ a$ style computation types are straightforward to express in Disciple, because we can define data types that have effect parameters. For example, eliding region and closure information we can write:

$$\textbf{data}\ T\ e_1\ a = MkT\ (() \xrightarrow{e_1} a)$$

Our $T\ e_1\ a$ data type simply encapsulates a function that produces a value of type a when applied to the unit value (), while having an effect e_1. The monadic return and bind operators are defined as follows:

$$\begin{aligned}
&returnT \quad :: \quad \forall a.\ a \rightarrow T \perp a \\
&returnT\ \ x \ = MkT\ (\lambda().\ x)
\end{aligned}$$

$$\begin{aligned}
&bindT \quad :: \quad \forall a\ b\ e_1\ e_2 \\
&\qquad\quad .\quad T\ e_1\ a \rightarrow (a \rightarrow T\ e_2\ b) \rightarrow T\ e_3\ b \\
&\qquad\quad \rhd \quad e_3 = e_1 \vee e_2
\end{aligned}$$

$$\begin{aligned}
&bindT\ (MkT\ \ f_1)\ \ mf_2 \\
&\quad = MkT\ (\lambda().\ \textbf{case}\ \ mf_2\ (f_1\ ())\ \textbf{of} \\
&\qquad\qquad\qquad MkT\ \ f_2 \rightarrow f_2\ ())
\end{aligned}$$

Although we can directly express T monads in a language with an effect system, the reverse is not true. A monadic system requires all effects to be encapsulated within a computation type such as T, and the function arrow, $\rightarrow$, must be pure. However, an effect system allows arbitrary function applications to have effects, and we can add these effects as annotations to the arrows, $\xrightarrow{\sigma}$.

What's more natural?

In [BK99] Benton and Kennedy suggest that "the monadic style takes the distinction between computations and values more seriously", and that it has a more well-behaved equational theory. However, their work has different goals to ours. On one hand, [BK99] includes rigorous proofs that their optimising transforms are correct. For this purpose, we can appreciate how reducing effect invocation to a single place in the language would make it easier to reason about. Their system was implemented in the MLj [BKR98] compiler, so it is demonstrably practical. In [BKBH07] Benton *et al* consider the semantics of a similar system extended with region variables and effect masking, and in [BB07] Benton and Buchlovsky present the semantics of an effect based analysis for exceptions. On the other hand, [BKR98] does not include effect polymorphism, and the more recent work of [BKBH07] and [BB07] does not discuss type inference and has not yet been implemented in a compiler.

For our purposes, we find it more natural to think of function application and primitive operators as causing effects, as this is closer to the operational reality. After spending time writing a compiler for a language that includes laziness, we don't feel too strongly about the distinction between computations and values. When we sleep we dream about thunks, and the fact that the inspection of a lazy "value" of type *Int* may diverge is precisely because that value represents a possibly suspended computation.

If we were going to follow Benton and Kennedy's approach then we would write $\mathrm{T}^{\mathtt{LIFT}}\ \mathit{Int}$ for the lazy case and *Int* for the direct one. Using (MonBind) above, this would have the benefit that the potential non-termination of lazy computations would be propagated into the types of terms that use them. However, for the reasons discussed in §1.4 we don't actually treat non-termination as a computational effect. We also remain unconvinced of the utility of introducing a separate monadic binding form into the core language, at least in the concrete implementation. Horses for courses.

4.5.2 System-Fc. 2007
Sulzmann, Chakravarty, Peyton Jones, Donnelly.

The core language of GHC is based on System-Fc [SCPJD07], which uses type equality witnesses to support generalised algebraic data types (GADTs) [XCC03] and associated types [CKJM05]. The kinds of such witnesses are written $a \sim b$, which express the fact that type a can be taken as being equivalent to type b. The witnesses express non-syntactic type equalities, which are a major feature of the work on GADTs and associated types.

The witness passing mechanism in DDC was inspired by an earlier draft of [SCPJD07] that included the dependent kind abstraction $\Pi a : \kappa_1.\ \kappa_2$.

In this draft, abstraction was used to write the kinds of polymorphic witness constructors such as:

$$\mathit{elemList} :: \Pi a : *.\ \mathit{Elem}\ [a] \sim a$$

Here, *Elem* is the constructor of an associated type. The kind of *elemList* says that elements of a list of type a have type a. In the published version of the paper, extra typing rules were introduced to compose and decompose types that include equality constraints, and these new rules subsumed the need for an explicit dependent kind abstraction. In the published version, the type of *elemList* is written:

$$\mathit{elemList} :: (\forall a : *.\ \mathit{Elem}\ [a]) \sim (\forall a : *.\ a)$$

Note that when this type is instantiated, the type argument is substituted for both bound variables. For example:

$$\mathit{elemList}\ \ \mathit{Int} :: \mathit{Elem}\ [\mathit{Int}] \sim \mathit{Int}$$

The dependent kind abstraction is still there in spirit, but the syntax has changed. System-Fc includes witness constructors such as *sym*, *trans*, *left* and *right* whose kinds express logical properties such as the symmetry and transitivity of the type equality relation, as well as providing decomposition rules. Although [SCPJD07] gives typing rules for these constructors, if we were prepared to limit their applicability to first order kinds then we could also express them with the dependent kind abstraction. For example:

$$\frac{\Gamma \vdash \varphi : \tau_1 \sim \tau_2}{\Gamma \vdash \mathit{sym}\ \ \varphi : \tau_2 \sim \tau_1} \qquad \text{(Sym)}$$

would become:

$$\mathit{sym} :: \Pi(a : *).\ \Pi(b : *).\ a \sim b \rightarrow b \sim a$$

Adding kind abstraction to the system would allow us to remove the restriction to first order kinds, and regain the full expressiveness of the original rules:

$$\mathit{sym} :: \lambda(k : \Box).\ \Pi(a : k).\ \Pi(b : k).\ a \sim b \rightarrow b \sim a$$

Here, the superkind $\Box$ restricts k to be something like $*$ or $* \rightarrow *$, and not another witness kind.

Note that the System-Fc witness constructors such as *sym*, and the DDC witness constructors such as *MkPureJoin* are of the same breed. They both express logical properties of the specific system, which are separate from the underlying LF [AHM89] style framework. It would be interesting to see how well both systems could be expressed in a more general one, such as Ωmega [She05], which has extensible kinds.

Chapter 5

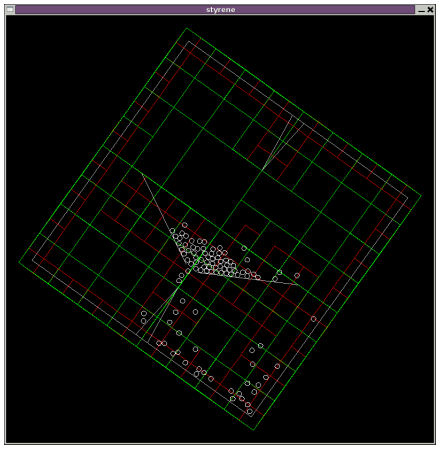
Conclusion

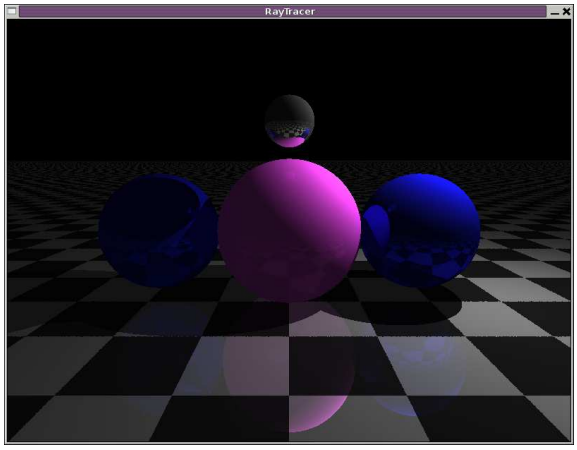
The work presented in this thesis is embodied in the Disciplined Disciple Compiler (DDC) which can be obtained from the `haskell.org` website. We have found it invaluable to develop the compiler alongside the source language and type system. Having a real compiler allows us to experiment with example programs that would be impractical to manipulate by hand, and we have been working on DDC since the very start of this project.

DDC is not yet 'industrial strength', but it does have enough functionality to compile non-trivial programs. Screenshots from two of our test programs are below. The one on the left is a real-time, 2-dimensional particle collision simulation that uses a quad-tree to determine when two particles are close enough to possibly collide. The second is a simple ray tracer which uses a vector library based on our projection system.

Developing these programs has given us insight into some of the strengths and weaknesses of our current system. In this chapter we briefly discuss our back-end implementation, identify opportunities for further work, summarise our contributions, and conclude.

Gratuitous Screenshots





5.1 Implementation

DDC is written in Haskell with GHC extensions. It uses three intermediate representations: a desugared form of the source language; the core language presented in this thesis, and an abstract C-like language. We have used Parsec [LM01] for the parser, and compile to ANSI C.

As the focus of our work has been on the type system and core language, we have not put a substantial amount of work into optimising the back end. However, we have gleaned a few points that may be of interest to others embarking on a similar endeavor. Although compiling via third party intermediate languages such as C−− [PJNO97] or LLVM [LA04] is likely to produce better code in the long run, we feel that targeting C still has a place if the developer "just wants to get something working". The primary benefits are that a given developer will invariably know C already, and that implementions of primitive functions can be written directly.

5.1.1 Implementing thunks and laziness

As Disciple uses call-by-value evaluation as default, we expect laziness to only be used occasionally. We desire straightforward, C-like programs that do not make heavy use of higher order functions or partial application to run as fast as if they were actually written in C. For this reason we avoid the heavy encodings that are associated with compiling via an abstract machine, such as the STG machine [PJ92].

After the core-level optimisations are finished, we perform lambda lifting to generate supercombinators [Hug83]. Each supercombinator is translated to a single C function. We handle partial application by building a thunk containing a pointer to the associated supercombinator, along with the provided arguments. This is the eval/apply method discussed in [MPJ04].

Thunks representing suspended computations are created with explicit calls to the *suspend* function that was introduced in §2.3.9. Thunks that represent numeric values are forced by the *force* function. Calls to *force* are introduced by the compiler during the local unboxing optimisation that was discussed in §4.4.2. We use `switch` statements to implement core-level case-expressions, and thunks that represent values of algebraic type are forced by extra alternatives that we add to these statements.

For example, the following expression:

case *xx* **of**
Nil → ... *alt1* ...
Cons x xs → ... *alt2* ...

compiles to:

```
again:
switch (_TAG(xx)) {
  case tag_Nil:   goto alt1;
  case tag_Cons:  goto alt2;
  case tag_INDIR: xx = ((Thunk*)xx)->next; goto again;
  case tag_SUSP:  xx = force(xx);          goto again;
  default:        ... error handling ...
}
```

The tags `INDIR` and `SUSP` are common to all objects. When the tag of an object is inspected, if it turns out to be an indirection or suspension then it is followed or forced appropriately. The object is then reinspected by jumping back to the start of the `switch` statement. Note that we place the alternatives for indirections and suspensions last in the list. This ensures that the handling of lazy objects does not degrade the speed of programs that use mostly call-by-value evaluation. If the type of the object to be inspected is constrained to be direct, then we can omit the `INDIR` and `SUSP` alternatives and gain a slight speedup due to a smaller executable.

The primary advantages of this method are its simplicity and portability. The disadvantage is that we incur a function call overhead every time we force a thunk. This method is unlikely to ever match the efficiency a purpose built system based on the STG machine [PJ92], but it works, and is significantly easier to implement.

5.2 Limitations and possible improvements

This section discusses some of the limitations that we have uncovered in our current system. Although we present ideas for addressing these limitations, they have not yet been fully developed or implemented. The limitations are presented in order, with the ones that we feel would most affect client programmers listed first. Most of these limitations were introduced earlier in this thesis, and we elaborate on them here as a guide for future work.

5.2.1 Masking mutability constraints

As mentioned in §2.3.7, although we can mask effects on region variables that correspond to fresh objects, we cannot mask mutability constraints on the same variables. The classic example is a function that destructively updates a counter when calculating the length of a list, and then returns the counter. With effect masking, our current system gives this function the following type:

$$\begin{array}{lll} \mathit{length} & :: & \forall a\ r_1\ r_2.\ \mathit{List}\ \ r_1\ a \xrightarrow{e_1} \mathit{Int}\ \ r_2 \\ & \rhd & \mathit{Read}\ \ r_1 \\ & , & \mathit{Mutable}\ \ r_2 \end{array}$$

For this example we can work around the problem by copying the counter value before returning it. This invokes the *Shape* constraint discussed in §2.6.2, and allows the resulting value to have a differing mutability. We expect to solve the general problem by introducing closure information into the core language, and then using a similar mutability masking mechanism to the one outlined by Gupta in [Gup95].

5.2.2 Blocked regions and region sums

The following program is rejected by our system:

$$
\begin{array}{l}
x = 5 \\
\mathit{fun}\ () \\
\quad = \textbf{do} \quad y = 23 \\
\qquad\qquad\ \ldots \\
\qquad\qquad\ y := 42 \\
\qquad\qquad\ \textbf{if}\ \ldots\ \textbf{then}\ x\ \textbf{else}\ y
\end{array}
$$

As x is defined at top level it defaults to being constant, and as y is updated it must be mutable. However, both x and y are returned by the if-expression, so our system requires them to have the same type. This creates a mutability conflict, so the program is rejected.

Our work-around is to explicitly copy either x, y, or both. This solves the immediate problem, but is clumsy and could result in a considerable run-time overhead when dealing with larger structures.

Note that if we could guarantee that the result of the if-expression was treated as *neither* constant nor mutable, then we could allow the above program. As discussed in §2.3.13, we view mutability as the capability of an object to be updated, and constancy as the capability of suspending a computation that reads it. We have in mind to introduce a new constraint, *Blocked*, which prevents a region variable from being additionally constrained as either *Const* or *Mutable*.

However, we do not want the *Blocked* constraint to 'leak' into the type of x. The fact that we choose between x and a mutable value does not mean there is any danger of suspending a function that reads x directly. Say x and y have the following types:

$$
\begin{array}{ll}
x & :: \mathit{Int}\ \ r_1 \rhd \mathit{Const}\ \ r_1 \\
y & :: \mathit{Int}\ \ r_2 \rhd \mathit{Mutable}\ \ r_2
\end{array}
$$

We would prefer to leave r_1 and r_2 as constant and mutable respectively, and use a *region sum* to express the fact that the result of *fun* could be in either region r_1 or r_2. The full type of *fun* would then be something like:

$$
\begin{array}{lll}
\mathit{fun} & :: & \forall\ c_1\ r_2\ r_3.\ () \xrightarrow{c_1} \mathit{Int}\ \ r_3 \\
 & \rhd & c_1 \quad \sqsupseteq x : \mathit{Int}\ \ r_1 \\
 & , & r_3 \quad \sqsupseteq r_1 \vee r_2 \\
 & , & \mathit{Const}\ \ r_1 \\
 & , & \mathit{Blocked}\ \ r_3
\end{array}
$$

The constraint $r_3 \sqsupseteq r_1 \vee r_2$ is read: "r_3 might include objects from r_1 or r_2", with the "or" being non-exclusive. The literature on union typing, such as [DP03], may provide clues on how to implement this.

5.2.3 Bounded quantification and effect strengthening

In section §4.3.4 we discussed bounded quantification and how it is included in our core language. We added this feature so that we can directly express the types inferred by our Talpin-Jouvelot style inference algorithm. However, we are not convinced that bounded quantification (beyond type class constraints)

is intrinsically necessary. We feel this way because there is never an operational need for the effect of one parameter function to be larger than another.

As discussed in §2.3.6, effect variables serve to propagate the effect of a parameter into to the manifest effect of the overall function. This mechanism is used to express the fact that a higher order function may invoke its parameter, and our optimising transforms must be aware of this. The only time we need to constrain the effect of a parameter function is when we suspend an application of it, and we do this with purity constraints via the type classing system.

The fact that a function such as *foo* from §2.3.6 has a $\sqsupseteq$ constraint on its parameter effect is an artefact of the bidirectional nature of Hindley-Milner type inference. This is related to the poisoning problem which is mentioned in [BK99] and discussed in [WPJ99]. Both [WPJ99] and [BK99] report that subtyping can be used to remedy this problem, but [WPJ99] concerns usage analysis and not effects, and [BK99] does not consider effect polymorphism. In [BKBH07] Benton *et al* extend the system presented in [BK99] with region variables, but do not discuss type inference.

More work is needed to determine whether the effect (and closure) constraints on the types of higher order functions can always be strengthened. The goal would be to either eliminate the need for bounded quantification in the core language, or to determine why this can not, or should not, be done.

