Technical Report

UCAM-CL-TR-835
ISSN 1476-2986

Number 835

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Call-by-need supercompilation

Maximilian C. Bolingbroke

May 2013

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted April 2013 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Robinson College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Summary

This thesis shows how supercompilation, a powerful technique for transformation and analysis of functional programs, can be effectively applied to a call-by-need language. Our setting will be core calculi suitable for use as intermediate languages when compiling higher-order, lazy functional programming languages such as Haskell.

We describe a new formulation of supercompilation which is more closely connected to operational semantics than the standard presentation. As a result of this connection, we are able to exploit a standard Sestoft-style operational semantics to build a supercompiler which, for the first time, is able to supercompile a call-by-need language with unrestricted recursive **let** bindings. We give complete descriptions of all of the (surprisingly tricky) components of the resulting supercompiler, showing in detail how standard formulations of supercompilation have to be adapted for the call-by-need setting.

We show how the standard technique of generalisation can be extended to the call-by-need setting. We also describe a novel generalisation scheme which is simpler to implement than standard generalisation techniques, and describe a completely new form of generalisation which can be used when supercompiling a typed language to ameliorate the phenomenon of supercompilers overspecialising functions on their type arguments.

We also demonstrate a number of non-generalisation-based techniques that can be used to improve the quality of the code generated by the supercompiler. Firstly, we show how **let**-speculation can be used to ameliorate the effects of the work-duplication checks that are inherent to call-by-need supercompilation. Secondly, we demonstrate how the standard idea of “rollback” in supercompilation can be adapted to our presentation of the supercompilation algorithm.

We have implemented our supercompiler as an optimisation pass in the Glasgow Haskell Compiler. We perform a comprehensive evaluation of our implementation on a suite of standard call-by-need benchmarks. We improve the runtime of the benchmarks in our suite by a geometric mean of 42%, and reduce the amount of memory which the benchmarks allocate by a geometric mean of 34%.

Acknowledgements

First of all, I thank Alan Mycroft for giving me my first introduction to the area of compiler optimisation, and supporting my application for a Ph.D position which I could use to investigate the world of optimisation for myself.

Working with my academic supervisor, Simon Peyton Jones, has been both an honour and a pleasure. Our frequent meetings were a constant source of insight and inspiration for me. His advice has not only contributed to the technical content of this thesis, but has immeasurably improved the presentation of the ideas herein.

This work could not have been possible without the financial support of Microsoft Research's Ph.D studentship program.

I would also like to take the opportunity to thank those who have discussed supercompilation with me over the years, including Dominic Orchard, Robin Message and Boris Feigin of the Cambridge CPRG. Particularly deserving of mention are Neil Mitchell, Peter Jonsson and Ilya Klyuchnikov, whose expert comments were invaluable.

Thanks are also due to John Hughes, Jeremy Gibbons, Daniel Peebles and the anonymous reviewers of the papers that were incorporated into this thesis for their thought-provoking comments and suggestions.

Finally, I am eternally grateful to my wife, Wenqian, for her constant love and support.

Contents

List of Figures

Chapter 1

Introduction

When programming, it is often valuable to work at a high level of abstraction, or to compose together many related subsystems to produce the final program. These practices are particularly prevalent in functional programs, where abstraction over functional arguments, and the generous use of intermediate data structures to communicate between multiple pieces of code, is idiomatic and commonplace [Hughes, 1989].

The use of abstraction and composition can be a considerable boon to programmer productivity, but it can impose runtime costs because the additional abstraction often hides information that a compiler could otherwise have used when optimising the program during compilation. In the context of functional programming, this cost manifests itself in many ways:

- Calls to higher-order functional arguments involve indirect branches, which are typically less efficient than direct jumps on modern processors due to the difficulties they cause with branch prediction.

- Compiled functions are typically compiled at a certain *arity*, which is the number of arguments a caller may apply them to before their code is entered [Marlow and Peyton Jones, 2004]. When making a call to a higher-order argument, the caller has to explicitly inspect this arity at runtime in order to check that it has supplied enough arguments to actually enter the code of the functional argument. In contrast, calls to known functions can omit these checks because the arity is known at compile time.

- Allocation of intermediate data structures requires calls to the runtime system to allocate the necessary memory. Even if the allocation code (e.g. a simple bump allocator) is inlined into at the call site by the compiler, it will cause memory traffic, and the garbage collector will need to be invoked if no memory is immediately available. These branches to the garbage collector can impede low-level optimisation because program data in registers needs to be spilled to the stack around the branch in order that the garbage collector may traverse it.

- Memory usage from intermediate data structures typically will not increase peak memory usage (intermediate data structures are transient by definition), but they will cause the collector to be run more frequently.

It has long been observed that many of these runtime penalties can be ameliorated or eliminated if the compiler is able to perform appropriate program specialisation or partial evaluation steps. Within this broad framework, there are many approaches that can be used, but one particularly attractive option is so-called *supercompilation* [Turchin, 1986].

Supercompilation is a partial evaluation strategy which is uniquely simple to implement and use: unlike many common partial evaluation algorithms, it does not make use of an offline "binding time analysis" to decide where specialisation should occur and it does not require program annotation. It also has many characteristics desirable for a compiler optimisation pass: the algorithm is guaranteed to terminate regardless of the form of the input program and it is meaning-preserving—in particular, it does not transform non-terminating programs into terminating ones or vice-versa. Furthermore, research has shown supercompilation to be effective at eliminating abstraction-induced inefficiencies of the sort we are concerned with [Jonsson, 2011; Mitchell, 2008]. Supercompilation is capable of achieving results similar to popular optimising transformations such as deforestation [Wadler, 1988], function specialisation and constructor specialisation [Peyton Jones, 2007].

1.1 Supercompilation by example

At the highest level, the goal of a supercompiler is simply to perform evaluation of a program. (For our purposes, a "program" will be a term (expression) in a purely functional programming language.) Unlike a simple evaluator, however, a supercompiler can perform symbolic evaluation—i.e. evaluate open terms where not all information is available at the time of evaluation.

The best way to begin to understand supercompilation in more detail is by example. Let's begin with a simple example of how supercompilation can specialise functions to their higher-order arguments:

$$\begin{array}{l}
\mathbf{let}\ \mathit{inc} = \lambda x.\, x + 1 \\
\qquad \mathit{map} = \lambda f\ \mathit{xs}.\, \mathbf{case}\ \mathit{xs}\ \mathbf{of}\ \begin{array}[t]{lcl} [\,] & \to & [\,] \\ (y : \mathit{ys}) & \to & f\ y : \mathit{map}\ f\ \mathit{ys} \end{array} \\
\mathbf{in}\ \mathit{map}\ \mathit{inc}\ \mathit{zs}
\end{array}$$

A supercompiler evaluates open terms, so that reductions that would otherwise be done at runtime are performed at compile time. Consequently, the first step of the algorithm is to reduce the term as much as possible, following standard evaluation rules:

$$\begin{array}{l}
\mathbf{let}\ \mathit{inc} = \ldots; \mathit{map} = \ldots \\
\mathbf{in}\ \mathbf{case}\ \mathit{zs}\ \mathbf{of}\ \begin{array}[t]{lcl} [\,] & \to & [\,] \\ (y : \mathit{ys}) & \to & \mathit{inc}\ y : \mathit{map}\ \mathit{inc}\ \mathit{ys} \end{array}
\end{array}$$

At this point, we become stuck on the free variable *zs*. One of the most important decisions when designing a supercompiler is how to proceed in such a situation, and we will spend considerable time later explaining how this choice is made when we cover the *splitter* in Section 3.5. In this particular example, we continue by recursively supercompiling two subexpressions. We intend to later recombine the two subexpressions into an output term where the **case** *zs* remains in the output program, but where both branches of the case have been further optimised by supercompilation.

The first subexpression is just [ ]. Because this is already a value, supercompilation makes no progress: the result of supercompiling that term is therefore [ ].

The second subexpression is:

$$\begin{array}{l}
\mathbf{let}\ \mathit{inc} = \ldots; \mathit{map} = \ldots \\
\mathbf{in}\ \mathit{inc}\ y : \mathit{map}\ \mathit{inc}\ \mathit{ys}
\end{array}$$

Again, evaluation of this term is unable to make progress: the rules of call-by-need reduction do not evaluate within non-strict contexts such as the arguments of data constructors.

It is once again time to use the splitter to produce some subexpressions suitable for further supercompilation. This time, the first subexpression is:

$$\textbf{let}\ \mathit{inc} = \ldots\ \textbf{in}\ \mathit{inc}\ y$$

Again, we perform reduction, yielding the supercompiled term $y + 1$. The other subexpression, originating from splitting the $(y : \mathit{ys})$ **case** branch, is:

$$\begin{array}{l}\textbf{let}\ \mathit{inc} = \ldots; \mathit{map} = \ldots\\ \textbf{in}\ \mathit{map}\ \mathit{inc}\ \mathit{ys}\end{array}$$

This term is identical to the one we started with, except that it has the free variable *ys* rather than *zs*. If we continued inlining and β-reducing the *map* call, the supercompiler would not terminate. This is not what we do.

Instead, the supercompiler uses a *memo function*. It records all of the terms it has been asked to supercompile as it proceeds, so that it never supercompiles the same term twice. In concrete terms, it builds up a set of *promises*, each of which is an association between a term previously submitted for supercompilation, its free variables, and a unique, fresh name (typically written *h0*, *h1*, etc.). At this point in the supercompilation of our example, the promises will look something like this:

$$\begin{array}{ll}\mathit{h0}\ \mathit{zs} & \mapsto \textbf{let}\ \mathit{inc} = \ldots; \mathit{map} = \ldots\ \textbf{in}\ \mathit{map}\ \mathit{inc}\ \mathit{zs}\\ \mathit{h1} & \mapsto [\,]\\ \mathit{h2}\ y\ \mathit{ys} & \mapsto \textbf{let}\ \mathit{inc} = \ldots; \mathit{map} = \ldots\ \textbf{in}\ \mathit{inc}\ y : \mathit{map}\ \mathit{inc}\ \mathit{ys}\\ \mathit{h3}\ y & \mapsto \textbf{let}\ \mathit{inc} = \ldots\ \textbf{in}\ \mathit{inc}\ y\end{array}$$

We have presented the promises in a rather suggestive manner, as if the promises were a sequence of bindings. Indeed, the intention is that the final output of the supercompilation process will be not only an optimised expression, but one optimised binding for each promise $\mathit{h0}, \mathit{h1}, \ldots, \mathit{hn}$.

Because the term we are now being asked to supercompile is simply a renaming of the original term (with which we associated the name *h0*) we can immediately return *h0 ys* as the supercompiled version of the current term. Producing a *tieback* like this we can rely on the (not yet known) optimised form of the original term (rather than supercompiling afresh), while simultaneously sidestepping a possible source of non-termination.

Now, both of the recursive supercompilations requested in the process of supercompiling *h1* have been completed. We can now rebuild the optimised version of the *h2* term from the optimised subterms, which yields:

$$\mathit{h3}\ y : \mathit{h0}\ \mathit{ys}$$

Continuing this process of rebuilding an optimised version of the supercompiler input from the optimised subexpressions, we eventually obtain this final program:

$$\begin{array}{l}\textbf{let}\ \mathit{h0}\ \mathit{zs} = \textbf{case}\ \mathit{zs}\ \textbf{of}\ [\,] \to \mathit{h1}; (y : \mathit{ys}) \to \mathit{h2}\ y\ \mathit{ys}\\ \quad \mathit{h1} = [\,]\\ \quad \mathit{h2}\ y\ \mathit{ys} = \mathit{h3}\ y : \mathit{h0}\ \mathit{ys}\\ \quad \mathit{h3}\ y = y + 1\\ \textbf{in}\ \mathit{h0}\ \mathit{zs}\end{array}$$

A trivial post-pass can eliminate some of the unnecessary indirections to obtain a version of the original input expression, where *map* has been specialised on its functional argument:

```
let h0 zs = case zs of [ ] → [ ]; (y : ys) → (y + 1) : h0 ys
in h0 zs
```

Throughout the thesis, we will usually present the results of supercompilation in a simplified form similar to the above, rather than the exact output of the supercompiler. In particular, we will tend to inline saturated *h*-functions when doing so does not cause any work duplication issues, and either:

- Have exactly one use site, as with *h1*, *h2* and *h3* above
- Have a body that is no larger than the call to the *h*-function, as for *h1* in a term like **let** *h1 f x* = *f x* **in** (*h1 g y*, *h1 l z*)

The motivation for these transformations is for clarity of the output code. These are transformations that will be performed by any optimising compiler anyway, so applying them to the presented code should not give any misleading impressions of the degree of optimisation the supercompiler performs.

1.1.1 Supercompilation for optimisation

Because supercompilation performs symbolic evaluation, by making use of supercompilation as an optimisation pass in a compiler we tend to perform at compile time some evaluation that would otherwise have happened at run time. For example, consider

```
let map f xs = case xs of [ ]      → [ ]
                          (y : ys) → f y : map f xs
in map f (map g ys)
```

Compiling this program typically produces machine code that heap-allocates the list produced by the *map g ys* invocation. A supercompiler is able to symbolically evaluate the above program to the following:

```
let h f g xs = case xs of [ ]      → [ ]
                          (y : ys) → f (g y) : h f g ys
in h f g xs
```

Note that this program omits the allocation of the intermediate list, and thus the compiled program will typically run faster than the pre-supercompilation program.

In this example, supercompilation has performed an optimisation usually known as "deforestation". Specialised techniques [Gill et al., 1993; Coutts et al., 2007; Wadler, 1988] already exist to perform deforestation, but the attraction of the supercompilation approach is that deforestation is only a special case of supercompilation—a supercompiler uses a single symbolic evaluation algorithm to perform a large class of optimisations. We characterise the class of optimisations it is able to perform more precisely in Section E.5.

The use of supercompilation for optimisation will be the focus of this thesis.

Optimisation is not the only use of supercompilation. The ability to evaluate open terms is also useful when proving properties that will hold for every possible execution of the program. This approach has been used to verify the correctness of cache-coherency protocols [Klimov, 2010; Lisitsa and Nemytykh, 2005] and also of general programs [Lisitsa and Nemytykh, 2008].

1.2 Contributions

Previous work has focused on supercompilation in a call-by-name [Klyuchnikov, 2009; Sørensen and Glück, 1995] context. The focus of call-by-name appears to be because historically supercompilation has been more popular as a proof technique than as a program optimisation.

More recently, supercompilation has been extended to call-by-value [Jonsson and Nordlander, 2009], with an eye to taking advantage of it to optimise popular call-by-value languages such as Lisp [Steele Jr and Common, 1984] and the commercial product F# [Syme and Margetson, 2008]. However, existing work cannot be straightforwardly adapted to a call-by-need setting (as used by the popular functional programming language Haskell). The key difference between call-by-need and call-by-name is *work sharing*: the work done reducing a term when it is used at one reference site is shared by all other reference sites, should they need to reduce the term themselves. The need to preserve work sharing has an impact on all parts of the supercompiler, and previous work on supercompilation in the context of Haskell [Mitchell and Runciman, 2008] has only dealt with a restricted form of programs not involving full recursive **let** expressions, as these create particular difficulties for the preservation of work sharing.

This thesis focuses on the issues surrounding supercompilation in a call-by-need setting: specifically in the setting of the compiler intermediate language of the Glasgow Haskell Compiler (GHC) [Peyton Jones et al., 1992]. Concretely, our contributions are:

1. We describe a way of structuring a supercompiler which is more modular than previous work (Chapter 3). A modular implementation is desirable not only from a software engineering perspective (in that it separates concerns) but also because clearly defined module boundaries make it straightforward to swap in alternative implementations of those modules for the purposes of research and experimentation.

   Our modularisation contains at its heart a call-by-need evaluator that is a straightforward implementation of the operational semantics of the language (in GHC, by contrast, evaluation and optimisation are intimately interwoven), which helps to make clear the relationship between supercompilation and the operational semantics of the language being supercompiled.

2. The choice of call-by-need semantics has a subtle but pervasive effect on the implementation of the supercompiler. We exploit the fact that our modular implementation is explicitly based on an operational semantics to define the first supercompilation algorithm for a call-by-need language that includes unrestricted recursive **let** expressions (Chapter 3, Chapter 4 and Chapter 5). Our central use of operational semantics is important because a Sestoft [Sestoft, 1997] or Launchbury-style [Launchbury, 1993] semantics are the theoretical tools by which we can understand the phenomenon of work-sharing.

   As a result of our use of operational semantics, our supercompiler is defined to optimise *abstract machine states* rather than expressions. As a first-order approximation, these states can be seen as a "zipperised" [Huet, 1997] version of an expression. In fact, the presence of Sestoft-style *update frames* in our states both breaks the strict analogy with zippers and will also provide precisely the information we need to correctly deal with call-by-need evaluation.

   A supercompiler which takes into account the full implications of call-by-need has two principal advantages:

- Our supercompiler can deforest value recursive programs. For example, the following term:

  $$\begin{array}{l}\textbf{let}\ \mathit{ones} = 1 : \mathit{ones}; \mathit{map} = \ldots\\ \textbf{in}\ \mathit{map}\ (\lambda x.\, x + 1)\ \mathit{ones}\end{array}$$

  will be optimised into the direct-style definition:

  $$\textbf{let}\ \mathit{xs} = 2 : \mathit{xs}\ \textbf{in}\ \mathit{xs}$$

  Previous supercompilers for lazy languages have dealt only with non-recursive **let** bindings, and so have been unable to perform this optimisation. Klyuchnikov's call-by-name supercompiler HOSC [Klyuchnikov, 2009] is able to deforest this example, but at the cost of sometimes duplicating work—something that we are careful to avoid.

- Because recursion is not special, unlike previous work, we do not need to give the program top-level special status, or λ-lift the input program. Not only is avoiding λ-lifting convenient, it is particularly useful to avoid it in a call-by-need setting because λ-lifting must be careful to preserve work sharing. For example, when given the input program:

  $$f\ x = \textbf{let}\ g = \lambda y.\, \ldots g \ldots \textbf{in} \ldots$$

  In order to avoid pessimising the program by λ-abstracting the lifted λ over the recursive use of g, we should lift it to:

  $$\begin{array}{l}g_{\mathit{lift}}\ x\ y = \ldots (g_{\mathit{lift}}\ x) \ldots\\ f\ x = \textbf{let}\ g = g_{\mathit{lift}}\ x\ \textbf{in} \ldots\end{array}$$

  And not to this more obvious program:

  $$\begin{array}{l}g_{\mathit{lift}}\ x\ g\ y = \ldots g \ldots\\ f\ x = \textbf{let}\ g = g_{\mathit{lift}}\ x\ g\ \textbf{in} \ldots\end{array}$$

  Conversely, if g is bound to a non-value, as in the following:

  $$f\ x = \textbf{let}\ g = h\ (\lambda y.\, \ldots g \ldots)\ \textbf{in} \ldots$$

  In order to preserve work sharing we *have* to lift this g using the second formulation and not the first one:

  $$\begin{array}{l}g_{\mathit{lift}}\ x\ g\ y = \ldots g \ldots\\ f\ x = \textbf{let}\ g = h\ (g_{\mathit{lift}}\ x\ g)\ \textbf{in} \ldots\end{array}$$

  So in a call-by-need setting with full recursive **let**, λ-lifting can potentially turn **let**-bound variables into λ-bound variables. All other things being equal, this can lead to compiler optimisations being significantly pessimised. By working with non-λ-lifted terms directly we avoid this need to pessimise the input program before we optimise it with the supercompiler.

3. We also show how the standard technique of *generalisation* can be extended to a call-by-need setting (Section 6.2), describing not only a call-by-need version of the most specific generalisation (Section 6.2.3) but also a novel generalisation technique which is easier to implement than most specific generalisation (Section 6.2.2).

It is critical that a supercompiler performs generalisation because without it only simple programs without accumulating arguments can be deforested and specialised. For example, a supercompiler without generalisation a *map f.map g* composition can both be deforested and specialised on the functions f and g, but a call *foldl* $(+)\ 0\ z$ will generate a residual call to an unspecialised *foldl* loop.

4. We describe how **let**-speculation can be used to make the work-duplication checks inherent to a call-by-need supercompiler less conservative so that, for example, "cheap" computations such as partial applications can be propagated to their use sites (Section 6.3).

5. We show how the known technique of rollback can be adapted to our formulation of supercompilation by using an exception-throwing mechanism, which we implement with a continuation-passing monad (Section 6.1.2).