5.2.4 Polymorphism, data sharing, and constraint masking

Suppose we wish to define a type class to help compute the areas of geometric figures. The obvious definition would be:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Area}\ \ a\ \textbf{where} \\
\quad \mathit{area} \quad :: \quad \forall r_1.\ a \xrightarrow{e_1} \mathit{Float}\ \ r_1 \\
\qquad\quad\ \ \rhd \quad e_1 = \mathit{ReadT}\ \ a
\end{array}
$$

This definition says that an instance of the *area* function produces a float into the fresh region named r_1, and is permitted to read its argument. Here is a data type to represent our figures:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Figure}\ \ r_1 \\
\quad = \quad \mathit{Rectangle}\ (\mathit{Float}\ \ r_1)\ (\mathit{Float}\ \ r_1) \\
\quad | \quad \mathit{Circle}\ (\mathit{Float}\ \ r_1)
\end{array}
$$

The parameters of *Rectangle* are its width and height, and the parameter of *Circle* is its radius. The area of a rectangle is its width multiplied by its height, and the area of a circle is π times its radius squared:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Area}\ (\mathit{Figure}\ \ r_1)\ \textbf{where} \\
\quad \mathit{area}\ \ \mathit{fig} \\
\quad = \textbf{case}\ \mathit{fig}\ \ \textbf{of} \\
\qquad\quad \mathit{Rectangle}\ \ w\ \ h \quad \rightarrow w\ *\ h \\
\qquad\quad \mathit{Circle}\ \ r \qquad\quad \rightarrow \mathit{pi}\ *\ r^2
\end{array}
$$

Unfortunately, this is not a valid instance of *Area*. The trouble is that the constant *pi* is free in the closure of *area*, but this information is not present in the class definition. If we assume *pi* is defined at top level and has the type $\mathit{Float}\ \ r_5$, then our instance function has type:

$$\begin{array}{lll} area_{Figure} & :: & \forall r_1\ r_2.\, Figure\ \ r_1 \xrightarrow{e_1\ c_1} Float\ \ r_2 \\ & \rhd & e_1 = Read\ \ r_1 \\ & , & c_1 = pi : Float\ \ r_5 \end{array}$$

Although we *could* go back and widen the class definition so that all instances of *area* are assumed to refer to *pi*, this is an unsatisfying solution. How could we decide *a priori* which constants would be needed? For example, the surface area of a torus is $4\pi^2 Rr$, where r and R are the inner and outer radii. When determining such an area we would prefer to use a constant *pi2* instead of computing the value of π^2 each time. Should we include just *pi* and *pi2* in the class definition, or will we need other constants as well?

In essence, our current system exposes too much useful information about the sharing properties of data. Although we need this information to reason about mutability and to perform effect masking, we sometimes want to ignore it in the interests of polymorphism. Using the language of [MNM04], it is nonlinearity and 'amnesia' which make a type system work. A given system must be able to forget about the exact details of the program, otherwise testing for well-typedness degenerates to simply running the program and checking for failure.

A seemingly obvious solution is to erase closure terms that refer to constant regions, but we have to carefully consider possible interactions with other extensions to the language. For example, if we allow masking of *Mutable* constraints as per §5.2.1, then what about masking of *Const* constraints as well? For example, say we have a function of the following type:

$$\begin{array}{lll} fun & :: & () \xrightarrow{c_1} Float\ \ r_1 \\ & \rhd & c_1 = x : Float\ \ r_1 \\ & , & Const\ \ r_1 \end{array}$$

As r_1 is constant, we could argue that the constraint on c_1 should be erased. Assuming that x is not in the type environment, this would also allow r_1 to be generalised:

$$\begin{array}{lll} fun & :: & \forall r_1.\, () \rightarrow Float\ \ r_1 \\ & , & Const\ \ r_1 \end{array}$$

Now, this type doesn't look too different from the type of *length* in §5.2.1. If we followed the reasoning presented there then we might like to (erroneously) mask the constancy constraint as well:

$$fun_{bad} \quad :: \quad \forall r_1.\, () \rightarrow Float\ \ r_1$$

However, this in invalid. With this type there is nothing stopping us from updating the return value. The point is that a constancy constraint does not *just* mean that a given object should not be updated, it means that it should not be updated because other expressions may refer to it in ways that are not visible in our type information.

This brings us back to our discussion of region allocation from §4.2.8. Recall that we do not currently use regions for allocation, or more importantly *deallocation*. This is because a 'value' of type $Int\ \ r_1$ may be represented by a thunk, and that thunk may hold references to objects in regions which are not present in its type. We cannot currently perform region allocation because our type system does not provide us with information about what objects might be invisibly referenced by thunks. Knowing that those objects are *constant* is not

enough, what we actually need is a list of region variables that have escaped our analysis and must be assumed to be shared at top-level. Importantly, this is also the same property we need to consider in our type class example.

We have in mind to introduce a new region constraint *Shared* that expresses this property. We would also introduce a constraint *SharedT* that refers to all the region variables in a particular value type, and another *SharedC* that refers to region variables in a closure.

Our *suspend* function would then have type:

$$\begin{array}{lll} \mathit{suspend} & :: & \forall a\ b.\ (a \xrightarrow{e_1\ c_1} b) \rightarrow a \rightarrow b \\ & \triangleright & \mathit{Pure}\ e_1 \\ & , & \mathit{SharedT}\ a \\ & , & \mathit{SharedC}\ c_1 \end{array}$$

This type expresses the fact that the closure of the parameter function, along with the value of type a, is still reachable after this function returns. Neither c_1 or a are present in the type b, but the core language could use the sharing constraints on them to ensure that they are not region-allocated.

5.2.5 Witnesses of no aliasing

As discussed in §4.4.3, if the core translation of a function has two region parameters r_1 and r_2 then we must assume that objects in the corresponding regions may alias. This prevents us from reordering bindings with effects such as $\mathit{Write}\ r_1$ and $\mathit{Read}\ r_2$. A natural extension would be to add a new witness constructor *MkNoAlias*, and use a witness of kind $\mathit{NoAlias}\ r_1\ r_2$ to express the fact that objects in regions named r_1 and r_2 are guaranteed not to alias. The optimiser could then use such witnesses to prove that effects on these regions do not interfere.

If two region variables are introduced at the same point, the generation of a *NoAlias* witness for them is straightforward. We could extend the letregion expression so that multiple region variables can be introduced, and require that witnesses of no aliasing are created at the same point. For example:

$$\begin{array}{ll} \textbf{letregion} & \{\ r_1, r_2\ \} \\ \textbf{with} & \{\ w_1 = \mathit{MkMutable}\ r_1 \\ & \quad w_2 = \mathit{MkMutable}\ r_2 \\ & \quad w_3 = \mathit{MkNoAlias}\ r_1\ r_2\} \\ \textbf{in}\ \ldots & \end{array}$$

However, if the *NoAlias* constructor only has two region parameters, then the number of witness we might want to create is quadratic in the number of region variables introduced by the letregion expression. It may be better to extend the syntax of kinds so they can contain sets of region variables. This would complicate the type system, but it is an orthogonal extension. We could also use this functionality to reduce the multitude of other sorts of witnesses. For example, by using a single witness of kind $\mathit{Mutable}\ \{r_1, r_2, r_3\}$ instead of three separate ones.

5.2.6 Should we treat top-level effects as interfering?

From a utopian viewpoint, effects such as *FileSystem*, *Console* and *Network* should not interfere, but in practice they do. On a unix based system, data written to the special file `/dev/stdout` appears on the console, so in general we should not reorder calls to library functions such as *writeFile* and *putStr*.[1]

However, as we allow programmers to define their own effect constructors, we can forsee use-cases for embedded systems that incorporate effects such as *MotionSensor*, *RobotArm*, *BlinkyLight* etc. Although *MotionSensor* and *RobotArm* would likely interfere, perhaps *MotionSensor* and *BlinkyLight* would not.

It would be straight-forward to allow the programmer to define their own effect interference relationships. This seems like a "fun" extension, but we are not sure how useful it would be in practice. For now, we treat all top level effects as interfering.

5.2.7 Add witnesses to write effects

In the type system for our core language, there is nothing that directly links the fact that write effects must only act on mutable regions. We rely on the signatures of primitive functions such as *updateInt* to contain both the mutability constraint and the write effect, but do not enforce it. We could perhaps change the kind of the *Write* constructor to require a witness that the region written to is mutable, that is:

$$Write :: \Pi(r_1 : \%).\, Mutable\ \ r_1 \rightarrow\ !$$

We have not yet thought of a use-case where a programmer would desire a region to be mutable in the absence of a write effect, or import a primitive function that performs a write but does not require mutability. However, we have avoided giving *Write* the above kind because it would noticeably increase the volume of type information in the core program. An alternative would be to emit a compiler warning if a primitive function was defined with one but not the other, but this is more of an *ad-hoc* solution.

5.2.8 A better type for *force*

In §4.4.2 we avoided assigning a type to *force* because the only sensible type we can give it is $\forall a.\ a \rightarrow a$, and that's not particularly useful. We wish to encode the fact that the resulting value is guaranteed not to be represented by a thunk, but we do not have a way of doing so. For example, if we limited *force* to act on integers then we might try:

$$\begin{array}{lll} force & :: & \forall r_1.\ Int\ \ r_1 \rightarrow Int\ \ r_1 \\ & \rhd & Direct\ \ r_1 \end{array}$$

However, this does not work because we are expecting to apply *force* to objects that are constrained to be in *Lazy* regions.

[1] With the meanings of these functions being the same as in Haskell.

Using a different region variable in the parameter and resulting type does not work either:

$$\begin{array}{lll} \mathit{force} & :: & \forall r_1\ r_2.\ \mathit{Int}\ r_1 \rightarrow \mathit{Int}\ r_2 \\ & \triangleright & \mathit{Direct}\ r_2 \end{array}$$

With this type, the link between the input and output region variables is lost. If we apply *force* to an integer that is constrained to be mutable, then this constraint will not be present on the resulting type. It appears as though we need a way to relate the region variables r_1 and r_2 in a way that maintains all possible region constraints except *Lazy*. In some senses, this is similar to the problem that the *Shape* constraint solves, which is discussed in §2.6.2. We could perhaps write something like:

$$\begin{array}{lll} \mathit{force} & :: & \forall r_1\ r_2.\ \mathit{Int}\ r_1 \rightarrow \mathit{Int}\ r_2 \\ & \triangleright & \mathit{LazyToDirect}\ r_1\ r_2 \\ & , & \mathit{Lazy}\ r_1 \\ & , & \mathit{Direct}\ r_2 \\ & , & \mathit{Alias}\ r_1\ r_2 \end{array}$$

The type inferencer would use the *LazyToDirect* constraint to ensure that all constraints placed on r_1 other than *Lazy* were propagated to r_2, and all constraints placed on r_2 other than *Direct* were propagated to r_1. We would need the extra constraint $\mathit{Alias}\ r_1\ r_2$ in the event our system also contained the *NoAlias* witnesses discussed in §5.2.5.

The question is then how to reflect this *LazyToDirect* constraint in the core language. We could perhaps use a higher kinded witness constructor to convert witnesses on r_1 to witnesses on r_2. Something like:

$$\begin{array}{l} \mathit{MkLazyToDirectConv} \\ \quad :: \quad \forall (k : \% \rightarrow \Diamond,\ k \neq \mathit{Lazy}). \\ \qquad \Pi r_1\ r_2.\ \mathit{LazyToDirect}\ r_1\ r_2 \rightarrow k\ r_1 \rightarrow k\ r_2 \end{array}$$

This encodes the fact that if an object in a region named r_1 is forced, and then considered to be in a fresh region named r_2, then any property of the initial object is also a property of the resulting one, except for the possibility of being a thunk. However, as we do not have any experience with the associated type inference or core-level transforms we cannot comment on its practicality.

5.3 Summary of Contributions

- I present a system that integrates region, effect and closure typing into a unified whole and uses type classes to express mutability and purity constraints. To my knowledge, the system in this thesis is the first to apply mutability constraints to region variables, or purity constraints to effect terms.

- I describe how laziness and arbitrary destructive update can be used sanely in the same program. This is done by applying purity constraints to the visible effects of suspended function applications, and satisfying these constraints by requiring objects read by the function to be constant. This ensures that impure function applications are not suspended, as the behavior of a program which did so would likely be incomprehensible to both the programmer and compiler.

- I present a System-Fc style core language that includes region, effect and closure information. I show how to encode information about mutability and purity using dependently kinded witnesses.

- I describe the behaviour of a Talpin-Jouvelot style effect typing system when applied to higher order functions. I show how some $\sqsupseteq$ constraints can be strengthened to equalities, but others cannot. Strengthening effect constraints allows the volume of type information in the core program to be reduced.

- I show how *Shape* constraints can be used to define type classes such as *Eq* and *Copy*. These constraints are used to require the parameter and return types of a function to have the same overall value type, while allowing the contained region and mutability information to vary.

- I discuss how *Lazy* and *Direct* constraints can be used to track which objects might be represented as thunks. I show how to use this information to optimise programs that use mostly call-by-value evaluation.

- I discuss the concept of material region variables and show how this concept can be used to trim the majority of information out of closure terms.

- I present pull-back projections and show how they can be used to eliminate the need for ML style mutable references. Using pull-back projections is preferable because ML style references pollute the value types of functions and data structures that use them, which can lead to a large amount of refactoring when developing programs.

- I describe how to perform Hindley-Milner style type inference without prior knowledge of the binding dependency graph. If the program uses type directed projections then this graph is not obtainable *a priori*, because the instance function to use for each projection depends on the type of the value being projected.

5.4 The Hair Shirt

When I started this project back in 2004, one of the first things I came across were the slides for Simon Peyton Jones's 15 year Haskell retrospective entitled "Wearing the Hair Shirt" [PJ03b]. When I looked up what a "hair shirt" was, it turned out to be a device of penance. Certain practitioners of the Christian faith wear (or wore) shirts made out of animal hair, because they are uncomfortable, and help to isolate the wearer from worldly passions.

Designing programming languages is almost too much fun. In the words of Aleister Crowley: "We ignore what created us; we adore what we create". It is all too easy to come up with a reasonable idea, fall in love with it, then begin to treat that idea as the one true way of solving any particular problem. Slide 41 of Wearing the Hair Shirt is titled "What really matters?". Four things are listed: Laziness, Purity and Monads, Type Classes, and Sexy Types. "Laziness" has a big red cross through it, and "Purity and Monads" are listed as one thing. I thought about that for some time, and that thinking turned into this thesis.

Five years later, I'd argue that in the context of functional programming, laziness and monads are merely tools, not universal truths. Purity is important to the extent that it reflects an understanding and control over side effects, and type classes and Sexy Types are one and the same. What matters, at least the way I see it, is the Curry-Howard isomorphism, but everyone knew that already.

I find the fact that we can leverage the Curry-Howard isomorphism to express relationships between region, effect and closure information to be highly reassuring. The core axioms, such as the fact that a read of a constant region is pure, are expressed in the kinds of witness constructors. The ambient type system does the rest.

The concrete implementation of DDC still has some wrinkles, but all the ones I know about are cosmetic and do not represent flaws in the overall approach or theory. The type system in this thesis, with its regions, effects, closures and various constraints is large in volume, but the various parts share much common ground. The type inferencer took a long time to work out, mainly because when I started I didn't know what I was doing, but the end result is surprisingly straightforward.

If I were to distill this thesis into one single point, it would be that the distinction between "pure" and "impure" languages is an artificial one. As we can express information about effects and mutability directly in the type system, using a standard framework, the difference between pure and impure is no greater than the difference between *Bool* and *Float*. Effect typing, closure typing, type classing, regions, dependent kinds and projections were all invented by other, eminently clever people. I've spent the last while pasting them together into a pleasing collage and smoothing out the corners. Now the world seems shiny and new.

14

Kappa-Epsilon-Phi-Alpha-Lambda-Eta Iota-Delta

ONION-PEELINGS[2]

The Universe is the Practical Joke of the General
at the Expense of the Particular, quoth FRATER
PERDURABO, and laughed.
But those disciples nearest to him wept, seeing the
Universal Sorrow.
Those next to them laughed, seeing the Universal
Joke.
Below these certain disciples wept.
Then certain laughed.
Others next wept.
Others next laughed.
Next others wept.
Next others laughed.
Last came those that wept because they could not
see the Joke, and those that laughed lest they
should be thought not to see the Joke, and thought
it safe to act like FRATER PERDURABO.
But though FRATER PERDURABO laughed
openly, He also at the same time wept secretly;
and in Himself He neither laughed nor wept.
Nor did He mean what He said.

[2]An excerpt from The Book Of Lies, Aliester Crowley, 1913.

Bibliography

[Abr90] Samson Abramsky, *The lazy lambda calculus*, Research Topics in Functional Programming, Addison-Wesley, 1990, pp. 65–116.

[AHM89] Arnon Avron, Furio Honsell, and Ian A. Mason, *An overview of the Edinburgh Logical Framework*, In Current Trends in Hardware Verification and Automated Theorem Proving, G. Birtwistle, Springer-Verlag, 1989, pp. 323–240.

[Aug84] Lennart Augustsson, *A compiler for lazy ML*, LFP 1984: Proc. of the Symposium on LISP and Functional Programming, ACM, 1984, pp. 218–227.

[AWW91] Alexander Aiken, John H. Williams, and Edward L. Wimmers, *The FL project: The design of a functional language*, 1991.

[Bac78] John Backus, *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs*, Communications of the ACM, vol. 21, number 8, August 1978.