6. Previous work has paid little attention to the issues arising from supercompiling a *typed* intermediate language. This is a problem because compiler intermediate languages commonly carry type information for reasons of sanity-checking and code generation; for example, the representation of a value at the machine-level may depend on the user-level type of that value. This is a source of unreliability for some supercompilers: e.g. the supercompiler for the call-by-value Timber language [Black et al., 2002] has an intermediate language that obeys the Hindley-Milner typing discipline [Damas and Milner, 1982], and supercompilation is performed by discarding the type information, supercompiling, and then performing type inference on the result. Unfortunately, in some cases supercompilation can transform programs typeable in Hindley-Milner to ones which are untypeable, so the last step can fail. For example, the input program

$$\begin{array}{l}\textbf{let}\ \mathit{choose} = \textbf{if}\ \mathit{fib}\ n > 100\ \textbf{then} \lambda x\ y.\, x\ \textbf{else} \lambda x\ y.\, y \\ \quad a = \textbf{if}\ \mathit{choose}\ \mathit{True}\ \mathit{False}\ \textbf{then}\ \mathit{choose}\ 1\ 2\ \textbf{else}\ \mathit{choose}\ 3\ 4 \\ \quad b = \textbf{if}\ \mathit{choose}\ \mathit{True}\ \mathit{False}\ \textbf{then}\ \mathit{choose}\ 5\ 6\ \textbf{else}\ \mathit{choose}\ 7\ 8 \\ \textbf{in}\ (a, b)\end{array}$$

might be transformed by a call-by-need supercompiler (that is careful to preserve work-sharing) into

$$\begin{array}{l}\textbf{let}\ \mathit{h0}\ n = \textbf{let}\ \mathit{choose} = \mathit{h1}\ n \\ \qquad\qquad\quad \textbf{in}\ (\mathit{h2}\ \mathit{choose}, \mathit{h3}\ \mathit{choose}) \\ \quad \mathit{h1}\ n = \textbf{if}\ \mathit{fib}\ n > 100\ \textbf{then} \lambda x\ y.\, x\ \textbf{else} \lambda x\ y.\, y \\ \quad \mathit{h2}\ \mathit{choose} = \textbf{if}\ \mathit{choose}\ \mathit{True}\ \mathit{False}\ \textbf{then}\ \mathit{choose}\ 1\ 2\ \textbf{else}\ \mathit{choose}\ 3\ 4 \\ \quad \mathit{h3}\ \mathit{choose} = \textbf{if}\ \mathit{choose}\ \mathit{True}\ \mathit{False}\ \textbf{then}\ \mathit{choose}\ 5\ 6\ \textbf{else}\ \mathit{choose}\ 7\ 8 \\ \textbf{in}\ \mathit{h0}\ n\end{array}$$

While the input is typeable thanks to **let**-generalisation, the output is untypeable because *choose* has been transformed from a **let**-bound into a λ-bound variable.

This thesis solves these issues by showing how to supercompile a typed language based on System Fω that contains explicit type abstraction and application, which we call Core (Chapter 2).

7. Supercompiling a typed intermediate language is mostly straightforward, but it leads to the phenomenon of *type overspecialisation*: the output of supercompilation can include in the output multiple specialisations that differ only in their type

information. It is necessary for the supercompiler to specialise on type information to at least some degree in order to ensure that the output is type-correct, but from a code-size perspective, it seems unfortunate to specialise *only* on type information. We describe a novel technique by which the supercompiler can avoid specialising on type information which is provably unnecessary (Section 6.2.6).

8. We evaluate an implementation of our supercompiler as an optimisation pass in GHC. We provide experimental results for our supercompiler, including an evaluation of the effects on benchmarks drawn from the standard "nofib" benchmark suite [Partain, 1993] (Chapter 7). As part of this work, we look at the effects that our new supercompilation techniques such as **let**-speculation have on the quality of the generated code.

In previously published work [Bolingbroke et al., 2011] we showed how the mathematical theory of *well-quasi-orders* could be realised as a Haskell combinator library, which can be used to concisely, efficiently and safely implement the termination-checking module of our supercompiler. This work can be seen as an alternative implementation of the termination module of our modular supercompiler design. As this work was done in the context of the Ph.D we reproduce it in Appendix D, but it is not necessary to read it in order to understand any other part of the thesis.

Appendix E discusses some of the practical considerations involved with supercompilation, which have seen scant attention in the published literature. To help rectify this, we share some of our experience with building an implementation of a supercomplier as part of an existing widely-used compiler: in particular we consider the issues arising from separate compilation (Section E.1), discuss the more unusual features of GHC's intermediate language (Section E.3), and contrast supercompilation with various other optimisations implemented by GHC (Section E.5).

Chapter 2

Preliminaries

In this chapter we introduce the grammar, type system and operational semantics of the language that we will supercompile. The language is chosen to be representative of a compiler intermediate language in a compiler for a call-by-need language such as Haskell, and is close to the core language of GHC, in which we have implemented our supercompiler. A detailed discussion of the differences between GHC's language and this one—and the complications which arise from the differences—is provided in Section E.3.

2.1 The Core language

The Core language, whose grammar is given in Figure 2.1, is based on System Fω, and as such is an explicitly typed, higher-order functional language. In Core, we make the following extensions to standard System Fω:

- Every subterm is associated with a *tag*, where each tag t is a natural number. A *tagged term* (d in Figure 2.1) is simply a term e with a tag t, written e^t. Before supercompilation begins, the untagged input program is tagged, once and for all, with a fresh tag at every node, for later use by the supercompilation algorithm. We will write *Tag* for the set of tags thus used to annotate the original term.

- We introduce algebraic data types and **case** statements to deconstruct them. We insist that all data constructors occur saturated, and use the standard technique of wrapper functions to handle any partial applications.

- We add a built-in **let** for both recursive and non-recursive binding.

- The language is in A-normal form (ANF) [Flanagan et al., 1993]: all arguments at applications are variables. However, for clarity of presentation we will often write non-ANFed expressions in our examples.

In Section E.2 and Section E.3 we discuss how Core can be further extended to encompass other language features which are often used by compiler intermediate languages for functional languages.

As Core is based on System Fω, it has a type system incorporating polymorphism. The full type system is presented in Figure 2.2. Note that we leave implicit the standard well-formedness requirement that there be no clashing binders in the same binding group (e.g. the same **let**, case alternative etc).

We leave implicit the (obvious) definition of the functions *fvs* and *ftvs* which find the free term and type variables (respectively) of an expression.

Type Vars α, β **Term Vars** x, y, z **Type Constructors** $\mathbf{T} ::= \mathit{Int}, (\to), \ldots$

Kinds

$$\begin{array}{lcll} \kappa & ::= & \star & \text{Kind of term types} \\ & | & \kappa \to \kappa & \text{Kind of type constructors} \end{array}$$

Types

$$\begin{array}{lcll} \tau, \upsilon & ::= & \alpha & \text{Type variable} \\ & | & \mathbf{T} & \text{Type constructors} \\ & | & \tau\ \tau & \text{Type application} \\ & | & \forall \alpha{:}\kappa.\, \tau & \text{Parametric polymorphism} \end{array}$$

Terms

$$\begin{array}{lcll} d & ::= & e^t & \text{Tagged term} \\ e & ::= & x & \text{Term variable} \\ & | & \lambda x{:}\tau.\, d & \text{Term abstraction} \\ & | & d\ x & \text{Term application} \\ & | & \mathbf{C}\ \overline{\tau}\ \overline{x} & \text{Algebraic data} \\ & | & \mathbf{case}\ d\ \mathbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau} \to d} & \text{Algebraic scrutinisation} \\ & | & \mathbf{let}\ \overline{x{:}\tau = d}\ \mathbf{in}\ d & \text{Recursive term binding} \\ & | & \Lambda \alpha{:}\kappa.\, d & \text{Type abstraction} \\ & | & d\ \tau & \text{Type application} \end{array}$$

Heaps

$$\begin{array}{lcll} H & ::= & \overline{h} & \text{Heap} \\ h & ::= & x{:}\tau \mapsto d & \text{Heap binding} \end{array}$$

$$\begin{array}{llcl} \textbf{States} & \mathcal{S} & ::= & \langle H \,|\, d \,|\, K \rangle_{\Sigma|\Gamma} \\ \textbf{Kinding Contexts} & \Sigma & ::= & \overline{\mathbf{T}{:}\kappa}, \overline{\alpha{:}\kappa} \\ \textbf{Typing Contexts} & \Gamma & ::= & \overline{\mathbf{C}{:}\tau}, \overline{x{:}\tau} \end{array}$$

Stacks

$$\begin{array}{lcll} K & ::= & \overline{\kappa^t} & \text{Stack} \\ \kappa & ::= & \mathbf{update}\ x{:}\tau & \text{Update frame} \\ & | & \bullet\ x & \text{Supply argument to function value} \\ & | & \bullet\ \tau & \text{Instantiate value} \\ & | & \mathbf{case} \bullet \mathbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau} \to d} & \text{Scrutinise value} \end{array}$$

Values

$$\begin{array}{lcll} u & ::= & v^t & \text{Tagged value} \\ v & ::= & \lambda x{:}\tau.\, d & \text{Function values} \\ & | & \Lambda \alpha{:}\kappa.\, d & \text{Type-abstracted values} \\ & | & \mathbf{C}\ \overline{\tau}\ \overline{x} & \text{Algebraic data values} \end{array}$$

Figure 2.1: Syntax of Core and its operational semantics

2.2 Operational semantics of Core

The dynamic semantics of Core are given by a small-step operational semantics which implements lazy evaluation Sestoft-style [Sestoft, 1997]. The operational semantics, presented in Figure 2.3, is concerned with reducing *machine states* rather than terms d. A state $\mathcal{S} \equiv \langle H \,|\, d \,|\, K \rangle_{\Sigma|\Gamma}$ has the following components (the full grammar was presented in

$$\boxed{\Sigma|\Gamma \vdash e \ : \ \tau}$$

$$\frac{x{:}\tau \in \Gamma}{\Sigma|\Gamma \vdash x \ : \ \tau}\ \textsc{Var}$$

$$\frac{\Sigma|\Gamma, x{:}\upsilon \vdash e \ : \ \tau \quad \Sigma \vdash^{\kappa} \upsilon \ : \ \star}{\Sigma|\Gamma \vdash \lambda x{:}\upsilon.\, e^{t} \ : \ \upsilon \to \tau}\ \textsc{Lam} \qquad \frac{\Sigma|\Gamma \vdash e \ : \ \upsilon \to \tau \quad x{:}\upsilon \in \Gamma}{\Sigma|\Gamma \vdash e^{t}\ x \ : \ \tau}\ \textsc{App}$$

$$\frac{\begin{array}{c}\mathbf{C}{:}\forall \overline{\alpha_\forall{:}\kappa_\forall}^{i}, \overline{\alpha_\exists{:}\kappa_\exists}^{j}.\, \overline{\tau_{\mathbf{C}}}^{k} \to \mathbf{T}\ \overline{\alpha_\forall} \in \Gamma \\ \overline{\Sigma \vdash^{\kappa} \upsilon_\forall \ : \ \kappa_\forall} \quad \overline{\Sigma \vdash^{\kappa} \upsilon_\exists \ : \ \kappa_\exists} \quad \overline{x{:}\tau_{\mathbf{C}}[\overline{\upsilon_\forall/\alpha_\forall}, \overline{\upsilon_\exists/\alpha_\exists}] \in \Gamma}\end{array}}{\Sigma|\Gamma \vdash \mathbf{C}\ \overline{\upsilon_\forall}^{i}, \overline{\upsilon_\exists}^{j}\ \overline{x}^{k} \ : \ \mathbf{T}\ \overline{\upsilon_\forall}^{i}}\ \textsc{Data}$$

$$\frac{\begin{array}{cc}\Sigma|\Gamma \vdash e \ : \ \mathbf{T}\ \overline{\upsilon_\forall}^{i} & \overline{\mathit{ftvs}\,(\tau) \cap \overline{\alpha_\exists} = \varnothing} \\ \overline{\mathbf{C}{:}\forall \overline{\alpha_\forall{:}\kappa_\forall}^{i}, \overline{\alpha_\exists{:}\kappa_\exists}^{j}.\, \overline{\tau_{\mathbf{C}}}^{k} \to \mathbf{T}\ \overline{\alpha_\forall}^{i} \in \Gamma} & \overline{\Sigma, \alpha_\exists{:}\kappa_\exists|\Gamma, x{:}\tau_{\mathbf{C}}[\upsilon_\forall/\alpha_\forall] \vdash e_{\mathbf{C}} \ : \ \tau}\end{array}}{\Sigma|\Gamma \vdash \textbf{case}\ e^{t}\ \textbf{of}\ \overline{\mathbf{C}\ \overline{\alpha_\exists{:}\kappa_\exists}^{j}\ \overline{x{:}\tau_{\mathbf{C}}[\upsilon_\forall/\alpha_\forall]}^{k} \ \to \ e_{\mathbf{C}}^{t_{\mathbf{C}}}} \ : \ \tau}\ \textsc{Case}$$

$$\frac{\overline{\Sigma|\Gamma, \overline{x{:}\upsilon} \vdash e_x \ : \ \upsilon} \quad \Sigma|\Gamma, \overline{x{:}\upsilon} \vdash e \ : \ \tau}{\Sigma|\Gamma \vdash \textbf{let}\ \overline{x{:}\upsilon \ = \ e_x^{t_x}}\ \textbf{in}\ e^{t} \ : \ \tau}\ \textsc{LetRec}$$

$$\frac{\Sigma, \alpha{:}\kappa|\Gamma \vdash e \ : \ \tau}{\Sigma|\Gamma \vdash \Lambda\alpha{:}\kappa.\, e^{t} \ : \ \forall\alpha{:}\kappa.\,\tau}\ \textsc{TyLam} \qquad \frac{\Sigma|\Gamma \vdash e \ : \ \forall\alpha{:}\kappa.\,\tau \quad \Sigma \vdash^{\kappa} \upsilon \ : \ \kappa}{\Sigma|\Gamma \vdash e^{t}\ \upsilon \ : \ \tau[\upsilon/\alpha]}\ \textsc{TyApp}$$

Figure 2.2: Type system of Core

Figure 2.1):

- The *heap*, H, is a finite mapping from variable names to the term to which that variable is bound.
- The *focus term*, d, is the focus of evaluation.
- The *stack*, K, describes the evaluation context of the focus term.
- The *kinding context*, Σ, records the kinds of free type variables.
- The *typing context*, Γ, records the types of free kind variables.

The inclusion of kinding and typing contexts in our states is unconventional, but as supercompilers are concerned with evaluating *open* terms it is often crucial to have this typing information available. We often omit the typing and kinding contexts from a state to avoid clutter, writing $\langle H \,|\, d \,|\, K \rangle$ instead. For example, the typing and kinding information is irrelevant to the operational semantics, so our reduction rules leave implicit the fact that this information is preserved by reduction.

Throughout the thesis, in order to avoid cluttering our definitions with name-management machinery, we will often implicitly assume that α-conversion has taken place. Our operational semantics is no exception to this convention. For example, the rule for performing β-reduction assumes we have implicitly α-converted the λ-bound variable to match the applied variable. Similarly, we assume in the rule for dealing with **let** that the group of **let**-bound variables has been α-renamed so none of the bound variables clash with any of those already present in the heap.

$$\boxed{\langle H \,|\, d \,|\, K \rangle \rightsquigarrow \langle H \,|\, d \,|\, K \rangle}$$

$$\begin{array}{ll}
\textsc{var} & \langle H, x{:}\tau \mapsto d \,|\, x^{t} \,|\, K \rangle \rightsquigarrow \langle H \,|\, d \,|\, \mathbf{update}\ x{:}\tau^{t}, K \rangle \quad (d \not\equiv v^{t_v}) \\
\textsc{updatev} & \langle H \,|\, u \,|\, \mathbf{update}\ x{:}\tau^{t_x}, K \rangle \rightsquigarrow \langle H, x{:}\tau \mapsto u \,|\, x^{t_x} \,|\, K \rangle \\
\textsc{update} & \langle H[x{:}\tau \mapsto u] \,|\, x^{t_x} \,|\, \mathbf{update}\ y{:}\tau^{t_y}, K \rangle \rightsquigarrow \langle H, y{:}\tau \mapsto x^{t_y} \,|\, x^{t_x} \,|\, K \rangle \\
\\
\textsc{app} & \left\langle H \,\middle|\, (d\ x)^{t} \,\middle|\, K \right\rangle \rightsquigarrow \langle H \,|\, d \,|\, \bullet\ x^{t}, K \rangle \\
\textsc{betav} & \left\langle H \,\middle|\, (\lambda x{:}\tau.\, d)^{t_\lambda} \,\middle|\, \bullet\ x^{t_@}, K \right\rangle \rightsquigarrow \langle H \,|\, d \,|\, K \rangle \\
\textsc{beta} & \left\langle H[f{:}\upsilon \mapsto (\lambda x{:}\tau.\, d)^{t_\lambda}] \,\middle|\, f^{t_f} \,\middle|\, \bullet\ x^{t_@}, K \right\rangle \rightsquigarrow \langle H \,|\, d \,|\, K \rangle \\
\\
\textsc{tyapp} & \left\langle H \,\middle|\, (d\ \tau)^{t} \,\middle|\, K \right\rangle \rightsquigarrow \langle H \,|\, d \,|\, \bullet\ \tau^{t}, K \rangle \\
\textsc{tybetav} & \left\langle H \,\middle|\, (\Lambda \alpha{:}\kappa.\, d)^{t_\Lambda} \,\middle|\, \bullet\ \tau^{t_@}, K \right\rangle \rightsquigarrow \langle H \,|\, d[\tau/\alpha] \,|\, K \rangle \\
\textsc{tybeta} & \left\langle H[f{:}\tau \mapsto (\Lambda \alpha{:}\kappa.\, d)^{t_\Lambda}] \,\middle|\, f^{t_f} \,\middle|\, \bullet\ \tau^{t_@}, K \right\rangle \rightsquigarrow \langle H \,|\, d[\tau/\alpha] \,|\, K \rangle \\
\textsc{case} & \left\langle H \,\middle|\, \left(\mathbf{case}\ d\ \mathbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau} \to d_{\mathbf{C}}}\right)^{t} \,\middle|\, K \right\rangle \rightsquigarrow \left\langle H \,\middle|\, d \,\middle|\, \mathbf{case} \bullet \mathbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau} \to d_{\mathbf{C}}}^{t}, K \right\rangle \\
\textsc{datav} & \left\langle H \,\middle|\, \left(\mathbf{C}\ \overline{\tau_\forall}^{i}, \overline{\tau_\exists}^{j}\ \overline{x}^{k}\right)^{t_{\mathbf{C}}} \,\middle|\, \mathbf{case} \bullet \mathbf{of}\ \left\{\ldots \mathbf{C}\ \overline{\alpha_\exists{:}\kappa_\exists}^{j}\ \overline{x{:}\upsilon}^{k} \to d \ldots\right\}^{t}, K \right\rangle \\
 & \qquad \rightsquigarrow \left\langle H \,\middle|\, d[\overline{\tau_\exists/\alpha_\exists}] \,\middle|\, K \right\rangle \\
\textsc{data} & \left\langle H[y{:}\mathbf{T}\ \overline{\tau_\forall} \mapsto \left(\mathbf{C}\ \overline{\tau_\forall}^{i}, \overline{\tau_\exists}^{j}\ \overline{x}^{k}\right)^{t_{\mathbf{C}}}] \,\middle|\, y^{t_y} \,\middle|\, \mathbf{case} \bullet \mathbf{of}\ \left\{\ldots \mathbf{C}\ \overline{\alpha_\exists{:}\kappa_\exists}^{j}\ \overline{x{:}\upsilon}^{k} \to d \ldots\right\}^{t}, K \right\rangle \\
 & \qquad \rightsquigarrow \left\langle H \,\middle|\, d[\overline{\tau_\exists/\alpha_\exists}] \,\middle|\, K \right\rangle \\
\textsc{letrec} & \left\langle H \,\middle|\, \left(\mathbf{let}\ \overline{x{:}\tau\ =\ d_x}\ \mathbf{in}\ d\right)^{t} \,\middle|\, K \right\rangle \rightsquigarrow \langle H, \overline{x{:}\tau \mapsto d_x} \,|\, d \,|\, K \rangle
\end{array}$$

Figure 2.3: Operational semantics of Core

The type-correctness of a state $\vdash \langle H \,|\, d \,|\, K \rangle_{\Sigma|\Gamma}\ :\ \tau$ is defined by the type-correctness judgement of its "rebuilding" into a term $\Sigma|\Gamma \vdash \langle\!| H \,|\, d \,|\, K |\!\rangle\ :\ \tau$, where rebuilding is defined as follows:

$$\begin{array}{rcl}
\left\langle\!\middle|\, \overline{x{:}\tau \mapsto d_x} \,\middle|\, d \,\middle|\, \epsilon \,\middle|\!\right\rangle & = & \mathbf{let}\ \overline{x{:}\tau = d_x}\ \mathbf{in}\ d \\
\langle\!| H \,|\, d \,|\, \mathbf{update}\ y{:}\tau, K |\!\rangle & = & \langle\!| H, y{:}\tau \mapsto d \,|\, y \,|\, K |\!\rangle \\
\langle\!| H \,|\, d \,|\, \bullet\ x, K |\!\rangle & = & \langle\!| H \,|\, d\ x \,|\, K |\!\rangle \\
\left\langle\!\middle|\, H \,\middle|\, d \,\middle|\, \mathbf{case} \bullet \mathbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau}\ \to\ d_{\mathbf{C}}}, K \,\middle|\!\right\rangle & = & \left\langle\!\middle|\, H \,\middle|\, \mathbf{case}\ d\ \mathbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau}\ \to\ d_{\mathbf{C}}} \,\middle|\, K \,\middle|\!\right\rangle \\
\langle\!| H \,|\, d \,|\, \bullet\ \tau, K |\!\rangle & = & \langle\!| H \,|\, d\ \tau \,|\, K |\!\rangle
\end{array}$$

Sometimes we will want to refer to the ith stack frame in a stack K, which we will write as $K[i]$. The frame at index 0 is the first frame in K, i.e. the one closest to the focus of evaluation. When we wish to make stack frame indexes explicit, we will write the stack in the form $\mathbf{0}{:}\kappa, \mathbf{1}{:}\kappa, \ldots$.