[Ban79] John P. Banning, *An efficient way to find the side effects of procedure calls and the aliases of variables*, POPL 1979: Proc. of the Symposium on Principles of Programming Languages, ACM, 1979, pp. 29–41.

[BB07] Nick Benton and Peter Buchlovsky, *Semantics of an effect analysis for exceptions*, TLDI 2007: Proc. of the International Workshop on Types in Languages Design and Implementation, ACM, 2007, pp. 15–26.

[BE04] Adrian Birka and Michael D. Ernst, *A practical type system and language for reference immutability*, OOPSLA 2004: Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM, 2004, pp. 35–49.

[BHA85] Geoffrey Burn, Chris Hankin, and Samson Abramsky, *The theory of strictness analysis for higher order functions*, Programs as data objects, Springer-Verlag, 1985, pp. 42–62.

[BK99] Nick Benton and Andrew Kennedy, *Monads, effects and transformations*, Electronic Notes in Theoretical Computer Science, Elsevier, 1999, pp. 1–18.

[BKBH07] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann, *Relational semantics for effect-based program transformations with dynamic allocation*, PPDP 2007: Proc. of the In-

ternational Conference on Principles and Practice of Declarative Programming, ACM, 2007, pp. 87–96.

[BKR98] Nick Benton, Andrew Kennedy, and George Russell, *Compiling standard ML to Java bytecodes*, ICFP 1998: Proc. of the International Conference on Functional Programming, ACM, 1998, pp. 129–140.

[BM98] Richard Bird and Lambert Meertens, *Nested datatypes*, MPC 1998: Proc.s of the Mathematics of Program Construction, Springer-Verlag, 1998, pp. 52–67.

[BR00] Adam Bakewell and Colin Runciman, *A model for comparing the space usage of lazy evaluators*, PPDP 2000: Proc. of the International Conference on Principles and Practice of Declarative Programming, ACM, 2000, pp. 151–162.

[BS93] Erik Barendsen and Sjaak Smetsers, *Conventional and uniqueness typing in graph rewrite systems*, FSTTCS 1993: Proc. of the Conference on the Foundations of Software Technology and Theoretical Computer Science, Springer-Verlag, 1993, pp. 41–51.

[BS94] ———, *Uniqueness typing in theory and practice*, PLILP 1995: Proc. of the International Symposium on Programming Languages, Springer, 1994.

[C05] *ISO/IEC 9899:TC2 The C programming language, committee draft*, May 2005.

[Can91] David Cann, *Retire Fortran? a debate rekindled*, ACM/IEEE Conference on Supercomputing, July 1991.

[CC91] Felice Cardone and Mario Coppo, *Type inference with recursive types: syntax and semantics*, Information and Computation **92** (1991), no. 1, 48–80.

[CF04] Robert Cartwright and Mike Fagan, *Soft typing*, SIGPLAN Notices **39** (2004), no. 4, 412–428.

[Cha02] *Haskell 98 foreign function interface 1.0*, 2002, http://www.cse.unsw.edu.au/ chak/haskell/ffi/ffi/ffi.html.

[Cha04] Gregory Chaitin, *Register allocation and spilling via graph coloring*, Best of PLDI 1979-1999 **39** (2004), no. 4, 66–74.

[Che05] Mun Hon Cheong, *Functional programming and 3D games*, Tech. report, University of New South Wales, 2005.

[CKJM05] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow, *Associated types with class*, POPL 2005: Proc. of the International Conference on Principles of Programming Languages, ACM Press, 2005, pp. 1–13.

[CLS07] Duncan Coutts, Roman Leshchinskiy, and Don Stewart, *Stream fusion: from lists to streams to nothing at all*, ICFP 2007: Proc. of the International Conference on Functional Programming, ACM, 2007, pp. 315–326.

[CO94] Kung Chen and Martin Odersky, *A type system for a lambda calculus with assignments*, TACS '94: Proc. of the International Conference on Theoretical Aspects of Computer Software, Springer-Verlag, 1994, pp. 347–364.

[Coo04] Matthew Cook, *Universiality in elementry cellular automata*, Complex Systems, vol. 1, 2004, pp. 1–40.

[CPN98] David Clarke, John Potter, and James Noble, *Ownership types for flexible alias protection*, SIGPLAN Not. **33** (1998), no. 10, 48–64.

[Cpp08] *ISO/IEC IS 14882: The C++ programming language standard, committie draft, 2008-10-08*, October 2008.

[CWM99] Karl Crary, David Walker, and Greg Morrisett, *Typed memory management in a calculus of capabilities*, POPL 1999: Proc. of the International Conference on Principles of Programming Languages, ACM Press, 1999, pp. 262–275.

[DB96] Dominic Duggan and Frederick Bent, *Explaining type inference*, Science of Computer Programming **27** (1996), no. 1, 37–83.

[DF01] Robert Deline and Manuel Fahndrich, *Enforcing high-level protocols in low-level software*, PLDI 2001: Proc. of the Conference on Programming Language Design and Implementation, vol. 36, ACM Press, May 2001, pp. 59–69.

[DM82] Luis Damas and Robin Milner, *Principal type-schemes for functional programs*, POPL 1982: Proc. of the Symposium on Principles of Programming Languages, ACM, 1982, pp. 207–212.

[dMS95] André Luís de Medeiros Santos, *Compilation by Transformation in Non-Strict Functional Languages*, Ph.D. thesis, Department of Computer Science, University of Glasgow, 1995.

[DP03] Joshua Dunfield and Frank Pfenning, *Type assignment for intersections and unions in call-by-value languages*, FOSSACS 2003: Proc. of the International Conference on Foundations of Software Science and Computation Structures, Springer LNCS, 2003, pp. 250–266.

[dVPA07] Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson, *Uniqueness typing simplified*, IFL 2007: Proc. of the Workshop on Implementation and Application of Functional Programming Languages, 2007.

[Erw06] Martin Erwig, *Visual type inference*, Journal of Visual Languages and Computing **17** (2006), no. 2, 161–186.

[Fel91] Matthias Felleisen, *On the expressive power of programming languages*, Science of Computer Programming **17** (1991), no. 1-3, 35–75.

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken, *A theory of type qualifiers*, SIGPLAN Notices **34** (1999), no. 5, 192–203.

[Fil94] Andrzej Filinski, *Representing monads*, POPL 1994: Proc. of the Symposium on Principles of Programming Languages, ACM, 1994, pp. 446–457.

[FM06] Matthew Fluet and Greg Morrisett, *Monadic regions*, Journal of Functional Programming **16** (2006), no. 4-5, 485–545.

[GA01] David Gay and Alex Aiken, *Language support for regions*, SIGPLAN Notices **36** (2001), no. 5, 70–80.

[Gar02] Jacques Garrigue, *Relaxing the value restriction*, APLAS 2002: Proc. of the Asian Workshop on Programming Languages and Systems, 2002, pp. 31–45.

[GCC09] *GCC 4.2.3 online documentation*, 2009, http://gcc.gnu.org.

[Gia00] Douglas C. Giancoli, *Physics for scientists and engineers, third edition*, Prentice Hall, 2000.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The Java language specification, third edition*, 2005.

[GJSO91] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, *Report on the FX programming language*, Tech. report, Massachusetts Institute of Technology, 1991.

[GL86] David K. Gifford and John M. Lucassen, *Integrating functional and imperative programming*, LFP 1986: Proc. of the Conference on LISP and Functional Programming, ACM, 1986, pp. 28–38.

[GMJ+02] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney, *Region-based memory management in cyclone*, SIGPLAN Notices **37** (2002), no. 5, 282–293.

[Gro06] Dan Grossman, *Quantified types in an imperative language*, ACM Trans. Programming Languages and Systems **28** (2006), no. 3, 429–475.

[GS01] Jörgen Gustavsson and David Sands, *Possibilities and limitations of call-by-need space improvement*, ICFP 2001: Proc. of the International Conference on Functional Programming, ACM, 2001, pp. 265–276.

[Gun92] Carl A. Gunter, *Semantics of Programming Languages*, MIT Press, 1992.

[Gup95] Shail Aditya Gupta, *Functional encapsulation and type reconstruction in a strongly-typed, polymorphic language*, Ph.D. thesis, Massachusetts Institute of Technology, 1995.

[Has08] *Haskell' deepseq proposal*, November 2008, http://hackage.haskell.org/trac/haskell-prime/wiki/deep_seq.

[Hee05] Bastian Heeren, *Top quality error messages*, Ph.D. thesis, Departement Informatica, Universiteit Utrecht, 2005.

[Hen02] Fergus Henderson, *Accurate garbage collection in an uncooperative environment*, ISMM '02: Proc. International Symposium on Memory Management, ACM, 2002, pp. 150–156.

[HHPJW96] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler, *Type classes in Haskell*, Trans. on Programming Languages and Systems **18** (1996), no. 2, 109–138.

[HHS02] Bastiaan Heeren, Jurriaan Hage, and Doaitse Swierstra, *Generalizing Hindley-Milner type inference algorithms*, Tech. report, Institute of Information and Computer Science, Universiteit Utrecht, 2002.

[HHS03] ———, *Constraint based type inferencing in Helium*, Tech. report, Institute of Information and Computer Science, Universiteit Utrecht, 2003.

[HJL06] Ralf Hinze, Johan Jeuring, and Andres Løh, *Comparing approaches to generic programming in Haskell*, Tech. report, ICS, Utrecht University, 2006.

[HJSA02] Bastiaan Heeren, Johan Jeuring, Doaitse Swierstra, and Pablo Azero Alcocer, *Improving type-error messages in functional languages*, Tech. report, Institute of Information and Computer Science, Universiteit Utrecht, 2002.

[HM98] Graham Hutton and Erik Meijer, *Monadic parsing in Haskell*, Journal of Functional Programming **8** (1998), no. 4, 437–444.

[HP96] John L. Hennessy and David A. Patterson, *Computer Architecture a Quantative Approach, 2nd Ed*, Morgan Kaufmann, 1996.

[HS07] Tim Harris and Satnam Singh, *Feedback directed implicit parallelism*, ICFP 2007: Proc. of the International Conference on Functional Programming (2007), 251–264.

[Hug83] John Hughes, *The design and implementation of programming languages*, Ph.D. thesis, Oxford University Computing Laboratory, 1983.

[Hug89] ———, *Why functional programming matters*, Computer Journal **32** (1989), no. 2, 98–107.

[Int06] Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual*, March 2006.

[JG91] Pierre Jouvelot and David Gifford, *Algebraic reconstruction of types and effects*, POPL '91: Proc. of the Symposium on Principles of Programming Languages, ACM, 1991, pp. 303–310.

[JMG+02] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang, *Cyclone: A safe dialect of C*, USENIX Annual Technical Conference, June 2002, pp. 275–288.

[Joh85] Thomas Johnsson, *Lambda lifting: transforming programs to recursive equations*, Proc. of the Conference on Functional Programming Languages and Computer Architecture (New York, NY, USA), Springer-Verlag Inc., 1985.

[Jon92] Mark P. Jones, *A theory of qualified types*, ESOP 1992: Proc. of the European Symposium on Programming, vol. 582, Springer-Verlag, 1992, pp. 287–306.

[Jon93] ———, *A system of constructor classes: overloading and implicit higher-order polymorphism*, FPCA 1993: Proc. of the Conference on Functional Programming and Computer Architecture, ACM Press, 1993, pp. 52–61.

[Jon99] ———, *Typing Haskell in Haskell*, Haskell Workshop, 1999.

[Jon00] ———, *Type classes with functional dependencies*, ESOP 2000: Proc. of the European Symposium on Programming, Springer-Verlag, 2000, pp. 230–244.

[JPJ99] Mark P. Jones and Simon Peyton Jones, *Lightweight extensible records for Haskell*, Proc. of the Haskell Workshop, 1999.

[JPJ08] Barry Jay and Simon Peyton Jones, *Scrap your type applications*, MPC '08: Proc. International Conference on Mathematics of Program Construction, Springer-Verlag, 2008, pp. 2–27.

[Kar08] J. Karczmarczuk, *Symbol type?*, October 2008, http://www.mail-archive.com/haskell-cafe@haskell.org/msg30981.html.

[Knu74] Donald E. Knuth, *Structured programming with go to statements*, Computing Surveys **6** (1974), 261–301.

[LA04] Chris Lattner and Vikram Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, CGO 2004: Proc. of the International Symposium on Code Generation and Optimization, Mar 2004.

[LA05] ———, *Transparent pointer compression for linked data structures*, MSP 2005: Proc. of the Workshop on Memory System Performance, June 2005.

[Lau93a] John Launchbury, *Lazy imperative programming*, Tech. report, Yale University, 1993.

[Lau93b] ———, *A natural semantics for lazy evaluation*, POPL 1993: Proc. of the Conference on Principles of Programming Languages, ACM Press, 1993, pp. 144–154.

[LDG+08] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôuillon, *The Objective Caml system, release 3.11, documentation and user's manual.*, Tech. report, Institut National de Recherche en Informatique et en Automatique, 2008.

[LDJC83] Shu Lin and Jr Daniel J. Costello, *Error Control Coding: Fundamentals and Applications*, Prentice-Hall, 1983.

[Ler92] Xavier Leroy, *Polymorphic typing of an algorithmic language*, Tech. report, Institut National de Recherche en Informatique et en Automatique (INRIA), 1992.

[Ler93] ———, *Polymorphism by name*, POPL 1993: Proc. of the Symposium on Principles of Programming Languages, 1993, pp. 220–231.

[Ler97] ———, *The effectiveness of type-based unboxing*, Tech. report, Boston College Computer Science Department, 1997.

[LG88] John. M. Lucassen and David. K. Gifford, *Polymorphic effect systems*, POPL 1988: Proc. of the Symposium on Principles of Programming Languages, ACM, 1988, pp. 47–57.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones, *Monad transformers and modular interpreters*, POPL 1995: Proc. of the Symposium on Principles of Programming Languages, ACM, 1995, pp. 333–343.

[LL05] Daan Leijen and Andres Löh, *Qualified types for mlf*, SIGPLAN Not. **40** (2005), no. 9, 144–155.

[LM01] Daan Leijen and Erik Meijer, *Parsec: Direct style monadic parser combinators for the real world*, Tech. Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[LNSW01] Martin Leucker, Thomas Noll, Perdita Stevens, and Michael Weber, *Functional programming languages for verification tools: Experiences with ML and Haskell*, Proc. of the Scottish Functional Programming Workshop., 2001.

[Lou08] J. Louis, *Why I don't use Haskell for Functional Programming*, 2008, http://jlouisramblings.blogspot.com/2008/03/why-i-dont-use-haskell-for-functional_08.html.

[LP96] John Launchbury and Ross Paterson, *Parametricity and unboxing with unpointed types*, ESOP 1996: European Symposium of Programming, vol. 1058, Springer, April 1996, pp. 204–218.

[LPJ94] John Launchbury and Simon Peyton Jones, *Lazy functional state threads*, PLDI 1994: Proc. of the Conference on Programming Languages Design and Implementation, ACM Press, 1994, pp. 24–35.

[LPJ03] Ralf Lämmel and Simon Peyton Jones, *Scrap your boilerplate: a practical design pattern for generic programming*, TLDI 2003: Proc. of the Workshop on Types in Language Design and Implementation, ACM Press, 2003.

[LW91] Xavier Leroy and Pierre Weis, *Polymorphic type inference and assignment*, POPL 1991: Proc. of the Symposium on Principles of Programming Languages, ACM Press, 1991, pp. 291–302.

[LY98] Oukseh Lee and Kwangkeun Yi, *Proofs about a folklore let-polymorphic type inference algorithm*, ACM Trans. Programming Languages and Systems **20** (1998), no. 4, 707–723.

[Mac91] David B. MacQueen, *Standard ML of New Jersey*, Proc. of the Symposium on Programming Language Implementation and Logic Programming, Springer-Verlag, 1991, pp. 1–13.

[Mil78] Robin Milner, *A theory of type polymorphism in programming*, Journal of Computer and System Sciences **17** (1978), 348–375.

[Mit07] Neil Mitchell, *Uniform boilerplate and list processing*, Proc. of the Haskell Workshop, ACM, September 2007.

[MNM04] Peter Møller-Neergaard and Harry G. Mairson, *Types, potency, and idempotency: why nonlinearity and amnesia make a type system work*, ICFP 2004: Proc. of the International Conference on Functional Programming, ACM, 2004, pp. 138–149.

[Mog89] Eugenio Moggi, *Computational lambda-calculus and monads*, Proc. of the Symposium on Logic in Computer Science, IEEE Computer Society Press, June 1989, pp. 14–23.

[MPJ04] Simon Marlow and Simon Peyton Jones, *Making a fast curry: push/enter vs. eval/apply for higher-order languages*, ICFP 2004: Proc. of the International Conference on Functional programming, ACM, 2004, pp. 4–15.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen (eds.), *The definition of Standard ML (revised)*, The MIT Press, 1997.

[Myc84] Alan Mycroft, *Polymorphic type schemes and recursive definitions*, Proc. of the International Symposium on Programming, Springer-Verlag, 1984, pp. 217–228.

[NAH+95] Rishiyur S. Nikhil, Arvind, James Hicks, Shaft Aditya, Lennart Augustsson, Jan-Willem Maessen, and Yuli Zhou, *pH language reference manual, version 1.0*, Tech. report, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, January 1995.