Our operational semantics, while not an entirely standard call-by-need operational semantics (as we discuss in Section 2.2.2 and Section 2.2.3), still enjoys all the properties you would expect: in particular, it is deterministic, type-preserving, and succeeds in reducing an initial state to a final "value" state exactly as often as a standard call-by-need operational semantics would. Value states are those of the form $\langle H \,|\, d \,|\, \epsilon \rangle$ such that $\mathit{deref}(H, d)$ is a value, where deref is defined by:

$$\mathit{deref}(H, e^{t}) = \begin{cases} d & e \equiv x \wedge x \mapsto d \in H \\ e^{t} & \text{otherwise} \end{cases}$$

2.2.1 Update frames

The operational semantics deals with the call-by-need memoisation of evaluation by using Sestoft-style *update frames*. When a heap binding $x \mapsto e$ is demanded by a variable x coming into the focus of the evaluator, e may not yet be a value. To ensure that we only reduce any given heap-bound e to a value at most once, the evaluator's VAR rule pushes an update frame **update** x on the stack, before beginning the evaluation of e . After e has been reduced to a value, v, the update frame will be popped from the stack, which is the cue for the evaluator's rule UPDATEV or UPDATE to update the heap with a binding $x \mapsto v$. Now, subsequent uses of x in the course of evaluation will be able to reuse that value directly, without reducing e again.

As an example of how update frames work, consider this reduction sequence:

$$\begin{array}{ll}
\multicolumn{2}{l}{\langle x \mapsto (\lambda z.\, \mathit{True})\; () \,|\, \mathbf{case}\; x\; \mathbf{of}\; \mathit{True} \to \mathbf{case}\; x\; \mathbf{of}\; \mathit{True} \to \mathit{False} \,|\, \epsilon \rangle} \\
\overset{\text{CASE}}{\rightsquigarrow} & \langle x \mapsto (\lambda z.\, \mathit{True})\; () \,|\, x \,|\, \mathbf{case} \bullet \mathbf{of}\; \mathit{True} \to \mathbf{case}\; x\; \mathbf{of}\; \mathit{True} \to \mathit{False} \rangle \\
\overset{\text{VAR}}{\rightsquigarrow} & \langle \epsilon \,|\, (\lambda z.\, \mathit{True})\; () \,|\, \mathbf{update}\; x, \mathbf{case} \bullet \mathbf{of}\; \mathit{True} \to \mathbf{case}\; x\; \mathbf{of}\; \mathit{True} \to \mathit{False} \rangle \\
\overset{\text{APP}}{\rightsquigarrow} & \langle \epsilon \,|\, \lambda z.\, \mathit{True} \,|\, \bullet\; (), \mathbf{update}\; x, \mathbf{case} \bullet \mathbf{of}\; \mathit{True} \to \mathbf{case}\; x\; \mathbf{of}\; \mathit{True} \to \mathit{False} \rangle \\
\overset{\text{BETAV}}{\rightsquigarrow} & \langle \epsilon \,|\, \mathit{True} \,|\, \mathbf{update}\; x, \mathbf{case} \bullet \mathbf{of}\; \mathit{True} \to \mathbf{case}\; x\; \mathbf{of}\; \mathit{True} \to \mathit{False} \rangle \\
\overset{\text{UPDATEV}}{\rightsquigarrow} & \langle x \mapsto \mathit{True} \,|\, x \,|\, \mathbf{case} \bullet \mathbf{of}\; \mathit{True} \to \mathbf{case}\; x\; \mathbf{of}\; \mathit{True} \to \mathit{False} \rangle \\
\overset{\text{DATA}}{\rightsquigarrow} & \langle x \mapsto \mathit{True} \,|\, \mathbf{case}\; x\; \mathbf{of}\; \mathit{True} \to \mathit{False} \,|\, \epsilon \rangle \\
\overset{\text{CASE}}{\rightsquigarrow} & \langle x \mapsto \mathit{True} \,|\, x \,|\, \mathbf{case} \bullet \mathbf{of}\; \mathit{True} \to \mathit{False} \rangle \\
\overset{\text{DATA}}{\rightsquigarrow} & \langle x \mapsto \mathit{True} \,|\, \mathit{False} \,|\, \epsilon \rangle
\end{array}$$

Notice that once the initial binding associated with x has been reduced to the value *True*, the update frame updates the heap with the evaluated version of x. The later use of the same variable x can use this recorded value directly, without evaluating the β-reduction $(\lambda z.\, \mathit{True})\; ()$ again, demonstrating that call-by-need is operating as it should.

Because the corresponding heap binding is removed from the heap whenever an update frame is pushed, the update frame mechanism is what causes reduction to become stuck if you evaluate a term which forms a so-called "black hole":

$$\langle x \mapsto x \,|\, x \,|\, \epsilon \rangle \rightsquigarrow \langle \epsilon \,|\, x \,|\, \mathbf{update}\; x \rangle \not\rightsquigarrow$$

2.2.2 Normalisation

Our exact choice of operational semantics is motivated by the desire to define a useful *normalising* subset of the operational semantics: a subset of the rules which are guaranteed to terminate on an arbitrary (well formed) input state. As we will see later (Section 3.9), by making use of a powerful normaliser in the supercompiler, we will be able to reduce the number of termination checks we have to do, which will in turn have beneficial effects on supercompiler runtime and the amount of reduction we can safely perform at compile time.

A more conventional operational semantics would include a rule dealing with update frames such as the following:

$$\text{UPDATE-COPY}\quad \langle H \,|\, v \,|\, \mathbf{update}\; x{:}\tau, K \rangle \rightsquigarrow \langle H, x{:}\tau \mapsto v \,|\, v \,|\, K \rangle$$

Our semantics does not include a rule of this form, which is its principal area of divergence from a standard operational semantics for System Fω. Instead, the UPDATEV rule reduces the LHS to a state with x in the focus, so that the value is not copied.

To ensure that our reduction rules can further reduce such a state, all of the standard reduction rules that expect to have a value in the focus come in two versions: one version deals with the familiar case where there is a value in the focus, and one which is capable of "looking through" a single variable in the focus into a heap-bound value.[1] An example of such a pair is BETAV and BETA respectively, with the other pairs being UPDATEV/UPDATE, TYBETAV/TYBETA and DATAV/DATA.

Given that we have pairs of rules like this, we can define a normaliser as a term evaluator that applies every rule in Figure 2.3, except for the two β-reduction rules BETA and TYBETA (note that we *do* allow the normaliser to apply the rules BETAV and TYBETAV). The intuition behind this restriction is that these are the only two rules which actually cause code to be duplicated: every other rule reduces the size of the abstract machine state in some sense.

Definition 2.2.1 (Multi-step reduction). $\mathcal{S} \rightsquigarrow^n \mathcal{S}'$ is defined to mean that $\mathcal{S}$ reduces to the state $\mathcal{S}'$ via exactly n uses of the non-normalising rules BETA and TYBETA, and any number of uses of the normalising rules.

Definition 2.2.2 (Convergence). For a closed state $\mathcal{S} \equiv \langle H \,|\, e \,|\, K \rangle$,

$$\begin{aligned}
\langle H \,|\, e \,|\, K \rangle \Downarrow^n &\iff \exists H', e', u. \langle H \,|\, e \,|\, K \rangle \rightsquigarrow^n \langle H' \,|\, e' \,|\, \epsilon \rangle \wedge \mathit{deref}(H', e') = u \\
\mathcal{S} \Downarrow &\iff \exists n. \mathcal{S} \Downarrow^n \\
\mathcal{S} \Downarrow^{\leq n} &\iff \exists m. \mathcal{S} \Downarrow^m \wedge m \leq n
\end{aligned}$$

Theorem 2.2.1 (Normalisation). *The operational semantics of Figure 2.3 without* BETA *and* TYBETA *is normalising.*

Proof. See Appendix A. □

Theorem 2.2.2 (Normalised forms). *Normalised states are guaranteed to have one of the following forms:*

- *Values:* $\langle H \,|\, d \,|\, \epsilon \rangle$ *where* $\mathit{deref}(H, d) = u$ *for some* u
- *Term β-redexes:* $\langle H, f : \tau \to \upsilon \mapsto (\lambda x{:}\tau.\, d)^{t_0} \,|\, f^{t_1} \,|\, \bullet\ x^{t_2}, K \rangle$
- *Type β-redexes:* $\langle H, f : \forall \alpha{:}\kappa.\, \upsilon \mapsto (\Lambda \alpha{:}\kappa.\, d)^{t_0} \,|\, f^{t_1} \,|\, \bullet\ \tau^{t_2}, K \rangle$
- *Free-variable references:* $\langle H \,|\, x^t \,|\, K \rangle$ *where* $x \notin \mathit{bvs}(H)$ *(remember that states can be open)*

Often, when we wish to present a (normalised) state, we will not write down an actual state in the form $\langle H \,|\, d \,|\, K \rangle$, but will rather write down the term $\langle\!| H \,|\, d \,|\, K |\!\rangle$ it rebuilds to. Expressions and normalised states are unambiguously interchangeable in this way since $\langle \epsilon \,|\, \langle\!| H \,|\, d \,|\, K |\!\rangle \,|\, \epsilon \rangle \rightsquigarrow^0 \langle H \,|\, d \,|\, K \rangle$. Therefore, the normalised state we refer to when we write the expression e can be determined by exhaustively applying the normalising reduction rules to the initial state $\langle \epsilon \,|\, e \,|\, \epsilon \rangle$.

[1] This ability to look through variables to the values they are bound to in the heap is similar to what can be achieved by the use of *indirections* in an implementation of call-by-need on stock hardware such as the STG machine [Peyton Jones, 1992].

2.2.3 Tags

The other unusual feature of the operational semantics is that the states it reduces are *tagged*. Tags are used for two quite separate purposes in our supercompiler. Firstly, they are used as the basis of a test to ensure that the supercompiler terminates (Section 3.2), and secondly they are used to guide generalisation (Section 6.2).

Tags take the form of an integer which is associated with every term. The intention is that the initial program input to the supercompiler has a *distinct* integer assigned to every subterm.

The operational semantics of Figure 2.3 define how tags are propagated during evaluation. The unbreakable rule is that *no new tags are generated*; it is this invariant that guarantees termination of the overall supercompiler. This however leaves plenty of room for variations in how tags are propagated. For example, UPDATE could sensibly tag the indirection it creates with the update frame tag rather than that of the value.

In fact, absolutely any choice of tags may be made by the operational semantics as long as all the tags present in a term after reduction were present in the input: the rest of the supercompiler will work unmodified with no changes given any choice of tags. The exact choice can however influence both the termination test and the effectiveness of generalisation.

2.3 Improvement theory

In order to precisely define what the components of the supercompiler should do, we make use of *improvement theory* for lazy languages [Moran and Sands, 1999a,b]. Improvement theory is a tool for reasoning about the soundness of equational transformations (such as supercompilation). When applied to lazy languages, it not only allows us to reason about the termination behaviour of the programs we are transforming, but also their work-sharing properties. We will use the tools of improvement theory to not only prove the correctness of the supercompiler (Section 3.8), but also to define the properties that the various modules of our modular supercompiler will have, so it is essential to introduce the basics of the theory at this point.

Because our operational semantics differs slightly from a standard Sestoft-style abstract machine, we cannot reuse Sand's improvement theorems directly and must instead adapt them to our new setting, but thankfully the changes required will turn out to be straightforward. What follows will be a brief introduction to the lexicon of improvement theory: detailed development of the theory for our language will be relegated to Appendix C.

For the purposes of defining improvement theory, we will omit tags entirely (i.e. as if $d ::= e$). Tags are unimportant to the correctness of the proofs as they are computationally irrelevant.

To define improvement, we need to describe the notion of program contexts. To aid our proofs, we will work with so-called "generalised contexts" [Sands, 1998], defined as follows:

$$\begin{array}{rcl}
\mathbb{C}, \mathbb{D} & ::= & \xi \cdot \overline{x} \mid x \\
 & \mid & \lambda x{:}\tau.\, \mathbb{C} \mid \mathbb{C}\ x \\
 & \mid & \mathbf{C}\ \overline{\tau}\ \overline{x} \mid \mathbf{case}\ \mathbb{C}\ \mathbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau}\ \to\ \mathbb{D}} \\
 & \mid & \mathbf{let}\ \overline{x{:}\tau\ =\ \mathbb{C}}\ \mathbf{in}\ \mathbb{D} \\
 & \mid & \Lambda\alpha{:}\kappa.\, \mathbb{C} \mid \mathbb{C}\ \tau
\end{array}$$

The variables ξ are *meta-level variables*, and the syntax $\xi \cdot \overline{x}$ indicates a meta-level application of the meta-variable ξ to the syntax for the object-level variables $\overline{x}$. We denote meta-level abstractions as $(\overline{x}).\, e$, where the $\overline{x}$ bind variables that may be free in e. We define the operation $\mathbb{C}[(\overline{x}).\, e/\xi]$ of substitution for a meta-level variable ξ in a context $\mathbb{C}$ as a standard inductively-defined non-capturing substitution with the additional identification that $(\xi \cdot \overline{x})[(\overline{x}).\, e/\xi] = e$.

Each meta-variable has an associated arity, and we implicitly assume that that arity is respected both at application sites and by the meta-level abstractions that are substituted for those variables.

In addition to term contexts $\mathbb{C}$, we define $\mathbb{V}$, $\mathbb{H}$ and $\mathbb{K}$ to range over value contexts (i.e. algebraic data, or λ/Λ abstractions whose bodies may syntactically contain applications to meta-variables), heap contexts and stack contexts respectively.

These contexts generalise the standard definition of contexts, since given a traditional context $\hat{\mathbb{C}}$, we can construct a generalised context $\mathbb{C} = \hat{\mathbb{C}}[\hat{\xi} \cdot \overline{x}]$, where $\overline{x}$ is a vector of variables bound around the holes in $\hat{\mathbb{C}}$ and $\hat{\xi}$ is a fresh meta-level variable of suitable arity. The result of the hole-filling operation $\hat{\mathbb{C}}[e]$ is then equivalent to $\mathbb{C}[(\overline{x}).\, e/\hat{\xi}]$. Generalised contexts are more convenient to work with for our purposes than standard contexts because they are freely α-convertible, whereas standard contexts are not. For more discussion on this point, see Section C.1.

Throughout the thesis, we will work with generalised contexts that contain a single "hole" meta-variable ξ (though the meta-variable may occur multiple times within the same context), and will write $\mathbb{C}[(\overline{x}).\, e]$ to abbreviate $\mathbb{C}[(\overline{x}).\, e/\xi]$.

We are now in a position to define the fundamental notion of improvement:

Definition 2.3.1 (Improvement)**.** A term e is improved by e', written $e \mathrel{\underset{\sim}{\rhd}} e'$, iff for all $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle$ such that $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle\, [(\overline{x}).\, e]$ and $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle\, [(\overline{x}).\, e']$ are closed,

$$\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle\, [(\overline{x}).\, e] \Downarrow^{n} \implies \langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle\, [(\overline{x}).\, e'] \Downarrow^{\leq n}$$

Intuitively, a term e is improved by another term e' if they both have the same meaning, and e' is at least as fast as e. For our purposes, "speed" is defined not in terms of machine cycles, but rather by the number of non-normalising reduction steps that are required. One intuitive explanation for this definition is that normalising reductions can be seen as purely administrative reductions which a good compiler would do anyway, and so "don't really count" for the purposes of defining speed. Thus, we can say:

$$\begin{array}{rcl} \textbf{let}\, f = (\lambda x.\, x) \,\textbf{in}\, f \ \mathit{True} & \mathrel{\underset{\sim}{\rhd}} & \mathit{True} \\ \mathit{True} & \mathrel{\underset{\sim}{\rhd}} & \mathit{True} \end{array}$$

Note that the first of these two is a strict improvement, i.e. it is not an improvement in the other direction. We can also define the stronger notion of a cost equivalence that relates only those terms that are exactly as fast as each other:

Definition 2.3.2 (Cost equivalence)**.** A term e is cost-equivalent to e', written $e \mathrel{\underset{\sim}{\lhd\!\rhd}} e'$ iff $e \mathrel{\underset{\sim}{\rhd}} e'$ and $e' \mathrel{\underset{\sim}{\rhd}} e$.

For example, we can say that $(\lambda x.\, x)\ \mathit{True} \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathit{True}$. It is useful to define an operator ${}^{\checkmark}e$ (pronounced "tick e") which "slows down" reduction by exactly one use of a non-normalising rule:

$$\begin{array}{rcl} {}^{\checkmark}e & = & \textbf{let}\ \mathit{bot} :: C = \mathit{bot}; \mathit{dly} = \lambda(\mathit{bot} :: C).\, e \ \textbf{in}\ \mathit{dly}\ \mathit{bot} \\ & & \text{for fresh variables } \mathit{bot} \text{ and } \mathit{dly},\ C \text{ an arbitrary nullary type constructor} \end{array}$$

Theorem 2.3.1 (Tick)**.** *For all* e, e', H, H', K, K', n, *if* $\langle H \,|\, e \,|\, K \rangle \leadsto^{n} \langle H' \,|\, e' \,|\, K' \rangle$ *then* $\langle H \,|\, {}^{\checkmark}e \,|\, K \rangle \leadsto^{n+1} \langle \mathit{bot} \colon C \mapsto \mathit{bot}, \mathit{dly} \mapsto \lambda(\mathit{bot} :: C).\, e, H' \,|\, e' \,|\, K' \rangle$

Proof. By considering the reduction of a ticked term:

$$\begin{array}{llll}
\langle H \,|\, {}^{\checkmark}e \,|\, K\rangle & \rightsquigarrow^{0} & \langle bot{:}\, C \mapsto bot, H \,|\, \lambda(bot :: C).\, e \,|\, \textbf{update}\ dly, \bullet\ bot, K\rangle & \textsc{letrec}, \textsc{app}, \textsc{var} \\
 & \rightsquigarrow^{1} & \langle bot{:}\, C \mapsto bot, dly \mapsto \lambda(bot :: C).\, e, H \,|\, e \,|\, K\rangle & \textsc{beta} \\
 & \rightsquigarrow^{n} & \langle bot{:}\, C \mapsto bot, dly \mapsto \lambda(bot :: C).\, e, H' \,|\, e' \,|\, K'\rangle & \text{Lemma C.0.2}
\end{array}$$

□

The notions of improvement and cost-equivalence will come into play when we discuss matching in Section 3.4, and when we prove the supercompiler correct in Section 3.8.

Chapter 3

Supercompilation by evaluation

In this section, we cast supercompilation in a new light. The result is a supercompiler which is:

- Modular, with logically separate parts of the algorithm are broken into their own modules. This not only makes it simpler to understand the algorithm, but also means we enjoy other benefits of modularity, such as the fact that modules can be independently varied for experimental purposes.

- Based *directly* on the operational semantics of the language, making central the role of the underlying abstract machine in a way that Turchin's original work always intended with its notion of configurations [Turchin, 1986].

  Viewing supercompilation in this way is valuable, because it makes it easier to derive a supercompiler in a systematic way from the language, and to adapt it to new language features.

- Proven to terminate, and proven to preserve the meaning of the program it transforms.

Previous work tends to intermingle the elements of supercompilation in a more complex way, and often is defined with reference to a complicating mutable "process tree" data structure.

3.1 Core of the algorithm

We will formulate key parts of the supercompiler by means of an implementation in Haskell. This implementation will draw on a standard set of types, including not only those defined in Figure 3.1 and Figure 3.2 but also in the Haskell standard library [Marlow et al.].

The core of the supercompilation algorithm is *sc*, whose key property is this: for *any* history *hist* and state $\mathcal{S}$, $(sc\ hist\ \mathcal{S})$ returns a term with exactly the same meaning as $\mathcal{S}$, but which is implemented more efficiently[1]:

```
sc, sc' :: History → State → ScpM Term
sc hist = memo (sc' hist)
sc' hist 𝒮 = case terminate hist 𝒮 of
```

[1] It may be counterintuitive that no relationship is required between *hist* and $\mathcal{S}$ for this property to hold, but this is in fact the case.

```
    Stop           → split (sc hist) 𝒮
    Continue hist′ → split (sc hist′) (reduce 𝒮)
```

In the subsequent text we will pick apart this definition and explain all the auxiliary functions. The main data types and function signatures are summarised in Figure 3.2. In particular, the type of *sc* shows a unique feature of our approach to supercompilation, namely that *sc* takes as input the *machine states* that are the subject of the reduction rules, rather than *terms* as is more conventional. We review the importance of this choice in Section 3.9.