[NN93] Flemming Nielson and Hanne Riis Nielson, *From CML to process algebras*, Theoretical Computer Science, Springer-Verlag, 1993, pp. 493–508.

[NN99] ———, *Type and effect systems*, Correct System Design, number 1710 in Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 114–136.

[NSvEP91] Eric Nøcker, Sjaak Smetsers, Marko van Eekelen, and Rinus Plasmeijer, *Concurrent Clean*, PARLE 1991: In Proc. of Parallel Architectures and Languages Europe (1991), 202–219.

[Ode91] Martin Odersky, *How to make destructive updates less destructive*, POPL 1991: Proc. of the Symposium on Principles of Programming Languages, ACM Press, 1991, pp. 25–26.

[OG98] Chris Okasaki and Andrew Gill, *Fast mergeable integer maps*, In Workshop on ML, 1998, pp. 77–86.

[Oka98a] Chris Okasaki, *Higher-order functions for parsing or why would anyone ever want to use a sixth-order function*, Journal of Functional Programming 8, 1998.

[Oka98b] ———, *Purely functional data structures*, Cambridge University Press, 1998.

[ORH93] Martin Odersky, Dan Rabin, and Paul Hudak, *Call by name, assignment, and the lambda calculus*, POPL 1993: Proc. of the Symposium on Principles of Programming Languages, ACM, 1993, pp. 43–56.

[Par92] Will Partain, *The nofib benchmark suite of Haskell programs*, Proc. of the Glasgow Workshop on Functional Programming, Springer-Verlag, 1992, pp. 195–202.

[PHL+77] Gerald J. Popek, Jim J. Horning, Butler W. Lampson, James G. Mitchell, and Ralph L. London, *Notes on the design of Euclid*, Proc. of the Conference on Language Design for Reliable Software, ACM, 1977, pp. 11–18.

[Pie02] Benjamin C. Pierce, *Types and Programming Languages*, The MIT Press, 2002.

[Pie05] _____, *Advanced Topics in Types and Programming Languages*, The MIT Press, 2005.

[PJ87] Simon Peyton Jones, *The Implementation of Functional Programming Languages*, ch. 13, Prentice-Hall, 1987.

[PJ92] _____, *Implementing lazy functional languages on stock hardware: The spineless tagless G-machine*, Journal of Functional Programming **2** (1992), 127–202.

[PJ94] _____, *Compilation by transformation in the Glasgow Haskell Compiler*, In Glasgow Workshop on Functional Programming, Springer, 1994, pp. 184–204.

[PJ03a] Simon Peyton Jones (ed.), *Haskell 98 language and libraries: The revised report.*, Cambridge University Press, April 2003.

[PJ03b] _____, *Wearing the hair shirt*, 2003, http://research.microsoft.com/users/Cambridge/simonpj/Papers/haskell-retrospective/HaskellRetrospective.pdf.

[PJJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer, *Type classes: an exploration of the design space*, In Haskell Workshop, 1997.

[PJM97] Simon Peyton Jones and Erik Meijer, *Henk: a typed intermediate language*, Proc. of the Workshop on Types in Compilation, 1997.

[PJNO97] Simon Peyton Jones, Thomas Nordin, and Dino Oliva, *C minus minus: A portable assembly language*, Proc. of the Workshop on Implementing Functional Languages, Springer Verlag, 1997, pp. 1–19.

[PJPS96] Simon Peyton Jones, Will Partain, and André Santos, *Let-floating: moving bindings to give faster programs*, ICFP 1996: Proc. of the International Conference on Functional Programming, ACM, 1996, pp. 1–12.

[PJS98] Simon Peyton Jones and André L. M. Santos, *A transformation-based optimiser for Haskell*, Science of Computer Programming **32** (1998), no. 1–3, 3–47.

[PJSLT98] Simon Peyton Jones, Mark Shields, John Launchbury, and Andrew Tolmach, *Bridging the gulf: a common intermediate language for ML and Haskell*, POPL 1998: Proc. of the Symposium on Principles of Programming Languages, ACM, 1998, pp. 49–61.

[PJVWS07] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields, *Practical type inference for arbitrary-rank types*, Journal of Functional Programming **17** (2007), no. 1, 1–82.

[PJW92] Simon Peyton Jones and Philip Wadler, *Imperative functional programming*, Proc. of the Symposium on Principles of Programming Languages, October 1992, pp. 71–84.

[PMN88] Carl G. Ponder, Patrick McGeer, and Anthony P-C. Ng, *Are applicative languages inefficient?*, SIGPLAN Notices **23** (1988), no. 6, 135–139.

[Pot95] François Pottier, *Type inference and simplification for recursively constrained types*, Actes du GDR Programmation 1995 (journe du ple Programmation Fonctionnelle), November 1995.

[Pot00] ———, *A versatile constraint-based type inference system*, Nordic Journal of Computing **7** (2000), no. 4, 312–347.

[PR05] François Pottier and Didier Remy, *The essence of ML type inference*, Advanced Topics in Types and Programming Languages (Benjamin C. Pierce, ed.), MIT Press, 2005, pp. 389–489.

[Rab96] Daniel Eli Rabin, *Calculi for Functional Programming Languages with Assignment*, Ph.D. thesis, Yale University, 1996.

[Rem94] Didier Remy, *Type inference for records in natural extension of ML*, Theoretical aspects of object-oriented programming: types, semantics, and language design (1994), 67–95.

[Ren02] Paul Rendell, *Turing universality of the game of life*, Collision-based Computing, Springer-Verlag, 2002, pp. 513–539.

[Rey74] John C. Reynolds, *Towards a theory of type structure*, Programming Symposium, Proceedings Colloque sur la Programmation, Springer-Verlag, 1974, pp. 408–423.

[Rey78] John C. Reynolds, *Syntactic control of interference*, POPL 1978: Proc. of the Symposium on Principles of Programming Languages, ACM, 1978, pp. 39–46.

[Ros84] J. Barkley Rosser, *Highlights of the history of the lambda-calculus*, IEEE Annals of the History of Computing **6** (1984), no. 4, 337–349.

[RY08] Didier Remy and Boris Yakobowski, *Graphic type constraints and efficient type inference: From ML to ML^F*, ICFP 2008: Proc. of the International Conference on Functional Programming, September 2008.

[Sab98] Amr Sabry, *What is a purely functional language?*, Journal of Functional Programming **8** (1998), no. 1, 1–22.

[San94] Patrick Sansom, *Time profiling a lazy functional compiler*, Functional Programming, Springer-Verlag, 1994.

[SCA93] A. V. S. Sastry, William Clinger, and Zena Ariola, *Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates*, FPCA '93: Proc. of the Conference on Functional Programming Languages and Computer Architecture, ACM, 1993, pp. 266–275.

[SCPJD07] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly, *System-F with type equality coercions*, Proc. of the Workshop on Types in Language Design and Implementation, ACM Press, 2007.

[She05] Tim Sheard, *Putting Curry-Howard to work*, Proc. of the Haskell Workshop, ACM Press, September 2005.

[SJ07] Don Stewart and Spencer Janssen, *XMonad: A tiling window manager*, Proc. of the Haskell Workshop, ACM Press, Sep 2007.

[SOW99] Martin Sulzmann, Martin Odersky, and Martin Wehr, *Type inference with constrained types*, Theory and Practice of Object Systems **5** (1999), no. 1, 35–55.

[SPJ95] Patrick Sansom and Simon Peyton Jones, *Time and space profiling for non-strict, higher-order functional languages*, POPL 1995: Proc. of the Symposium on Principles of Programming Languages, ACM, 1995, pp. 355–366.

[SR95] R. Sekar and I.V. Ramakrishnan, *Fast strictness analysis based on demand propagation*, ACM Transactions on Programming Languages and Systems **17** (1995), 17–6.

[SRH04] Michael D. Smith, Norman Ramsey, and Glenn Holloway, *A generalized algorithm for graph-coloring register allocation*, PLDI 2004: Proc. of the Conference on Programming Language Design and Implementation, ACM, 2004, pp. 277–288.

[SRI91] Vipin Swarup, Uday S. Reddy, and Evan Ireland, *Assignments for applicative languages*, Proc. of the Conference on Functional Programming Languages and Computer Architecture, Springer-Verlag New York, Inc., 1991, pp. 192–214.

[SS76] Guy L. Steele and Gerald J. Sussman, *Lambda: The ultimate imperative*, Tech. report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1976.

[SS90] Harold Søndergaard and Peter Sestoft, *Referential transparency, definiteness and unfoldability*, Acta Informatica **27** (1990), no. 6, 505–517, Reviewed in *Computing Reviews* **32** (3): 144–145, 1991.

[SSD08a] Jonathan Shapiro, Swaroop Sridhar, and Scott Doerrie, *BitC language specification*, Tech. report, The EROS Group LLC and Johns Hopkins University, 2008.

[SSD08b] _____, *The origins of the BitC programming language*, Tech. report, The EROS Group LLC and Johns Hopkins University, 2008.

[SSh08] Christian Skalka, Scott Smith, and David Van horn, *Types and trace effects of higher order programs*, Journal of Functional Programming **18** (2008), no. 2, 179–249.

[SSS08] Swaroop Sridhar, Jonathan S. Shapiro, and Scott F. Smith, *Sound and complete type inference for a systems programming language*, APLAS 2008: Proc. of the Asian Symposium on Programming Languages and Systems, Springer-Verlag, 2008, pp. 290–306.

[SSS09] Swaroop Sridhar, Jonathan S. Shapiro, and Scott F. Smitt, *Formalization of the BitC type system*, Tech. report, Johns Hopkins University, 2009.

[SSW04] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny, *Improving type error diagnosis*, Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell (New York, NY, USA), ACM, 2004, pp. 80–91.

[Str86] Bjarne Stroustrup, *An overview of C++*, Proc. of the Workshop on Object-Oriented programming, ACM, 1986, pp. 7–18.

[Sun02] Sun Microsystems, Inc, *SPARC assembly language reference manual*, May 2002.

[TA05] Tachio Terauchi and Alex Aiken, *Witnessing side effects*, ICFP 2005: Proc. of the International Conference on Functional Programming, 2005.

[TB98] Mads Tofte and Lars Birkedal, *A region inference algorithm*, ACM Transactions on Programming Languages and Systems **20** (1998), no. 4, 724–767.

[TBE+06] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft, *Programming with regions in the MLKit (revised for version 4.3.0)*, Tech. report, IT University of Copenhagen, Denmark, January 2006.

[THLPJ98] Phil Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon Peyton Jones, *Algorithm + Strategy = Parallelism*, Journal of Functional Programming **8** (1998), 23–60.

[TJ92a] Jean-Pierre Talpin and Pierre Jouvelot, *Polymorphic type, region and effect inference*, Journal of Functional Programming **2** (1992), 245–271.

[TJ92b] ———, *The type and effect discipline*, Proc. of the Symposium on Logic in Computer Science, IEEE Computer Society Press, 1992, pp. 162–173.

[Tof90] Mads Tofte, *Type inference for polymorphic references*, Information and Computation **89** (1990), no. 1, 1–34.

[Tol98] Andrew Tolmach, *Optimizing ML using a hierarchy of monadic types*, In Workshop on Types in Compilation, Springer Verlag, 1998, pp. 97–113.

[Wad87] Philip Wadler, *Fixing some space leaks with a garbage collector*, Software Practice and Experience **17** (1987), no. 9, 595–609.

[Wad90a] ———, *Comprehending monads*, Mathematical Structures in Computer Science, 1990, pp. 61–78.

[Wad90b] ———, *Linear types can change the world*, Working Conference on Programming Concepts and Methods, 1990, pp. 347–359.

[Wad04] ———, *Functional programming in the real world*, SIGPLAN Notices **33** (2004), no. 2, 25–30.

[Wan86] Mitchell Wand, *Finding the source of type errors*, POPL 1986: Proc. of the Symposium on Principles of Programming Languages, ACM, 1986, pp. 38–43.

[WB89] Philip Wadler and Stephen Blott, *How to make ad-hoc polymorphism less ad hoc*, POPL 1989: Proc. of the Symposium on Principles of Programming Languages, ACM Press, 1989, pp. 60–76.

[WH87] Philip Wadler and R. J. M. Hughes, *Projections for strictness analysis*, Proc. of the Conference on Functional Programming Languages and Computer Architecture, Springer-Verlag, 1987, pp. 385–407.

[WPJ99] Keith Wansbrough and Simon Peyton Jones, *Once upon a polymorphic type*, POPL 1999: Proc. of the Symposium on Principles of Programming Languages, ACM, 1999, pp. 15–28.

[Wri92] Andrew K. Wright, *Typing references by effect inference*, ESOP 1992: Proc. of the European Symposium on Programming, vol. 582, Springer-Verlag, 1992, pp. 473–491.

[Wri96] Andrew Wright, *Polymorphism for imperative languages without imperative types*, Tech. Report TR93-200, Rice University, 1996.

[WT03] Philip Wadler and Peter Thiemann, *The marriage of effects and monads*, ACM Trans. Computation and Logic **4** (2003), no. 1, 1–32.

[XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen, *Guarded recursive datatype constructors*, SIGPLAN Notices **38** (2003), no. 1, 224–235.

[YR97] Hongseok Yang and Uday Reddy, *Imperative lambda calculus revisited*, Tech. report, University of Illinois at Urbana-Champaign, August 1997.

Appendix A

Proofs of Language Properties

In this appendix we present the formal proofs of language properties for the system in §4.2, culminating in a proof of soundness. Each of these proofs is by induction over the derivation of a typing judgement. When presenting each case, we will assume the statement being considered and then invoke the standard inversion lemmas [Pie02] to fill in the appropriate premises.

For example, the proof of Substitution of Values in Values starts with the case $t = x$. We assume the statement $\Gamma,\ x : \tau_2 \mid \Sigma \vdash x :: \tau_1 \,;\, \bot$, and use the inversion lemma to give $x : \tau_1 \in \Gamma,\ x : \tau_2$.

$$\frac{(2)\ x : \tau_1 \in \Gamma,\ x : \tau_2}{(\underline{1})\ \Gamma,\ x : \tau_2 \mid \Sigma \vdash x :: \tau_1 \,;\, \bot}$$

Each statement is numbered for identification purposes, and we underline the numbers of statements which are assumptions. In some cases, not all statements obtained by the inversion lemmas will be used, but will include them as premises so that the typing rules maintain their familiar shapes.

We will omit the quantifiers "for all" and "for some" when they are obvious from the context, as they clutter the proof without providing much additional information.

Firstly, some standard lemmas:

Lemma: (Forms of Terms and Types)

When a term is in normal form we can determine its shape by inspecting its type [Pie02]. Similarly, when a type is in normal form we can determine its shape by inspecting its kind.

For example:

If	t is a value
and	$\emptyset \mid \Sigma \vdash t :: \tau_1 \to \tau_2 \,;\, \sigma$
then	$t = \lambda(x : \tau_1).\, t'$

By inspection of the typing rules. The only values which can have function types are lambda abstractions and variables, but if the type environment is empty the value cannot be a variable.

Lemma: (No free witness variables in effects)

If $\Gamma \mid \Sigma \vdash t :: \varphi ; \sigma$
and $\Gamma \mid \Sigma \vdash_{\mathrm{T}} w :: \kappa$ and $\Gamma \mid \Sigma \vdash_{\mathrm{K}} \kappa :: \Diamond$
then $\sigma[\delta/w] \equiv \sigma$

By inspection of the kinding rules for effect constructors.

Lemma: (No free witness variables in term types)

If $\Gamma \mid \Sigma \vdash t :: \varphi ; \sigma$
and $\Gamma \mid \Sigma \vdash_{\mathrm{T}} w :: \kappa$ and $\Gamma \mid \Sigma \vdash_{\mathrm{K}} \kappa :: \Diamond$
then $\varphi[\delta/w] \equiv \varphi$

By inspection of the kinding rules for value type constructors.

Lemma: (Weaken Store Typing)

If we can assign a term t some type and effect, then we can also assign t the same type and effect under a larger store typing. This property is also true for kind and similarity judgements.

If $\Gamma \mid \Sigma \vdash t :: \tau ; \sigma$
and $\Sigma' \supseteq \Sigma$
then $\Gamma \mid \Sigma' \vdash t :: \tau ; \sigma$

By induction over the derivation of $\Gamma \mid \Sigma \vdash t :: \tau ; \sigma$. At the top of the derivation tree we will have uses of (TyLoc) which include statements such as $l : \tau \in \Sigma$. These statements remain true when Σ is extended.

Lemma: (Strengthen Type Environment)

If $\Gamma, x : \tau \mid \Sigma \vdash_{\mathrm{T}} \varphi :: \kappa$
then $\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi :: \kappa$

By inspection of the forms of types. Types to not contain value variables.

Lemma: (Similarity under Substitution)

If $\varphi_1 \sim_\Sigma \varphi_1'$
and $\varphi_2 \sim_\Sigma \varphi_2'$
then $\varphi_1[\varphi_2/a] \sim_\Sigma \varphi_1'[\varphi_2'/a]$

Easy induction.

Lemma: (Region Witness Assertion)

If we add a property to the heap, then we can always evaluate the witness constructor that tests for it.

The statement H, $\overline{\mathrm{propOf}(\Delta)}$; $\overline{\delta \rightsquigarrow \Delta}$ and $\delta \in \{\mathit{MkConst}\ r, \mathit{MkMutable}\ r\}$ for some r, Δ is true.

By inspection of the transition rules (EwConst) and (EwMutable).

Lemma: (Progress of Purity)

If $\emptyset \mid \Sigma \vdash_{\mathrm{T}} \delta :: \mathit{Pure}\ \sigma$
and nofab(δ)
then $\delta = \underline{\mathrm{pure}\ \sigma}$
or H ; $\delta \rightsquigarrow \delta'$ for some H, δ'.

Proof:

(1)	$\emptyset \mid \Gamma \vdash_{\mathrm{T}} \delta :: \mathit{Pure}\ \sigma$	(assume)
(2)	nofab(δ)	(assume)
(3)	$\delta \in \{\mathit{MkPure}\ \bot,\ \mathit{MkPurify}\ \underline{\rho}\ \delta_1,$ $\mathit{MkPureJoin}\ \sigma_2\ \sigma_3\ \delta_2\ \delta_3\}$	(Forms of Types 1)
Case: $\delta = \mathit{MkPure}\ \bot$		
(5)	H ; $\mathit{MkPure}\ \bot \rightsquigarrow \underline{\mathrm{pure}\ \bot}$	(EwPure)
Case: $\delta = \mathit{MkPurify}\ \underline{\rho}\ \delta_1$		
(6)	$\delta_1 = \underline{\mathrm{const}\ \rho}$	(Kind of *MkPurify* 2)
(7)	H ; $\mathit{MkPurify}\ (\underline{\mathrm{const}\ \rho}) \rightsquigarrow \underline{\mathrm{pure}\ (\mathit{Read}\ \rho)}$	(EwPurify 6 7)
Case: $\delta = \mathit{MkPureJoin}\ \sigma_2\ \sigma_3\ \delta_2\ \delta_3$		
(8)	$\emptyset \mid \Sigma \vdash_{\mathrm{T}} \delta_2 :: \mathit{Pure}\ \sigma_2$	(Kind of *MkPureJoin*)
(9)	nofab(δ_2)	(Def. nofab 2)
(10)	$\delta_2 = \underline{\mathrm{pure}\ \sigma_2}$ or H ; $\delta_2 \rightsquigarrow \delta_2'$	(IH 8 9)
(11)	$\delta_3 = \underline{\mathrm{pure}\ \sigma_3}$ or H ; $\delta_3 \rightsquigarrow \delta_3'$	(Similarly)
(12)	Either (EwPureJoin1), (EwPureJoin2) or (EwPureJoin3) applies	

Lemma: (Substitution of Values in Values)

If $\Gamma,\ x : \tau_2 \mid \Sigma \vdash\ t\ ::\ \tau_1\ ;\ \sigma$
and $\Gamma \mid \Sigma \vdash\ v^\circ\ ::\ \tau_2'\ ;\ \bot$
and $\tau_2 \sim_\Sigma \tau_2'$
then $\Gamma \mid \Sigma \vdash\ t[v^\circ/x]\ ::\ \tau_1'\ ;\ \sigma'$
and $\tau_1 \sim_\Sigma \tau_1'$
and $\sigma\ \sim_\Sigma \sigma'$

Proof: By induction over the derivation of $\Gamma,\ x : \tau_2 \mid \Sigma \vdash\ t\ ::\ \tau_1\ ;\ \sigma$

(IH) Subst. Values/Values holds for all subterms of t. (assume)

Case : $t = y$ / TyVar

Trivial. $x \neq y$ so t is unaffected.

Case : $t = x$ / TyVar

$$\frac{(4)\ x : \tau_1 \in \Gamma,\ x : \tau_2}{(\underline{1})\ \Gamma,\ x : \tau_2 \mid \Sigma \vdash\ x\ ::\ \tau_1\ ;\ \bot}$$

($\underline{2}$)	$\Gamma \mid \Sigma \vdash\ v^\circ\ ::\ \tau_2'\ ;\ \bot$	(assume)
($\underline{3}$)	$\tau_2 \sim_\Sigma \tau_2'$	(assume)
(5)	$\tau_1 \equiv \tau_2$	(Def. Type Env 4)
(6)	$\Gamma \mid \Sigma \vdash\ x[v^\circ/x]\ ::\ \tau_2'\ ;\ \bot$	(Def. Sub. 2)
(7)	$\tau_1 \sim_\Sigma \tau_2'$	(3 5)

Case : $t = \Lambda(x : \kappa).\, t_1$ / TyAbsT

$$\frac{(4)\ \Gamma,\ x : \tau_2,\ a : \kappa \mid \Sigma \vdash\ t_1\ ::\ \tau_1\ ;\ \sigma}{(\underline{1})\ \Gamma,\ x : \tau_2 \mid \Sigma \vdash\ \Lambda(a : \kappa).\, t_1\ ::\ \forall(a : \kappa).\, \tau_1\ ;\ \sigma}$$

($\underline{2}$)	$\Gamma \mid \Sigma \vdash\ v^\circ\ ::\ \tau_2'\ ;\ \bot$	(assume)
($\underline{3}$)	$\tau_2 \sim_\Sigma \tau_2'$	(assume)
(5)	$\Gamma,\ a : \kappa \mid \Sigma \vdash\ v^\circ\ ::\ \tau_2'\ ;\ \bot$	(Weak. Type Env 2)
(6..8)	$\Gamma,\ a : \kappa \mid \Sigma \vdash\ t_1[v^\circ/x]\ ::\ \tau_1'\ ;\ \sigma'$, $\tau_1 \sim_\Sigma \tau_1',\ \sigma \sim_\Sigma \sigma'$	(IH 4 5 3)
(9)	$\Gamma \mid \Sigma \vdash\ \Lambda(a : \kappa).\, (t_1[v^\circ/x])\ ::\ \forall(a : \kappa).\tau_1'\ ;\ \sigma'$	(TyAbsT 6)
(10)	$\Gamma \mid \Sigma \vdash\ (\Lambda(a : \kappa).\, t_1)[v^\circ/x]\ ::\ \forall(a : \kappa).\tau_1'\ ;\ \sigma'$	(Def. Sub. 9)

Case: $t = t_1\ \varphi_2$ / TyAppT

$$\frac{(4)\ \Gamma,\ x : \tau_2 \mid \Sigma \vdash\ t_1\ ::\ \forall(a : \kappa_{11}).\ \varphi_{12}\ ;\ \sigma \qquad (5)\ \Gamma,\ x : \tau_2 \mid \Sigma \vdash_{\mathrm{T}} \varphi_2\ ::\ \kappa_2 \quad (6)\ \kappa_{11} \sim_\Sigma \kappa_2}{(\underline{1})\ \Gamma,\ x : \tau_2 \mid \Sigma \vdash\ t_1\ \varphi_2\ ::\ \varphi_{12}[\varphi_2/a]\ ;\ \sigma[\varphi_2/a]}$$

($\underline{2}$)	$\Gamma \mid \Sigma \vdash\ v^\circ\ ::\ \tau_2'\ ;\ \bot$	(assume)
($\underline{3}$)	$\tau_2 \sim_\Sigma \tau_2'$	(assume)
(7..9)	$\Gamma \mid \Sigma \vdash\ t_1[v^\circ/x]\ ::\ \forall(a : \kappa_{11}').\ \varphi_{12}'\ ;\ \sigma'$ $\forall(a : \kappa_{11}).\ \varphi_{12} \sim_\Sigma \forall(a : \kappa_{11}').\ \varphi_{12}'$ $\sigma[\varphi_2/a] \sim_\Sigma \sigma'$	(IH 4 2 3)
(10)	$\kappa_{11} \sim_\Sigma \kappa_{11}'$	(SimAll 8)
(11)	$\kappa_{11}' \sim_\Sigma \kappa_2$	(SimTrans 6 10)
(12)	$\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_2\ ::\ \kappa_2$	(Str. Type Env 5)
(13)	$\Gamma \mid \Sigma \vdash\ t_1[v^\circ/x]\ \varphi_2\ ::\ \varphi_{12}'[\varphi_2/a]\ ;\ \sigma'[\varphi_2/a]$	(TyAppT 7 12 11)
(14)	$\Gamma \mid \Sigma \vdash\ (t_1\ \varphi_2)[v^\circ/x]\ ::\ \varphi_{12}'[\varphi_2/a]\ ;\ \sigma'[\varphi_2/a]$	(Def. Sub. 13)

Case: $t = \lambda(x : \tau).\ t_1$ / TyAbs

Similarly to TyAbsT case.

Case: $t = t_1\ t_2$ / TyApp

$$\frac{(4)\ \Gamma,\ x : \tau_3 \mid \Sigma \vdash\ t_1\ ::\ \tau_{11} \xrightarrow{\sigma} \tau_{12}\ ;\ \sigma_1 \qquad (5)\ \Gamma,\ x : \tau_3 \mid \Sigma \vdash\ t_2\ ::\ \tau_2\ ;\ \sigma_2 \quad (6)\ \tau_{11} \sim_\Sigma \tau_2}{(\underline{1})\ \Gamma,\ x : \tau_3 \mid \Sigma \vdash\ t_1\ t_2\ ::\ \tau_{12}\ ;\ \sigma_1 \vee \sigma_2 \vee \sigma}$$

($\underline{2}$)	$\Gamma \mid \Sigma \vdash\ v^\circ\ ::\ \tau_3'\ ;\ \bot$	(assume)
($\underline{3}$)	$\tau_3 \sim_\Sigma \tau_3'$	(assume)
(7..9)	$\Gamma \mid \Sigma \vdash\ t_1[v^\circ/x]\ ::\ \tau_{11}' \xrightarrow{\sigma'} \tau_{12}'\ ;\ \sigma_1'$ $\tau_{11}' \xrightarrow{\sigma'} \tau_{12}' \sim_\Sigma \tau_{11} \xrightarrow{\sigma} \tau_{12}$ $\sigma_1 \sim_\Sigma \sigma_1'$	(IH 4 2 3)
(10..12)	$\Gamma \mid \Sigma \vdash\ t_2[v^\circ/x]\ ::\ \tau_2'\ ;\ \sigma_2'$ $\tau_2 \sim_\Sigma \tau_2'$ $\sigma_2 \sim_\Sigma \sigma_2'$	(IH 5 2 3)
(13)	$\tau_{11}' \sim_\Sigma \tau_2'$	(SimApp, SimTrans 8 6 11)
(14)	$\Gamma \mid \Sigma \vdash\ t_1[v^\circ/x]\ t_2[v^\circ/x]\ ::\ \tau_{12}'\ ;\ \sigma_1' \vee \sigma_2' \vee \sigma'$	(TyApp 7 10 13)
(15)	$\Gamma \mid \Sigma \vdash\ (t_1\ t_2)[v^\circ/x]\ ::\ \tau_{12}'\ ;\ \sigma_1' \vee \sigma_2' \vee \sigma'$	(Def. Sub. 14)

Case: $t = (\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2)$ / TyLet
Case: $t = (\mathbf{letregion}\ r\ \mathbf{with}\ \{\overline{w_i :: \delta_i}\}\ \mathbf{in}\ t_1)$ / TyLetRegion
Case: $t = (\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3)$ / TyIf

Similarly to TyApp case.

Case: $t = \mathit{True}\ \varphi$ / TyTrue
Case: $t = \mathit{False}\ \varphi$ / TyFalse

Trivial. Types contain do not contain value variables.

Case: $t = \mathit{update}\ \delta\ t_1\ t_2$ / TyUpdate
Case: $t = \mathit{suspend}\ \delta\ t_1\ t_2$ / TySuspend

Similarly to TyApp case.

Case: $t = ()$ / TyUnit

Trivial. Unit does not contain value variables.

Case: $t = \underline{l}$ / TyLoc

Trivial. Locations do not contain value variables.

Lemma: (Substitution of Types in Values)

If $\Gamma,\ a : \kappa_3 \mid \Sigma \vdash\ t\ ::\ \tau_1\ ;\ \sigma$
and $\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_2\ ::\ \kappa_2$
and $\kappa_3 \sim_\Sigma \kappa_2$
then $\Gamma[\varphi_2/a] \mid \Sigma \vdash\ t[\varphi_2/a]\ ::\ \tau_1[\varphi_2/a]\ ;\ \sigma[\varphi_2/a]$

Proof: by induction over the derivation of $\Gamma,\ a : \kappa_3 \mid \Sigma \vdash\ t\ ::\ \tau_1\ ;\ \sigma$

(IH) Substitution holds for all subterms of t. (assume)

Case: $t = x$ / TyVar

Trival. No type vars in value vars.

Case: $t = \Lambda(a : \kappa).\, t$ / TyAbsT

$$\frac{(4)\ \Gamma,\ a : \kappa_3,\ a_{11} : \kappa_{11} \mid \Sigma \vdash\ t_{12}\ ::\ \tau_{12}\ ;\ \sigma_1}{(\underline{1})\ \Gamma,\ a : \kappa_3 \mid \Sigma \vdash\ \Lambda(a_{11} : \kappa_{11}).\, t_{12}\ ::\ \forall(a_{11} : \kappa_{11}).\, \tau_{12}\ ;\ \sigma_1}$$

($\underline{2}$) $\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_2\ ::\ \kappa_2$ (assume)

($\underline{3}$) $\kappa_3 \sim_\Sigma \kappa_2$ (assume)

(5) $\Gamma,\ a_{11} : \kappa_{11} \mid \Sigma \vdash_{\mathrm{T}} \varphi_2\ ::\ \kappa_2$ (Weak. Type Env 2)

(6) $(\Gamma,\ a_{11} : \kappa_{11})[\varphi_2/a] \mid \Sigma \vdash\ t_{12}[\varphi_2/a]\ ::\ \tau_{12}[\varphi_2/a]\ ;\ \sigma_1[\varphi_2/a]$ (IH 4 5 3)

(7) $\Gamma[\varphi_2/a],\ a_{11} : \kappa_{11}[\varphi_2/a] \mid \Sigma \vdash\ t_{12}[\varphi_2/a]\ ::\ \tau_{12}[\varphi_2/a]\ ;\ \sigma_1[\varphi_2/a]$ (Def. Sub. 6)

(8) $\Gamma[\varphi_2/a] \mid \Sigma \vdash (\Lambda(a_{11} : \kappa_{11}).\, t_{12})[\varphi_2/a]\ ::\ (\forall(a_{11} : \kappa_{11}).\, \tau_{12})[\varphi_2/a]\ ;\ \sigma_1[\varphi_2/a]$ (Def. Sub, TyAbsT 7)

Case: $t = t_{11}\ \varphi_{12}$ / TyAppT

$$\frac{(4)\ \Gamma,\ a : \kappa_4 \mid \Sigma \vdash\ t_1\ ::\ \forall(a_1 : \kappa_{11}).\, \varphi_{12}\ ;\ \sigma_1 \quad (5)\ \Gamma,\ a : \kappa_4 \mid \Sigma \vdash_{\mathrm{T}} \varphi_2\ ::\ \kappa_2 \quad (6)\ \kappa_{11} \sim_\Sigma \kappa_2}{(\underline{1})\ \Gamma,\ a : \kappa_4 \mid \Sigma \vdash\ t_1\ \varphi_2\ ::\ \varphi_{12}[\varphi_2/a_1]\ ;\ \sigma_1[\varphi_2/a_1]}$$

($\underline{2}$) $\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_3\ ::\ \kappa_3$ (assume)

($\underline{3}$) $\kappa_4 \sim_\Sigma \kappa_3$ (assume)

(7) $\Gamma[\varphi_3/a] \mid \Sigma \vdash\ t_1[\varphi_3/a]\ ::\ (\forall(a_1 : \kappa_{11}).\, \varphi_{12})[\varphi_3/a]\ ;\ \sigma_1[\varphi_3/a]$ (IH 4 2 3)

(8) $\Gamma[\varphi_3/a] \mid \Sigma \vdash\ t_1[\varphi_3/a]\ ::\ \forall(a_1 : \kappa_{11}[\varphi_3/a]).\, \varphi_{12}[\varphi_3/a]\ ;\ \sigma_1[\varphi_3/a]$ (Def. Sub. 7)

(9) $\Gamma[\varphi_3/a] \mid \Sigma \vdash_{\mathrm{T}} \varphi_2[\varphi_3/a]\ ::\ \kappa_2[\varphi_3/a]$ (Sub. Type/Type 5 2 3)

(10) $\kappa_{11}[\varphi_3/a] \sim_\Sigma \kappa_2[\varphi_3/a]$ (Def. Sub, Def. ($\sim$), 6)

(11) $\Gamma[\varphi_3/a] \mid \Sigma \vdash\ t_1[\varphi_3/a]\ \ \varphi_2[\varphi_3/a]\ ::\ (\varphi_{12}[\varphi_3/a])[\varphi_2[\varphi_3/a]/a_1]\ ;\ (\sigma_1[\varphi_3/a])[\varphi_2[\varphi_3/a]/a_1]$ (TyAppT 8 9 10)