The main supercompiler *sc* is built from four, mostly independent, subsystems:

1. The *memoiser* (Section 3.4). The call to *memo* checks whether the input *State* is equal (modulo renaming) to a *State* we have already encountered. It is this memoisation that "ties the knot" so that a recursive function is not specialised forever. The memoiser records its state in the *ScpM* monad.

2. The *termination criterion* (Section 3.2). With luck the memoiser will spot an op-

```
type Subst = ...                -- Term var to term var, type var to type
substVar    :: Subst → Var → Var
substTyVar  :: Subst → TyVar → Type
substTerm   :: Subst → Term → Term
substType   :: Subst → Type → Type

type Tag = Int

type Heap = ...                 -- See H in Figure 2.1

type Stack = [StackFrame]       -- See K in Figure 2.1
data UStackFrame = ...          -- See κ in Figure 2.1
type StackFrame = (Tag, UStackFrame)

data UTerm = ...                -- See e in Figure 2.1
type Term = (Tag, UTerm)        -- See d in Figure 2.1
data Value = ...                -- See v in Figure 2.1

var        :: Var → Term
apps       :: Term → [Var] → Term
tyApps     :: Term → [Type] → Term
lambdas    :: [(Var, Type)] → Term → Term
tyLambdas :: [(TyVar, Kind)] → Term → Term

type Type = ...                 -- See τ in Figure 2.1
type Kind = ...                 -- See κ in Figure 2.1

forAllTys  :: [(TyVar, Kind)] → Type → Type
funTys     :: [Type] → Type → Type

data QA = Question Var | Answer Value

type State  = (Heap, (Tag, QA), Stack)   -- Normalised
type UState = (Heap, Term, Stack)        -- Unnormalised

stateType :: State → Type
freeVars :: State → ([(TyVar, Kind)], [(Var, Type)])
rebuild :: State → Term
```

Figure 3.1: Types used in the basic supercompiler

```
sc :: History → State → ScpM Term
  -- The normaliser (Section 2.2.2)
normalise :: UState → State
  -- The evaluator (Section 3.3)
step :: State → Maybe UState
reduce :: State → State
  -- The splitter (Section 3.5)
split :: Monad m ⇒ (State → m Term)
                 → State → m Term
  -- Termination checking (Section 3.2)
type History = [State]
emptyHistory = [ ] :: History
data TermRes = Stop | Continue History
terminate :: History → State → TermRes
  -- Memoisation and the ScpM monad (Section 3.4)
memo :: (State → ScpM Term)
     → State → ScpM Term
match :: State → State → Maybe Subst
runScpM :: ScpM Term → Term
freshVar :: ScpM Var
unit     :: Var
unitTy   :: Type
bind     :: Var → Type → Term → ScpM ()
promises :: ScpM [Promise]
promise  :: Promise → ScpM ()
data Promise = P {name :: Var,
                  ftvs :: [TyVar], fvs :: [Var],
                  meaning :: State}
```

Figure 3.2: Types of the basic supercompiler

portunity to tie back. Unfortunately, for some functions—for example, one with an accumulating parameter—each recursive call looks different, so the memoiser will never encounter an identical state. So, if *memo* does not fire, *sc′* uses *terminate* to detect divergence (conservatively, of course). The *History* argument of *terminate* allows the caller to accumulate the *State*s that *terminate* has seen before.

3. The *splitter* (Section 3.5). If the *State* fails the termination test (the *Stop* branch), the supercompiler abandons the attempt to optimise the entire term. Instead, it splits the term into a *residual* “shell” and some (smaller) sub-terms which it then supercompiles, gluing the results back together into their context.

4. The *evaluator* (Section 3.3). If the *State* passes the termination test (the *Continue* branch), we call *reduce* to perform symbolic evaluation of the *State*. This may take many steps, but will eventually halt (it has its own internal termination test). When it does so, we split and residualise just as before. This use of *split* to recursively supercompile some subcomponents of the reduced state allows us to optimise parts of the term that reduction didn’t reach.

To get things rolling, the supercompiler is invoked as follows:

```
supercompile :: Term → Term
supercompile e = runScpM (sc emptyHistory init_state)
  where init_state = normalise (ε, e, [ ])
```

(The *normalise* function implements normalising reduction, as previously discussed in Section 2.2.2.)

A simple demonstration of the parts of *sc* we have seen so far is in order. Imagine that this state was input to *sc* (recall from Section 2.2.2 that we use expressions as shorthand for their corresponding normalised machine states):

```
let x = True; y = (λz. z) 1
in case x of True → Just y; False → Nothing
```

Assuming that this state has never been previously supercompiled, *sc′* will be invoked by *memo*, after *memo* associates this state with a new promise *h0* in the *ScpM* monad. Further assuming that the termination check in *sc′* returns *Continue*, we would *reduce* the input state to head normal form, giving a new state, $\mathcal{S}'$:

```
let y = (λz. z) 1 in Just y
```

The **case** computation and x binding have been reduced away. It would be possible to return this $\mathcal{S}'$ as the final, supercompiled form of our input—indeed, in general the supercompiler is free to stop at any time, using *rebuild* to construct a semantically-equivalent result term. However, doing so misses the opportunity to supercompile some *subcomponents* of $\mathcal{S}'$ that are not yet in head normal form. Instead, we feed $\mathcal{S}'$ to *split*, which:

1. Invokes *sc hist′* on the subterm $(\lambda z.\, z)\ 1$, achieving further supercompilation (and hence optimisation). Let's say for the purposes of the example that this then returns the final optimised term *h1*, with a corresponding optimised binding *h1* = 1 recorded in the monad.

2. Reconstructs the term using the optimised subexpressions. So in this case the *Term* returned by *split* would be **let** y = *h1* **in** *Just* y.

When *sc* returns to the top level call to *supercompile*, *runScpM* wraps the accumulated bindings around the final term, like so:

```
let h1 = 1
    h0 = let y = h1 in Just y
in h0
```

We will now consider each module involved in the definition of *sc* in more detail.

3.2 The termination criterion

The core of the supercompiler's termination guarantee is provided by a single function, *terminate*:

```
terminate :: History → State → TermRes
data TermRes = Stop | Continue History
```

As the supercompiler proceeds, it builds up an ever-larger *History* of previously-observed *State*s. This history is both interrogated and extended by calling *terminate*. Termination is guaranteed by making sure that *History* cannot grow indefinitely. More precisely, our system guarantees that for the function *terminate*, for any history $hist_0$ and states $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \ldots$ there can be no infinite sequence of calls to *terminate* of this form:

$$\begin{array}{rcl} terminate\ hist_0\ \mathcal{S}_0 & = & Continue\ hist_1 \\ terminate\ hist_1\ \mathcal{S}_1 & = & Continue\ hist_2 \\ & \ldots & \end{array}$$

Instead, there will always exist some j such that:

$$terminate\ hist_j\ \mathcal{S}_j \ = \ Stop$$

In Section 3.3 we will see how *reduce* uses *terminate* to ensure that it only performs a bounded number of reduction steps. The same *terminate* function (with a different *History*) is also used by *sc*, to ensure that it only recurses a finite number of times, thus ensuring the supercompiler is total (Section 3.7).

So much for the specification, but how can *terminate* be implemented? Of course, $(\lambda x\ y.\ Stop)$ would be a sound implementation of *terminate*, in that it satisfies the property described above, but it is wildly over-conservative because it forces the supercompiler to stop reduction *immediately*. We want an implementation of *terminate* that is correct, but which nonetheless waits for as long as possible before preventing further reduction by answering *Stop*.

One good way to implement such a termination criterion is by defining a *well-quasi-order* [Higman, 1952; Leuschel, 1998]. A relation $\trianglelefteq \in State \times State$ is a well-quasi-order (WQO) iff for all infinite sequences of elements of type *State* $(\mathcal{S}_0, \mathcal{S}_1, \ldots)$, it holds that: $\exists ij.\, i < j \wedge \mathcal{S}_i \trianglelefteq \mathcal{S}_j$. Given any WQO $\trianglelefteq \in State \times State$, we can then implement a sound *terminate* function:

```
type History = [State]
terminate :: History → State → TermRes
terminate prevs here | any (⊴ here) prevs = Stop
                     | otherwise          = Continue (here : prevs)
```

To implement a sound WQO on *State*s we follow Mitchell [2010] by extracting a compact summary of the *State* (a bag of tags), and compare these *tag-bags*[2]:

$$\mathcal{S}_1 \trianglelefteq \mathcal{S}_2 = tagBag\ \mathcal{S}_1 \trianglelefteq_b tagBag\ \mathcal{S}_2$$

The relation $(\trianglelefteq)$ is a WQO on *State*s if $\trianglelefteq_b$ is a WQO on tag-bags, where we again follow Mitchell in defining $\trianglelefteq_b$ as:

$$b_1 \trianglelefteq_b b_2 \iff set(b_1) = set(b_2) \wedge |b_1| \leq |b_2|$$

Where the expression $|b|$ returns the number of elements in the bag b, and the *set* function returns a set of all elements in the input bag (i.e. it removes duplicates).

Theorem 3.2.1. *If there are a finite number of tags, $\trianglelefteq_b$ is a well-quasi-order.*

Proof. See Section D.6.3. □

[2] It is simple matter to memoise the repeated use of *tagBag* on *prevs*, an optimisation we describe thoroughly in Section D.5.2.

We can compute a tag-bag from a *State* like so:

$$
\begin{array}{lcl}
\mathit{tagBag} :: \mathit{State} \to \mathit{TagBag} & & \\
\mathit{tagBag}\ \langle H \,|\, e^t \,|\, K \rangle & = & \mathit{tagBag}_H(H) \cup \{\!\{t\}\!\} \cup \mathit{tagBag}_K(K) \\
 & & \\
\mathit{tagBag}_H(H) & = & \{\!\{t \,|\, x{:}\tau \mapsto e^t \in H\}\!\} \\
\mathit{tagBag}_K(K) & = & \{\!\{t \,|\, \kappa^t \in K\}\!\}
\end{array}
$$

We use $\{\!\{\ldots\}\!\}$ notation to introduce bag literals and comprehensions with the standard meaning, and overload the set operator $\cup$ for bags. Notice carefully that we only collect the *root* tag from each heap binding, the focus term, and stack frame. We do not collect tags from internal nodes. Doing so would make the bags larger without making the termination test more discriminating because, remembering that tags all derive from the originally-tagged input program, a node tagged t will usually have the same children tags as another node tagged t.