(12) $a \neq a_1$ (No Var Capture 4)

(13) $\Gamma[\varphi_3/a] \mid \Sigma \vdash\ (t_1\ \varphi_2)[\varphi_3/a]\ ::\ (\varphi_{12}[\varphi_2/a_1])[\varphi_3/a]\ \ ;\ (\sigma_1[\varphi_2/a_1])[\varphi_3/a]$ (Def. Sub. 11 12)

Case: $t = \lambda(x : \tau_{11}).\, t_{12}$ / TyAbs
Case: $t = (t_1\ t_2)$ / TyApp
Case: $t = (\textbf{let}\ x = t_1\ \textbf{in}\ t_2)$ / TyLet

Similarly to TyAbsT Case

Case: $t = \textbf{letregion}\ r\ \textbf{with}\ \{\overline{w_i = \delta_i}\}\ \textbf{in}\ t_1$ / TyLetRegion

$$\frac{(4)\ \Gamma,\ a : \kappa_3,\ r : \%,\ \overline{w_i = \kappa_i} \mid \Sigma \vdash t_1 :: \tau\,;\,\sigma \qquad (6)\ \overline{\delta_i}\ \text{well formed} \qquad (5)\ \Gamma,\ a : \kappa_3 \mid \Sigma \vdash_{\mathrm{T}} \delta_i :: \kappa_i \qquad (7)\ \Gamma \mid \Sigma \vdash_{\mathrm{K}} \kappa_i :: \Diamond}{(\underline{1})\ \Gamma,\ a : \kappa_3 \mid \Sigma \vdash \textbf{letregion}\ r\ \textbf{with}\ \{\overline{w_i = \delta_i}\}\ \textbf{in}\ t_1 :: \sigma\,;}$$

($\underline{2}$)	$\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_2 :: \kappa_2$	(assume)
($\underline{3}$)	$\kappa_3 \sim_\Sigma \kappa_2$	(assume)
(8)	$\Gamma,\ r : \%,\ \overline{w_i : \kappa_i} \mid \Sigma \vdash_{\mathrm{T}} \varphi_2 :: \kappa_2$	(Weak. Type Env 2)
(9)	$(\Gamma,\ r : \%,\ \overline{w_i : \kappa_i})[\varphi_2/a] \mid \Sigma \vdash t[\varphi_2/a] :: \tau[\varphi_2/a]\,;\,\sigma[\varphi_2/a]$	(IH 4 8 3)
(10)	$\Gamma[\varphi_2/a],\ r : \%,\ (\overline{w_i : \kappa_i[\varphi_2/a]}) \mid \Sigma \vdash t[\varphi_2/a] :: \tau[\varphi_2/a]\ ;\ \sigma[\varphi_2/a]$	(Def. Sub. 9)
(11)	$\Gamma[\varphi_2/a] \mid \Sigma \vdash_{\mathrm{T}} \delta_i[\varphi_2/a] :: \kappa_i[\varphi_2/a]$	(Sub. Type/Type 5 2 3)
(12)	$\Gamma \mid \kappa_i[\varphi_2/a] \vdash_{\mathrm{K}} \Diamond ::$	(Insp. Kinding Rules)
(13)	$\overline{\delta_i}[\varphi_2/a]$ well formed	(Def. Well Formed 6)
(14)	$\Gamma[\varphi_2/a] \mid \Sigma \vdash \textbf{letregion}\ r\ \textbf{with}\ \{\overline{w_i = \delta_i[\varphi_2/a]}\}\ \textbf{in}\ t[\varphi_2/a] :: \tau[\varphi_2/a]\,;\sigma[\varphi_2/a]$	(TyLetRegion 10..13)
(15)	$\Gamma[\varphi_2/a] \mid \Sigma \vdash (\textbf{letregion}\ r\ \textbf{with}\ \{\overline{w_i = \delta_i}\}\ \textbf{in}\ t)[\varphi_2/a] :: \tau[\varphi_2/a]\,;\sigma[\varphi_2/a]$	(Def. Sub. 14)

Case: $t = \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3$ / TyIf

Similarly to TyAbsT case.

Case: $t = \mathit{True}\ \varphi$ / TyTrue

$$\frac{(4)\ \Gamma,\ a : \kappa_3 \mid \Sigma \vdash_{\mathrm{T}} \varphi :: \%}{(\underline{1})\ \Gamma,\ a : \kappa_3 \mid \Sigma \vdash \mathit{True}\ \varphi :: \mathit{Bool}\ \varphi\,;\,\bot}$$

($\underline{2}$)	$\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_2 :: \kappa_2$	(assume)
($\underline{3}$)	$\kappa_3 \sim_\Sigma \kappa_2$	(assume)
(5)	$\Gamma[\varphi_2/a] \mid \Sigma \vdash_{\mathrm{T}} \varphi[\varphi_2/a] :: \%[\varphi_2/a]$	(Sub Type/Type 4 2 3)
(6)	$\Gamma[\varphi_2/a] \mid \Sigma \vdash_{\mathrm{T}} \varphi[\varphi_2/a] :: \%$	(Def. Sub. 5)
(7)	$\Gamma[\varphi_2/a] \mid \Sigma \vdash \mathit{True}\ (\varphi[\varphi_2/a]) :: \mathit{Bool}\ (\varphi[\varphi_2/a])\,;\,\bot$	(TyTrue 6)
(8)	$\Gamma[\varphi_2/a] \mid \Sigma \vdash (\mathit{True}\ \varphi)[\varphi_2/a] :: (\mathit{Bool}\ \varphi)[\varphi_2/a]\,;\,\bot$	(Def. Sub. 7)

Case: $t = \mathit{False}\ \varphi$

Similarly to TyTrue Case.

Case: $t = \mathit{update}\ \delta\ t_1\ t_2$ / TyUpdate

$$\frac{(4)\ \Gamma,\ a : \kappa_4 \mid \Sigma \vdash_{\mathrm{T}} \delta\ ::\ \mathit{Mutable}\ \sigma \qquad \begin{array}{r}(5)\ \Gamma,\ a : \kappa_4 \mid \Sigma \vdash\ t_1\ ::\ \mathit{Bool}\ \varphi_1\ ;\ \sigma_1 \\ (6)\ \Gamma,\ a : \kappa_4 \mid \Sigma \vdash\ t_2\ ::\ \mathit{Bool}\ \varphi_2\ ;\ \sigma_2\end{array}}{(\underline{1})\ \Gamma,\ a : \kappa_4 \mid \Sigma \vdash\ \mathit{update}\ \delta\ t_1\ t_2\ ::\ ()\ ;\ \sigma_1 \vee \sigma_2 \vee \mathit{Read}\ \varphi_2 \vee \mathit{Write}\ \varphi_1}$$

($\underline{2}$)	$\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_3\ ::\ \kappa_3$	(assume)
($\underline{3}$)	$\kappa_4 \sim_\Sigma \kappa_3$	(assume)
(7)	$\Gamma,\ a : \kappa_4 \mid \Sigma \vdash_{\mathrm{T}} \varphi_3\ ::\ \kappa_3$	(Weak. Type Env 2)
(8)	$\Gamma[\varphi_3/a] \mid \Sigma \vdash\ t_1[\varphi_3/a]\ ::\ \mathit{Bool}\ (\varphi_1[\varphi_3/a])\ ;\ \sigma_1[\varphi_3/a]$	(IH, Def. Sub. 5 7 3)
(9)	$\Gamma[\varphi_3/a] \mid \Sigma \vdash\ t_2[\varphi_3/a]\ ::\ \mathit{Bool}\ (\varphi_2[\varphi_3/a])\ ;\ \sigma_2[\varphi_3/a]$	(IH, Def. Sub. 6 7 3)
(10)	$\Gamma[\varphi_3/a] \mid \Sigma \vdash_{\mathrm{T}} \delta[\varphi_3/a]\ ::\ (\mathit{Mutable}\ \delta)[\varphi_3/a]$	(Sub. Type/Type 4 2 3)
(11)	$\Gamma[\varphi_3/a] \mid \Sigma \vdash_{\mathrm{T}} \delta[\varphi_3/a]\ ::\ \mathit{Mutable}\ (\delta[\varphi_3/a])$	(Def. Sub. 10)
(12)	$\Gamma[\varphi_3/a] \mid \Sigma \vdash\ \mathit{update}\ (\delta[\varphi_3/a])\ (t_1[\varphi_3/a])\ (t_2[\varphi_3/a])$ $::\ ()$ $;\ \sigma_1[\varphi_3/a] \vee \sigma_2[\varphi_3/a] \vee \mathit{Read}\ (\varphi_2[\varphi_3/a])$ $\vee\ \mathit{Write}\ (\varphi_1[\varphi_3/a])$	(TyUpdate 8 9 11)
(13)	$\Gamma[\varphi_3/a] \mid \Sigma \vdash\ (\mathit{update}\ \delta\ t_1\ t_2)[\varphi_3/a]$ $::\ ()$ $;\ (\sigma_1 \vee \sigma_2 \vee \mathit{Read}\ \varphi_2 \vee \mathit{Write}\ \varphi_1)[\varphi_3/a]$	(Def. Sub. 12)

Case: $t = \mathit{suspend}\ \delta\ t_1\ t_2$ / TySuspend

Similarly to TyApp / TyUpdate case.

Case: $t = ()$ / TyUnit
Case: $t = \underline{l}$ / TyLoc

Trivial. No free type vars.

Lemma: (Substitution of Types in Types)

If $\Gamma,\ a : \kappa_3 \mid \Sigma \vdash_{\mathrm{T}} \varphi_1 \ ::\ \kappa_1$
and $\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_2 \ ::\ \kappa_2$
and $\kappa_3 \sim_\Sigma \kappa_2$
then $\Gamma[\varphi_2/a] \mid \Sigma \vdash_{\mathrm{T}} \varphi_1[\varphi_2/a] \ ::\ \kappa_1[\varphi_2/a]$

Proof: by induction over the derivation of $\Gamma,\ a : \kappa_3 \mid \Sigma \vdash_{\mathrm{T}} \varphi_1 \ ::\ \kappa_1$

(IH) Substitution holds for all subterms of φ_1. (assume)

Case: $\varphi = a$ / KiVar

Similarly to Subst Var/Var TyVar case.

Case: $\varphi = \forall(b : \kappa_1).\, \tau$ / KiAll

$$\frac{(4)\ \Gamma,\ a : \kappa_4 \mid \Sigma \vdash_{\mathrm{T}} \sigma \ ::\ \kappa_1 \quad (5)\ \Gamma,\ a : \kappa_4,\ b : \kappa_1 \mid \Sigma \vdash_{\mathrm{T}} \tau_1 \ ::\ \kappa_2}{(\underline{1})\ \Gamma,\ a : \kappa_4 \mid \varphi_1 \vdash_{\mathrm{T}} \forall(b : \kappa_1).\, \tau_1 \ ::\ \kappa_2}$$

($\underline{2}$)	$\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_3 \ ::\ \kappa_3$	(assume)
($\underline{3}$)	$\kappa_4 \sim_\Sigma \kappa_3$	(assume)
(6)	$\Gamma[\varphi_3/a] \mid \Sigma \vdash_{\mathrm{T}} \varphi_1[\varphi_3/a] \ ::\ \kappa_1[\varphi_3/a]$	(IH 4 2 3)
(7)	$\Gamma,\ b : \kappa_1 \mid \Sigma \vdash_{\mathrm{T}} \varphi_3 \ ::\ \kappa_3$	(Weak. Type Env 2)
(8)	$\Gamma[\varphi_3/a],\ b : \kappa_1[\varphi_3/a] \mid \Sigma \vdash_{\mathrm{T}} \tau_1[\varphi_3/a] \ ::\ \kappa_2[\varphi_3/a]$	(IH, Def. Subst 5 7 3)
(9)	$b \notin fv(\Gamma[\varphi_3/a])$	(Uniqueness of Vars)
(10)	$\Gamma[\varphi_3/a] \mid \Sigma \vdash_{\mathrm{T}} (\forall(b : \kappa_1).\, \tau_1)[\varphi_3/a] \ ::\ \kappa_2[\varphi_3/a]$	(KiAll, Def. Sub. 6 8 9)

Case: $\varphi = \varphi_1\ \varphi_2$ / KiApp

$$\frac{(4)\ \Gamma,\ a : \kappa_4 \mid \Sigma \vdash_{\mathrm{T}} \varphi_1 \ ::\ \Pi(b : \kappa_{11}).\, \kappa_{12} \quad (5)\ \Gamma,\ a : \kappa_4 \mid \Sigma \vdash_{\mathrm{T}} \varphi_2 \ ::\ \kappa_{11}}{(\underline{1})\ \Gamma,\ a : \kappa_4 \mid \Sigma \vdash_{\mathrm{T}} \varphi_1\ \varphi_2 \ ::\ \kappa_{12}[\varphi_2/a]}$$

($\underline{2}$)	$\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_3 \ ::\ \kappa_3$	(assume)
($\underline{3}$)	$\kappa_4 \sim_\Sigma \kappa_3$	(assume)
(6)	$\Gamma[\varphi_3/a] \mid \Sigma \vdash_{\mathrm{T}} \varphi_1[\varphi_3/a] \ ::\ \Pi(b : \kappa_{11}[\varphi_3/a]).\, (\kappa_{12}[\varphi_3/a])$	(IH, Def. Sub. 4 2 3)
(7)	$\Gamma[\varphi_3/a] \mid \Sigma \vdash_{\mathrm{T}} \varphi_2[\varphi_3/a] \ ::\ \kappa_{11}[\varphi_3/a]$	(IH 5 2 3)
(8)	$\Gamma[\varphi_3/a] \mid \Sigma \vdash_{\mathrm{T}} \varphi_1[\varphi_3/a]\ \varphi_2[\varphi_3/a] :: (\kappa_{12}[\varphi_3/a])[\varphi_2[\varphi_3/a]/b]$	(KiApp 6 7)
(9)	$b \notin fv(\varphi_3)$	(Uniqueness of Var)
(10)	$\Gamma[\varphi_3/a] \mid \Sigma \vdash_{\mathrm{T}} (\varphi_1\ \varphi_2)[\varphi_3/a] \ ::\ (\kappa_{12}[\varphi_2/b])[\varphi_3/a]$	(Def. Sub. 8)

The remaining cases are similar to the KiApp case.

Theorem: (Progress)
Suppose we have a state H ; t with store H and term t. Let Σ be a store typing which models H. If H is well typed, and t is closed and well typed, and t contains no fabricated region witnesses, then either t is a value or H ; t can transition to the next state.

If	$\emptyset \mid \Sigma \vdash t :: \tau ; \sigma$
and	$\Sigma \models \mathrm{H}$
and	$\Sigma \vdash \mathrm{H}$
and	nofab(t)
then	$t \in$ Value
or	for some H′, t' we have
	$(\mathrm{H} ; t \longrightarrow \mathrm{H}' ; t'$ and nofab(t'))

Proof: By induction over the derivation of $\emptyset \mid \Sigma \vdash t :: \tau ; \sigma$

Let (H ; t can step) ≡ (for some H, t we have H ; $t \longrightarrow$ H′ ; t' and nofab(t'))

We will not formally prove nofab(t') in the conclusion of each case. This property can be verified by inspecting (EvLetRegion) and noting that unevaluated applications of witness constructors are not substituted into the body of the term.

(IH) Progress holds for all subterms of t. (assume)

Case: t is one of x, $\Lambda(a :: \kappa).\, t'$, $\lambda(x :: \tau).\, t'$, $()$, $\underline{l}$

$t \in$ Value

Case: $t = (t_1\ \varphi_2)$ / TyAppT

$$\frac{(5)\ \emptyset \mid \Sigma \vdash t_1 :: \forall(a : \kappa_{11}).\, \varphi_{12} ; \sigma \quad (6)\ \emptyset \mid \Sigma \vdash_{\mathrm{T}} \varphi_2 :: \kappa_2 \quad (7)\ \kappa_{11} \sim_\Sigma \kappa_2}{(\underline{1})\ \emptyset \mid \Sigma \vdash t_1\ \varphi_2 :: \varphi_{12}[\varphi_2/a] ; \sigma[\varphi_2/a]}$$

($\underline{2..4}$)	$\Sigma \models \mathrm{H}$, $\Sigma \vdash \mathrm{H}$, nofab(t)	(assume)
(5)	$t_1 \in$ Value or H ; t_1 can step	(IH 5 2..4)
(6)	*Case:* $t_1 \in$ Value	
(7)	$t_1 = \Lambda(a : \kappa_{11}).\, t_{12}$	(Forms of Terms 6 5)
(8)	H ; t can step	(EvTAppAbs 6)
(8)	*Case:* H ; t_1 can step	
(9)	H ; t can step	(EvTApp1 7)

Case: $t = t_1\ t_2$ / TyApp

$$\frac{(5)\ \emptyset \mid \Sigma \vdash t_1 :: \tau_{11} \xrightarrow{\sigma} \tau_{12} ; \sigma_1 \qquad (6)\ \emptyset \mid \Sigma \vdash t_2 :: \tau_2 ; \sigma_2 \qquad (7)\ \tau_{11} \sim_\Sigma \tau_2}{(\underline{1})\ \emptyset \mid \Sigma \vdash t_1\ t_2 :: \tau_{12} ; \sigma_1 \vee \sigma_2 \vee \sigma}$$

(<u>2..4</u>)	$\Sigma \models \mathrm{H},\ \Sigma \vdash \mathrm{H},\ \mathrm{nofab}(t)$	(assume)
(8)	$t_1 \in \mathrm{Value}$ or $\mathrm{H} ; t_1$ can step	(IH 5 2..4)
(9)	$t_2 \in \mathrm{Value}$ or $\mathrm{H} ; t_2$ can step	(IH 6 2..4)
(10)	*Case* : $\mathrm{H} ; t_1$ can step	
(11)	$\mathrm{H} ; t$ can step	(EvApp1 10)
(12, 13)	*Case* : $t_1 \in \mathrm{Value},\ \mathrm{H} ; t_2$ can step	
(14)	$\mathrm{H} ; t$ can step	(EvApp2 12 13)
(15, 16)	*Case* : $t_1 \in \mathrm{Value},\ t_2 \in \mathrm{Value}$	
(17)	$t_1 = \lambda(x : \tau_{11}).\, t_{12}$	(Forms of Terms 15 5)
(18)	$\mathrm{H} ; t$ can step	(EvAppAbs 17 16)

Case: $t = (\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2)$ / TyLet

Similarly to TyApp case.

Case: $t = (\mathbf{letregion}\ r\ \mathbf{with}\ \{\overline{w_i = \delta_i}\}\ \mathbf{in}\ t_1)$ / TyLetRegion

(<u>1</u>)	$\emptyset \mid \Sigma \vdash (\mathbf{letregion}\ r\ \mathbf{with}\ \{\overline{w_i = \delta_i}\}\ \mathbf{in}\ t_1) :: \tau_1 ; \sigma_1$	(assume)
(<u>2..4</u>)	$\Sigma \models \mathrm{H},\ \Sigma \vdash \mathrm{H},\ \mathrm{nofab}(t)$	(assume)
(5)	$\mathrm{H},\ \overline{\mathrm{propOf}(\Delta_i)}\ ;\ \overline{\delta_i \rightsquigarrow \Delta_i}$	(Region Wit. Assert)
(6)	$\mathrm{H} ; t$ can step	(EvLetRegion 5)

Case: $t = (\textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3)$ / TyIf

$$\frac{(5)\ \emptyset \mid \Sigma \vdash t_1 :: \mathit{Bool}\ \varphi\,;\,\sigma_1 \quad (6)\ \emptyset \mid \Sigma \vdash t_2 :: \tau_2\,;\,\sigma_2 \quad (7)\ \emptyset \mid \Sigma \vdash t_3 :: \tau_3\,;\,\sigma_3 \quad (8)\ \tau_2 \sim_\Sigma \tau_3}{(\underline{1})\ \emptyset \mid \Sigma \vdash (\textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3) :: \tau_2\,;\,(\sigma_1 \vee \sigma_2 \vee \sigma_3 \vee \mathit{Read}\ \varphi)}$$

($\underline{2..4}$)	$\Sigma \models \mathrm{H},\ \Sigma \vdash \mathrm{H},\ \mathrm{nofab}(t)$	(assume)
(9)	$t_1 \in \mathrm{Value}$ or $\mathrm{H}\,;\,t_1$ can step	(IH 5 2..4)
(10)	*Case* : $t_1 \in \mathrm{Value}$	
(11)	$t_1 = \underline{l}$	(Forms of Terms 10 5)
(12)	$\emptyset \mid \Sigma \vdash \underline{l} :: \mathit{Bool}\ \varphi\,;\,\bot$	(5 11)
(13)	$\underline{l} : \mathit{Bool}\ r \in \Sigma$	(TyLoc 12)
(14)	$l \overset{\rho}{\mapsto} \mathrm{V} \in \mathrm{H}$ for some $\mathrm{V} \in \{\mathrm{T}, \mathrm{F}\}$	(Def. Store Model 2 13)
(15)	$H\,;\,t$ can step	(EvIfThen, EvIfElse 14)
(16)	*Case* : $H\,;\,t_1$ can step	
(17)	$H\,;\,t$ can step	(EvIf 16)

Case: $t = (\mathit{True}\ \varphi_1)$ / TyTrue

$$\frac{(5)\ \emptyset \mid \Sigma \vdash_{\mathrm{T}} \varphi_1 :: \%}{(\underline{1})\ \emptyset \mid \Sigma \vdash \mathit{True}\ \varphi_1 :: \mathit{Bool}\ \varphi_1\,;\,\bot}$$

($\underline{2..4}$)	$\Sigma \models \mathrm{H},\ \Sigma \vdash \mathrm{H},\ \mathrm{nofab}(t)$	(assume)
(6)	$\varphi_1 = \underline{\rho}$	(Forms of Types, t is closed, 1)
(7)	$\emptyset \mid \Sigma \vdash_{\mathrm{T}} \underline{\rho} :: \%$	(5 6)
(8)	$\underline{\rho} \in \Sigma$	(KiHandle 7)
(9)	$\rho \in \mathrm{H}$	(Def. Store Model 2 8)
(10)	$H\,;\,t$ can step	(EvTrue 9 6)

Case: $t = (\mathit{False}\ \varphi_1)$ / TyFalse

Similarly to TyTrue case.

Case: $t = (\mathit{update}\ \delta\ t_1\ t_2)$ / TyUpdate

$$\frac{(5)\ \emptyset \mid \Sigma \vdash_{\mathrm{T}} \delta\ ::\ \mathit{Mutable}\ \varphi_1 \qquad \begin{array}{l}(6)\ \emptyset \mid \Sigma \vdash t_1\ ::\ \mathit{Bool}\ \varphi_1\ ;\ \sigma_1 \\ (7)\ \emptyset \mid \Sigma \vdash t_2\ ::\ \mathit{Bool}\ \varphi_2\ ;\ \sigma_2\end{array}}{(\underline{1})\ \emptyset \mid \Sigma \vdash (\mathit{update}\ \delta\ t_1\ t_2)\ ::\ ()\ ;\ (\sigma_1 \vee \sigma_2 \vee \mathit{Read}\ \varphi_2 \vee \mathit{Write}\ \varphi_1)}$$

(2..4)	$\Sigma \models \mathrm{H},\ \Sigma \vdash \mathrm{H},\ \mathrm{nofab}(t)$	(assume)
(8, 9)	*Case* : $t_1 \in \mathrm{Value},\ \ t_2 \in \mathrm{Value}$	
(10)	$\delta = \underline{\mathrm{mutable}\ \rho_1}$	(Forms of Types 4 5)
(11)	$\underline{\mathrm{mutable}\ \rho_1} \in \Sigma$	(KiMutable 5 10)
(12)	$\mathrm{mutable}\ \rho_1 \in \mathrm{H}$	(Def. Store Model 2 11)
(13, 14)	$t_1 = \underline{l_1},\ t_2 = \underline{l_2}$	(Forms of Terms 8 9 6 7)
(15)	$l_1 \overset{\rho_1'}{\mapsto} \mathrm{V}_1' \in \mathrm{H}$ for some $\mathrm{V}_1' \in \{\mathrm{T}, \mathrm{F}\}$	(as per TyIf case 2 6 13)
(16)	$l_2 \overset{\rho_2}{\mapsto} \mathrm{V}_2 \in \mathrm{H}$ for some $\mathrm{V}_2 \in \{\mathrm{T}, \mathrm{F}\}$	(as per TyIf case 2 7 14)
(17)	$\underline{\rho_1} = \varphi_1$	(KiMutable 5 10)
(18)	$\emptyset \mid \Sigma \vdash \underline{l_1}\ ::\ \mathit{Bool}\ \underline{\rho_1}\ ;\ \bot$	(6 13 17)
(19)	$\underline{l_1} : \mathit{Bool}\ \underline{\rho_1} \in \Sigma$	(Tyloc 18)
(20)	$l_1 \overset{\rho_1}{\mapsto} \mathrm{V}_1 \in \mathrm{H}$ for some $\mathrm{V}_1 \in \{\mathrm{T}, \mathrm{F}\}$	(Def. Store Model 2 19)
(21)	$\underline{\rho_1} = \underline{\rho_1'}$	(Def. Store 15 20)
(22)	$\mathrm{H}\ ;\ t$ can step	(EvUpdate3, 12 20 16 10 13 14)

Other cases via EvUpdate1 or EvUpdate2 as per TyApp case.

Case: $t = (\mathit{suspend}\ \delta\ t_1\ t_2)$ / TySuspend

$$\frac{\begin{array}{l}(5)\ \tau_{11} \sim_\Sigma \tau_2 \\ (6)\ \emptyset \mid \Sigma \vdash_{\mathrm{T}} \delta\ ::\ \mathit{Pure}\ \sigma\end{array} \qquad \begin{array}{l}(7)\ \emptyset \mid \Sigma \vdash t_1\ ::\ \tau_{11} \overset{\sigma}{\to} \tau_{12}\ ;\ \sigma_1 \\ (8)\ \emptyset \mid \Sigma \vdash t_2\ ::\ \tau_2\ ;\ \sigma_2\end{array}}{(\underline{1})\ \emptyset \mid \Sigma \vdash (\mathit{suspend}\ \delta\ t_1\ t_2)\ ::\ \tau_{12}\ ;\ \sigma_1 \vee \sigma_2}$$

(2..4)	$\Sigma \models \mathrm{H},\ \Sigma \vdash \mathrm{H},\ \mathrm{nofab}(t)$	(assume)
(9)	$\delta \in \{\mathit{MkPurify}\ \underline{\rho}\ \delta_2,\ \mathit{MkPureJoin}\ \sigma_3\ \sigma_4\ \delta_3\ \delta_4,$ $\mathit{MkPure}\ \bot,\ \underline{\mathrm{pure}\ \sigma}\}$	(Forms of Types 6)
(10)	*Case* : $\delta_1 \in \{\mathit{MkPurify}\ \underline{\rho}\ \delta_2,\ \mathit{MkPureJoin}\ \sigma_3\ \sigma_4\ \delta_3\ \delta_4,\ \mathit{MkPure}\ \bot\}$	
(11)	$\mathrm{H}\ ;\ \delta_1 \rightsquigarrow \delta_1'$	(Progress of Purity 6 3 10)
(12)	$\mathrm{H}\ ;\ t$ can step	(EvSuspend1 11)
(13..15)	*Case* : $\delta_1 = \underline{\mathrm{pure}\ \sigma},\ t_1 \in \mathrm{Value},\ t_2 \in \mathrm{Value}$	
(16)	$t_1 = \lambda(x : \tau).\ t_3$	(Forms of Terms 6 16)
(17)	$\mathrm{H}\ ;\ t_1$ can step	(EvSuspend4 13 16 15)

Other cases via EvSuspend2 or EvSuspend3 as per TyApp case.

Theorem: (Preservation)
Suppose we have a state H ; t with store H and term t. Let Σ be a store typing which models H. If H and t are well typed, and H ; t can transition to a new state H′ ; t' then for some Σ' which models H′, H′ is well typed, t' has a similar type to t, and the effect σ' of t' is no greater than the effect σ of t.

If $\quad \Gamma \mid \Sigma \vdash t :: \tau ; \sigma$
and $\quad \mathrm{H} ; t \longrightarrow \mathrm{H}' ; t'$
and $\quad \Sigma \vdash \mathrm{H}$ and $\Sigma \models \mathrm{H}$

then for some Σ', τ', σ' we have
$\quad \Gamma \mid \Sigma' \vdash t' :: \tau' ; \sigma'$
and $\quad \Sigma' \supseteq \Sigma$ and $\Sigma' \models \mathrm{H}'$ and $\Sigma' \vdash \mathrm{H}'$
and $\quad \tau' \sim_{\Sigma'} \tau$ and $\sigma' \sqsubseteq_{\Sigma'} \sigma$

Proof: By induction over the derivation of $\Gamma \mid \Sigma \vdash t :: \tau ; \sigma$.

(IH)	Progress holds for all subterms of t.	(assume)

Case: t is one of x, $\Lambda(a :: \kappa).\, t'$, $\lambda(x :: \tau).\, t'$, $()$, $\underline{l}$

Can't happen. There is no transition rule for H ; t

Case: $t = t_1\ \varphi_2$ / TyAppT / EvApp1

$$\frac{(5)\ \Gamma \mid \Sigma \vdash t_1 :: \forall(a : \kappa_{11}).\, \varphi_{12} ; \sigma \qquad (6)\ \Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_2 :: \kappa_2 \qquad (7)\ \kappa_{11} \sim_{\Sigma} \kappa_2}{(\underline{1})\ \Gamma \mid \Sigma \vdash t_1\ \varphi_2 :: \varphi_{12}[\varphi_2/a] ; \sigma[\varphi_2/a]}$$

$$\frac{(8)\ \mathrm{H} ; t_1 \longrightarrow \mathrm{H}' ; t_1'}{(\underline{2})\ \mathrm{H} ; t_1\ \varphi_2 \longrightarrow \mathrm{H}' ; t_1'\ \varphi_2}$$

($\underline{3}$, $\underline{4}$)	$\Sigma \models \mathrm{H}$, $\Sigma \vdash \mathrm{H}$	(assume)
(9..14)	$\Gamma \mid \Sigma' \vdash t_1' :: \forall(a : \kappa_{11}').\, \varphi_{12}' ; \sigma'$, $\Sigma' \supseteq \Sigma$, $\sigma' \sqsubseteq_{\Sigma'} \sigma$, $\forall(a : \kappa_{11}').\, \varphi_{12}' \sim_{\Sigma'} \forall(a : \kappa_{11}).\, \varphi_{12}$ $\Sigma' \vdash \mathrm{H}'$, $\Sigma' \models \mathrm{H}'$	(IH 5 8 3 4)
(15)	$\Gamma \mid \Sigma' \vdash_{\mathrm{T}} \varphi_2 :: \kappa_2$	(Weak. Store Typing 6 10)
(16)	$\kappa_{11}' \sim_{\Sigma'} \kappa_{11}$	(SimAll 12)
(17)	$\kappa_{11} \sim_{\Sigma'} \kappa_2$	(Weak. ($\sim_{\Sigma}$) 7 10)
(18)	$\kappa_{11}' \sim_{\Sigma'} \kappa_2$	(16 17)
(19)	$\Gamma \mid \Sigma' \vdash t_1'\ \varphi_2 :: \varphi_{12}'[\varphi_2/a] ; \sigma'[\varphi_2/a]$	(TyAppT 9 13 18)

Case: $t = t_1\ \varphi_2$ / TyAppT / EvAppAbs

$$\dfrac{\dfrac{(8)\ \Gamma,\ a : \kappa_{11} \mid \Sigma \vdash\ t_{12}\ ::\ \varphi_{12}\ ;\ \sigma}{(5)\ \Gamma \mid \Sigma \vdash\ \Lambda(a : \kappa_{11}).\ t_{12}\ ::\ \forall(a : \kappa_{11}).\ \varphi_{12}\ ;\ \sigma} \qquad \begin{array}{r}(7)\ \kappa_{11} \sim_\Sigma \kappa_2\\ (6)\ \Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_2\ ::\ \kappa_2\end{array}}{(\underline{1})\ \Gamma \mid \Sigma \vdash\ (\Lambda(a : \kappa_{11}).\ t_{12})\ \varphi_2\ ::\ \varphi_{12}[\varphi_2/a]\ ;\ \sigma[\varphi_2/a]}$$

$$(\underline{2})\ \mathrm{H}\ ;\ (\Lambda(a :: \kappa_{11}).\ t_{12})\ \varphi_2 \longrightarrow \mathrm{H}\ ;\ t_{12}[\varphi_2/a]$$

($\underline{3}$, $\underline{4}$)	$\Sigma \models \mathrm{H},\ \Sigma \vdash \mathrm{H}$	(assume)
(9)	$\Gamma[\varphi_2/a] \mid \Sigma \vdash\ t_{12}[\varphi_2/a]\ ::\ \varphi_{12}[\varphi_2/a]\ ;\ \sigma[\varphi_2/a]$	(Sub. Type/Value 8 6 7)
(10)	$a \notin \Gamma$	(No Var Capture 1)
(11)	$\Gamma[\varphi_2/a] \equiv \Gamma$	(Def. Sub. 10)

Case: $t = t_1\ t_2$ / TyApp / EvApp1

$$\dfrac{(5)\ \Gamma \mid \Sigma \vdash\ t_1\ ::\ \tau_{11} \xrightarrow{\sigma} \tau_{12}\ ;\ \sigma_1 \qquad (6)\ \Gamma \mid \Sigma \vdash\ t_2\ ::\ \tau_2\ ;\ \sigma_2 \qquad (7)\ \tau_{11} \sim_\Sigma \tau_2}{(\underline{1})\ \Gamma \mid \Sigma \vdash\ t_1\ t_2\ ::\ \tau_{12}\ ;\ \sigma_1 \vee \sigma_2 \vee \sigma}$$

$$\dfrac{(8)\ \mathrm{H}\ ;\ t_1 \longrightarrow \mathrm{H}'\ ;\ t_1'}{(\underline{2})\ \mathrm{H}\ ;\ t_1\ t_2 \longrightarrow \mathrm{H}'\ ;\ t_1'\ t_2}$$

($\underline{3}$, $\underline{4}$)	$\Sigma \models \mathrm{H},\ \Sigma \vdash \mathrm{H}$	(assume)
(9..14)	$\Gamma \mid \Sigma' \vdash\ t_1'\ ::\ \tau_{11}' \xrightarrow{\sigma'} \tau_{12}'\ ;\ \sigma_1'$ $\Sigma' \supseteq \Sigma,\ \Sigma' \vdash \mathrm{H}',\ \ \Sigma' \models \mathrm{H}'$ $(\tau_{11}' \xrightarrow{\sigma'} \tau_{12}') \sim_{\Sigma'} (\tau_{11} \xrightarrow{\sigma} \tau_{12})$ $\sigma_1' \sim_{\Sigma'} \sigma_1$	(IH 5 8 3 4)
(15)	$\Gamma \mid \Sigma' \vdash\ t_2\ ::\ \tau_2\ ;\ \sigma_2$	(Weak. Store Typing 6 10)
(16)	$\tau_{11}' \sim_{\Sigma'} \tau_{11}$	(SimApp 13)
(17)	$\tau_{11} \sim_{\Sigma'} \tau_2$	(Weak. ($\sim_\Sigma$) 7 10)
(18)	$\tau_{11}' \sim_{\Sigma'} \tau_2$	(16 17)
(19)	$\Gamma \mid \Sigma' \vdash\ t_1'\ t_2\ ::\ \tau_{12}'\ ;\ \sigma_1' \vee \sigma_2 \vee \sigma'$	(TyApp 9 15 18)
(20)	$\sigma' \sim_{\Sigma'} \sigma$	(SimApp 13)
(21)	$\sigma_1' \vee \sigma_2 \vee \sigma' \sqsubseteq_{\Sigma'} \sigma_1 \vee \sigma_2 \vee \sigma$	(14 20)

Case: $t = t_1\ t_2$ / TyApp / EvApp2

Similarly to TyApp/EvApp1 case.

Case: $t = t_1\ t_2$ / TyApp / EvAppAbs

$$\dfrac{\dfrac{(8)\ \Gamma,\ x : \tau_{11} \mid \Sigma \vdash\ t_{12}\ ::\ \tau_{12}\ ;\ \sigma}{(5)\ \Gamma \mid \Sigma \vdash\ \lambda(x : \tau_{11}).\, t_{12}\ ::\ \tau_{11} \xrightarrow{\sigma} \tau_{12}\ ;\ \sigma_1} \qquad (6)\ \Gamma \mid \Sigma \vdash\ v^\circ\ ::\ \tau_2\ ;\ \bot \qquad (7)\ \tau_{11} \sim_\Sigma \tau_2}{(\underline{1})\ \Gamma \mid \Sigma \vdash\ (\lambda(x : \tau_{11}).\, t_{12})\ v^\circ\ ::\ \tau_{12}\ ;\ \sigma_1 \vee \sigma_2 \vee \sigma}$$

$$(\underline{2})\ \mathrm{H}\ ;\ (\lambda(x : \tau_{11}).\, t_{12})\ v^\circ\ \longrightarrow\ \mathrm{H}'\ ;\ t_{12}[v^\circ/x]$$

($\underline{3}$,$\underline{4}$)	$\Sigma \models \mathrm{H},\ \Sigma \vdash \mathrm{H}$	(assume)
(9..11)	$\Gamma \mid \Sigma \vdash\ t_{12}[v^\circ/x]\ ::\ \tau'_{12}\ ;\ \sigma'$ $\tau_{12} \sim_\Sigma \tau'_{12}$ $\sigma\ \sim_\Sigma \sigma'$	(Sub. Value/Value 8 6 7)
(12)	$\sigma' \sqsubseteq_\Sigma \sigma_1 \vee \sigma_2 \vee \sigma$	(11)

Case: $t = \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2$ / TyLet / EvLet1

$$\dfrac{(5)\ \Gamma \mid \Sigma \vdash\ t_1\ ::\ \tau_1\ ;\ \sigma_1 \qquad (6)\ \Gamma,\ x : \tau_3 \mid \Sigma \vdash\ t_2\ ::\ \tau_2\ ;\ \sigma_2 \qquad (7)\ \tau_1 \sim_\Sigma \tau_3}{(\underline{1})\ \Gamma \mid \Sigma \vdash\ \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ ::\ \tau_2\ ;\ \sigma_1 \vee \sigma_2}$$

$$\dfrac{(8)\ \mathrm{H}\ ;\ t_1\ \longrightarrow\ \mathrm{H}'\ ;\ t'_1}{(\underline{2})\ \mathrm{H}\ ;\ \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \longrightarrow\ \mathrm{H}'\ ;\ \mathbf{let}\ x = t'_1\ \mathbf{in}\ t_2}$$

($\underline{3}$, $\underline{4}$)	$\Sigma \models \mathrm{H},\ \Sigma \vdash \mathrm{H}$	(assume)
(9..14)	$\Gamma \mid \Sigma' \vdash\ t_1\ ::\ \tau'_1\ ;\ \sigma'_1$ $\Sigma' \supseteq \Sigma,\ \Sigma' \vdash \mathrm{H}',\ \Sigma' \models \mathrm{H}'$ $\tau'_1 \sim_{\Sigma'} \tau_1,\ \sigma'_1 \sqsubseteq_{\Sigma'} \sigma_1$	(IH 5 2 3 4)
(15)	$\tau_1 \sim_{\Sigma'} \tau_3$	(Weak. ($\sim_\Sigma$) 7 10)
(16)	$\tau'_1 \sim_{\Sigma'} \tau_3$	(13 15)
(17)	$\Gamma,\ x : \tau_3 \mid \Sigma' \vdash\ t_2\ ::\ \tau_2\ ;\ \sigma_2$	(Weak. Store Typing 6 10)
(18)	$\Gamma \mid \Sigma' \vdash\ \mathbf{let}\ x = t'_1\ \mathbf{in}\ t_2\ ::\ \tau_2\ ;\ \sigma'_1 \vee \sigma_2$	(TyLet 9 17 16)
(20)	$\sigma'_1 \vee \sigma_2\ \sqsubseteq_{\Sigma'}\ \sigma_1 \vee \sigma_2$	(14)

Case: $t = \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2$ / TyLet / EvLet

Similarly to TyApp / EvAppAbs case.

Case: $t = \textbf{letregion}\ r\ \textbf{with}\ \{\overline{w_i = \delta_i}\}\ \textbf{in}\ t_1$ / TyLetRegion / EvLetRegion

$$\frac{(5)\ \Gamma,\ r : \%,\ \overline{w_i : \kappa_i} \mid \Sigma \vdash t_1 :: \tau\ ;\ \sigma \qquad (6)\ \Gamma \mid \Sigma \vdash_{\mathrm{K}} \kappa_i :: \Diamond \qquad (7)\ \Gamma \mid \Sigma \vdash_{\mathrm{T}} \delta_i :: \kappa_i \qquad (8)\ \overline{\delta_i}\ \text{well formed}}{(\underline{1})\ \Gamma \mid \Sigma \vdash (\textbf{letregion}\ r\ \textbf{with}\ \{\overline{w_i = \delta_i}\}\ \textbf{in}\ t_1) :: \tau\ ;\ \sigma}$$

$$\frac{(9)\ \mathrm{H},\ \overline{\mathrm{propOf}(\Delta_i)}\ ;\ \overline{\delta_i} \rightsquigarrow \overline{\Delta_i} \qquad (10)\ \underline{\rho}\ \text{fresh}}{(\underline{2})\ \mathrm{H}\ ;\ \textbf{letregion}\ r\ \textbf{with}\ \{\overline{w_i = \delta_i}\}\ \textbf{in}\ t_1 \longrightarrow \mathrm{H},\ \underline{\rho},\ r \sim \rho,\ \overline{\Delta_i}\ ;\ t_1[\overline{\Delta_i/w_i}][\underline{\rho}/r]}$$

($\underline{3}$, $\underline{4}$)	$\Sigma \models \mathrm{H},\ \Sigma \vdash \mathrm{H}$	(assume)
(11)	$\Gamma[\overline{\Delta_i/w_i}][\underline{\rho}/r] \mid \Sigma \vdash t_1[\overline{\Delta_i/w_i}][\underline{\rho}/r] :: \tau_1[\overline{\Delta_i/w_i}][\underline{\rho}/r]\ ;\ \sigma[\overline{\Delta_i/w_i}][\underline{\rho}/r]$	(Sub. Type/Value 5 7)
(12)	$\Gamma[\overline{\Delta_i/w_i}][\underline{\rho}/r] \equiv \Gamma$	(No Var Capture 1)
(13)	$\sigma[\overline{\Delta_i/w_i}][\underline{\rho}/r] \equiv \sigma[\underline{\rho}/r]$	(No Wit. Vars in Effects 7 6)
(14)	$\tau[\overline{\Delta_i/w_i}][\underline{\rho}/r] \equiv \tau[\underline{\rho}/r]$	(No Wit. Vars in Value Types 7 6)
(15)	$\Sigma' = \Sigma,\ \underline{\rho},\ r \sim \rho,\ \overline{\Delta_i}$	(let)
(16)	$\Gamma \mid \Sigma' \vdash t_1[\overline{\Delta_i/w_i}][\underline{\rho}/r] :: \tau[\underline{\rho}/r]\ ;\ \sigma[\underline{\rho}/r]$	(Weak. Store Typing 11..15)
(17)	$\tau \sim_{\Sigma'} \tau[\underline{\rho}/r]$	(SimHandle 15)
(18)	$\sigma \sqsubseteq_{\Sigma'} \sigma[\underline{\rho}/r]$	(SubReflSim, SimHandle 15)

Case: $t = \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3$ / TyIf / EvIf

Similarly to TyApp / EvApp1 case.

Case: $t = \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3$ / TyIf / EvIfThen

$$\frac{(5)\ \Gamma \mid \Sigma \vdash \underline{l} :: \mathit{Bool}\ \rho\ ;\ \sigma_1 \qquad (6)\ \Gamma \mid \Sigma \vdash t_2 :: \tau_2\ ;\ \sigma_2 \qquad (7)\ \Gamma \mid \Sigma \vdash t_3 :: \tau_3\ ;\ \sigma_3 \qquad \tau_2 \sim_\Sigma \tau_3}{(\underline{1})\ \Gamma \mid \Sigma \vdash \textbf{if}\ \underline{l}\ \textbf{then}\ t_2\ \textbf{else}\ t_3 :: \tau_2\ ;\ \sigma_1 \vee \sigma_2 \vee \sigma_3 \vee \mathit{Read}\ \rho}$$

$$(\underline{2})\ \mathrm{H},\ l \overset{\rho}{\mapsto} \mathrm{T}\ ;\ \textbf{if}\ \underline{l}\ \textbf{then}\ t_2\ \textbf{else}\ t_3 \longrightarrow \mathrm{H},\ l \overset{\rho}{\mapsto} \mathrm{T}\ ;\ t_2$$

($\underline{3}$, $\underline{4}$)	$\Sigma \models \mathrm{H},\ \Sigma \vdash \mathrm{H}$	(assume)
(8)	$\Gamma \mid \Sigma \vdash t_2 :: \tau_2\ ;\ \sigma_2$	(Repeat 6)
(9)	$\sigma_2 \sqsubseteq_\Sigma \sigma_1 \vee \sigma_2 \vee \sigma_3 \vee \mathit{Read}\ \rho$	(SubJoin2)

Case: $t = \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3$ / TyIf / EvIfElse

Similarly to TyIf / EvIfThen case.

Case: $t = \mathit{True}\ \varphi$ / TyTrue / EvTrue

$$\dfrac{\dfrac{(6)\ \underline{\rho} \in \Sigma}{(5)\ \Gamma \mid \Sigma \vdash_{\mathrm{T}} \underline{\rho} :: \%}}{(\underline{1})\ \Gamma \mid \Sigma \vdash \mathit{True}\ \underline{\rho} :: \mathit{Bool}\ \underline{\rho}\,;\ \bot}$$

$$\dfrac{\underline{l}\ \text{fresh}}{(\underline{2})\ \mathrm{H},\ \underline{\rho}\,;\ \mathit{True}\ \underline{\rho} \longrightarrow \mathrm{H},\ \underline{\rho},\ \underline{l} \overset{\rho}{\mapsto} \mathrm{T}\,;\ \underline{l}}$$

$(\underline{3}, \underline{4})$	$\Sigma \models \mathrm{H},\ \Sigma \vdash \mathrm{H}$	(assume)
(7)	$\Sigma' = \Sigma,\ l : \mathit{Bool}\ \underline{\rho}$	(let)
(8)	$\Gamma \mid \Sigma' \vdash \underline{l} :: \mathit{Bool}\ \underline{\rho}\,;\ \bot$	(TyLoc 7)

Case: $t = \mathit{False}\ \varphi$ / TyFalse / EvFalse

Similarly to TyFalse / EvFalse case.

Case: $t = \mathit{update}\ \delta\ t_1\ t_2$ / TyUpdate / {EvUpdate1, EvUpdate2}

Similarly to TyApp / EvApp1 case.

Case: $t = \mathit{update}\ \delta\ t_1\ t_2$ / TyUpdate / EvUpdate3

$$(\underline{2})\quad \mathrm{H},\ \text{mutable}\ \rho_1,\ l_1 \overset{\rho_1}{\mapsto} V_1,\ l_2 \overset{\rho_2}{\mapsto} V_2\ \textbf{;}\ \mathit{update}\ \underline{\text{mutable}\ \rho_1}\ \underline{l_1}\ \underline{l_2}$$
$$\longrightarrow \mathrm{H},\ \text{mutable}\ \rho_1,\ l_1 \overset{\rho_1}{\mapsto} V_2,\ l_2 \overset{\rho_2}{\mapsto} V_2\ \textbf{;}\ ()$$

$(\underline{1})$	$\Gamma \mid \Sigma \vdash \mathit{update}\ \underline{\text{mutable}\ \rho_1}\ \underline{l_1}\ \underline{l_2} :: ()$ $;\ \sigma_1 \vee \sigma_2 \vee \mathit{Read}\ \rho_1 \vee \mathit{Read}\ \rho_2$	(assume)
$(\underline{3}, \underline{4})$	$\Sigma \models \mathrm{H},\ \Sigma \vdash \mathrm{H}$	(assume)
(5)	$\Gamma \mid \Sigma \vdash () :: ()\,;\ \bot$	(TyUnit)

Case: $t = \mathit{suspend}\ \delta\ t_1\ t_2$ / TySuspend / EvSuspend1

Immediate

Case: $t = \mathit{suspend}\ \delta\ t_1\ t_2$ / TySuspend / {EvSuspend2, EvSuspend3}

Similarly to TyApp / EvApp1 case.

Case: $t = \mathit{suspend}\ \delta\ t_1\ t_2$ / TySuspend / EvSuspend

$$\dfrac{\dfrac{(9)\ \Gamma,\ x : \tau_{11} \mid \Sigma \vdash\ t_{12}\ ::\ \tau_{12}\ ;\ \sigma}{(5)\ \Gamma \mid \Sigma \vdash\ \lambda(x : \tau_{11}).\, t_{12}\ ::\ \tau_{11} \xrightarrow{\sigma} \tau_{12}\ ;\ \bot} \qquad \dfrac{(8)\ \tau_{11} \sim_\Sigma \tau_2 \qquad (7)\ \Gamma \mid \Sigma \vdash\ v^\circ\ ::\ \tau_2\ ;\ \bot}{(6)\ \Gamma \mid \Sigma \vdash_{\mathrm{T}} \underline{\mathrm{pure}\ \sigma}\ ::\ \mathit{Pure}\ \sigma}}{(\underline{1})\ \Gamma \mid \Sigma \vdash\ \mathit{suspend}\ \ \underline{\mathrm{pure}\ \sigma}\ \ (\lambda(x : \tau_{11}).\, t_{12})\ \ v^\circ\ ::\ \tau_{12}\ ;\ \bot}$$

$$(\underline{2})\ \mathrm{H}\ ;\ \mathit{suspend}\ \ \underline{\mathrm{pure}\ \sigma}\ \ (\lambda(x : \tau_{11}).\, t_{12})\ \ v^\circ\ \longrightarrow\ \mathrm{H}\ ;\ t[v^\circ/x]$$

$(\underline{3}, \underline{4})$	$\Sigma \models \mathrm{H},\ \Sigma \vdash \mathrm{H}$	(assume)
(10..12)	$\Gamma \mid \Sigma \vdash\ t_{12}[v^\circ/x]\ ::\ \tau'_{12}\ ;\ \sigma'$, $\tau'_{12} \sim_\Sigma \tau_{12},\ \sigma' \sqsubseteq_\Sigma \sigma$	(Sub. Value/Value 9 7 8)
(13)	$\sigma \sqsubseteq_\Sigma \bot$	(SubPurify 6)
(14)	$\sigma' \sqsubseteq_\Sigma \bot$	(12 13)