Appendix D discusses further the theory of well-quasi-orders and the design of a Haskell library concretely implementing several lemmas about them. Section D.6.3 shows how the resulting library can be used to define a correct-by-construction *terminate* function exactly equivalent to the one we discuss above: this is the definition of the termination check we use in our implementation.

3.3 Reduction

The reduction rules are used in the supercompiler in two places. Firstly, they are used to implement the *normalise* function, and secondly they are used to implement *reduce*.

We discuss *normalise* first. The *State* type is the type of *normalised* states[3], and we represent the fact that the focus of such a state will only ever be a variable or a value by introducing a new sum type *QA* (see Figure 3.1). The *normalise* function allows a non-normalised state (*UState*) to be injected into the *State* data type in the obvious way:

$$\mathit{normalise} :: \mathit{UState} \to \mathit{State}$$

All of the modules of the supercompiler accept only normalised states, and *normalise* is used by any module whenever it needs to convert some unnormalised state into a normalised one to pass to another module.

Normalisation performs lots of reduction, but the supercompiler still needs to perform some potentially non-normalising reduction steps in order to make good progress.

The *reduce* function tries to reduce a *State* to head normal form. In case the term diverges, *reduce* includes a termination check that allows it to stop after a finite number of steps. (This check is conservative, of course, so *reduce* might fail to find a head normal form when one does exist.) The two key properties of *reduce* are:

- Reduction preserves meaning: the input and output *State*s have the same semantics
- Regardless of what meaning the input *State* may have, *reduce* always terminates

We can implement *reduce* in terms of a function *step* that performs the non-normalising BETA and TYBETA reduction rules of Figure 2.3.

$$\mathit{step} :: \mathit{State} \to \mathit{Maybe}\ \mathit{UState}$$

[3] To reduce clutter, we leave implicit the treatment of the typing and kinding contexts.

If either of the β rules fire, *step* returns an unnormalised state, *UState*; otherwise it returns *Nothing*. We can use *step* and *normalise* together to build the terminating multi-step evaluator *reduce*, which in turn is called by the main supercompiler *sc′*:

```
reduce :: State → State
reduce = go emptyHistory
  where
    go hist 𝒮 = case terminate hist 𝒮 of
      Stop          → 𝒮
      Continue hist′ → case step 𝒮 of
        Nothing → 𝒮
        Just u𝒮′ → go hist′ (normalise u𝒮′)
```

The totality of *reduce* is achieved using the *terminate* function from Section 3.2. If *terminate* reports that evaluation appears to be diverging, *reduce* immediately returns. As a result, the *State* triple $\langle H \mid d \mid K \rangle$ returned by *reduce* might not be fully reduced (though it will of course be normalised)—in particular, it might be the case that K begins with the frame $\bullet\ x$, and $d \equiv f^t$ where f is bound to a function value in H.

3.4 Memoisation and matching

The purpose of the memoisation function, *memo*, is to ensure that we reuse the results of supercompiling a *State* if we come to supercompile an equivalent *State* later on.

We achieve this by using the *ScpM* state monad: whenever we supercompile a new *State* we give it a fresh name of the form *hn* for some n, and record the *State* and name in the monad as a *promise*. The supercompiled version of that *State* will be λ-abstracted over its free variables and bound to that name at the top level of the output program. Now if we are ever asked to supercompile a later *State* equivalent to that earlier one (up to renaming) we *tie back* by just looking in the promises and producing a call to the name of the earlier state, applied to appropriate arguments.

Precisely, the *ScpM* monad is a simple state monad with three pieces of state:

1. The *promises*, which comprise all the *State*s that have been previously submitted for supercompilation, along with:

   - The names that the supercompiled versions of those *State*s will be bound to in the final program (e.g. *h0*, *h1*)
   - The list of free term variables and type variables that those bindings will be abstracted over. By instantiating these free variables several different ways, we can reuse the supercompiled version of a *State* several times.

   The data structure used to store all this information is called a *Promise* (Figure 3.2).

2. The *optimised bindings*, each of the form $x : \tau = e$. The *runScpM* function, which is used to actually execute *ScpM Term* computations, wraps the optimised bindings collected during the supercompilation process around the final supercompiled *Term* using a **let** in order to produce the final output.

3. A supply of fresh names (*h0*, *h1*, ...) to use for the optimised bindings.

When *sc* begins to supercompile a *State*, it records a promise for that state; when it finishes supercompiling that state it records a corresponding optimised binding for it. At any moment there may be unfulfilled promises that lack a corresponding binding, but every binding has a corresponding promise. Moreover, every promise will *eventually* be fulfilled by an entry appearing in the optimised bindings. Figure 3.2 summarises the signatures of the functions provided by *ScpM*.

We can now implement *memo* as follows:

```
memo :: (State → ScpM Term)
     → State → ScpM Term
memo opt 𝒮 = do
  ps ← promises
  let ress = [ (var (name p) ‘tyApps‘ map (substTyVar θ) (ftvs p)
                              ‘apps‘ (unit : map (substVar θ) (fvs p)))
             | p ← ps
             , Just θ ← [match (meaning p) 𝒮]
             ]
  case ress of
    res : _ → return res
    [ ]     → do
      hn ← freshVar
      let (tvs, vs) = freeVars 𝒮
      promise P {name = hn,
                 ftvs = map fst tvs, fvs = map fst vs,
                 meaning = 𝒮}
      e ← opt 𝒮
      bind hn (forAllTys tvs (funTys (unitTy : map snd vs) (stateType 𝒮)))
              (tyLambdas tvs (lambdas ((unit, unitTy) : vs) e))
      return (var hn ‘tyApps‘ map fst tvs ‘apps‘ map fst (unit : vs))
```

The *memo* function proceeds as follows:

1. Firstly, it examines all existing *promises*. If the *match* function (Section 3.4.2) reports that some existing promise matches the *State* we want to supercompile (up to renaming), *memo* returns a call to the optimised binding corresponding to that existing promise.

2. Assuming no promise matches, *memo* continues:

   (a) A new promise for this novel *State* is made, in the form of a new *Promise* entry. A fresh *name* of the form *hn* (for some *n*) is associated with the *Promise*.

   (b) The input state $\mathcal{S}$ is optimised by calling *opt* (i.e. the supercompiling function sc'), obtaining an optimised term e.

   (c) A final optimised binding $hn = \Lambda\overline{\mathrm{ftvs}\,(\mathcal{S})}.\,\lambda\overline{\mathrm{fvs}\,(\mathcal{S})}.\,e$ is recorded using *bind*. This binding will be placed in a **let** that encloses the output program by *runScpM*.

   (d) Finally, a call to that binding, $hn\ \overline{\mathrm{ftvs}\,(\mathcal{S})}\ \overline{\mathrm{fvs}\,(\mathcal{S})}$, is returned.

Note that because we are working in a typed language, the new bindings created by *memo* are abstracted over not just their free term variables, but also their free type variables. We abstract over *all* the type variables before *any* of the term variables to ensure that the resulting term is well-scoped, since the types of the term variables can mention the free type variables.

3.4.1 Avoiding unintended sharing

We additionally abstract every h-function over a variable unit of type unitTy, and apply the same variable whenever making a call to a h-function. We expect that:

- unit is the name of a fresh variable not bound anywhere else
- unitTy is the type of the nullary tuple, $()$
- $\mathit{runScpM}$ will wrap a binding $\mathit{unit}{:}\mathit{unitTy} = ()$ around the final output term along with all of the h-function bindings added by bind, to ensure that the initial free occurrence of unit is bound

The purpose of this extra abstraction is to ensure that every h-function binding is a *value* and is not subject to call-by-need work-sharing. If we did not add these extra abstractions, then we might end up increasing work sharing for terms which have no free type or term variables[4]. For example, we could transform the term $(\mathit{fib}\ 100, \mathit{fib}\ 100)$ into $\mathbf{let}\ h = \mathit{fib}\ 100\ \mathbf{in}\ (h, h)$. With the extra abstraction, we instead supercompile it to $\mathbf{let}\ \mathit{unit} = (); h = \lambda \mathit{unit}.\, \mathit{fib}\ 100\ \mathbf{in}\ (h\ \mathit{unit}, h\ \mathit{unit})$, which has the same work-sharing properties as the input.

Increasing work sharing may look harmless and perhaps even beneficial in some cases, but it can cause unexpected "space leaks" [Hughes, 1983] if the evaluated form of a binding causes more memory to be live for garbage collection purposes than the unevaluated version does. Space leaks are the reason that general common subexpression elimination is not commonly used for call-by-need language [Chitil, 1997], and we also want to avoid introducing space leaks during supercompilation.

For clarity of exposition, we will almost always omit these unit abstractions, applications and bindings from our examples.

3.4.2 Matching

The match function is used to compare States. If the two States being matched are seen to be the same up to renaming, a substitution (of type Subst) from the first argument State to the second is returned:

$$\mathit{match} :: \mathit{State} \to \mathit{State} \to \mathit{Maybe}\ \mathit{Subst}$$

A substitution θ is a mapping from term variables to term variables and from type variables to types. A more usual notion of substitution would contain a mapping from term variables to terms, but this an unnecessary complication in our ANFed setting. We will assume functions are available to apply substitutions to various syntactic entities (Figure 3.1).

The key properties of the match function are that:

- If two states match, they should have the same meaning. One way to phrase this is to say that if $\mathit{match}\ \mathcal{S}_0\ \mathcal{S}_1\ =\ \mathit{Just}\ \theta$ then the meaning of $\mathit{rebuild}\ \mathcal{S}_1$ is the same as that of $\mathit{substTerm}\ \theta\ (\mathit{rebuild}\ \mathcal{S}_0)$, i.e.

  $$(\mathit{rebuild}\ \mathcal{S}_0)\theta \underset{\sim}{\lhd\!\rhd} \mathit{rebuild}\ \mathcal{S}_1$$

  (The $\mathit{rebuild}$ function (whose type is given by Figure 3.1) turns a State back into a Term, i.e. using the definition of Section 2.2, $\mathit{rebuild}\ \langle H \,|\, d \,|\, K \rangle = (\!| H \,|\, d \,|\, K |\!))$

[4] Our actual implementation only inserts unit abstractions if there are no other variables to be abstracted over: for simplicity, the memo we define here always inserts them.

In fact, in order to prove that memoisation is correct, we will require a slightly refined version of this property to hold (Section 3.8, Definition 3.8.1). However, this simpler version of the property is an excellent intuitive guide to what is expected of a matcher.

- If a state $\mathcal{S}_0$ is syntactically identical to $\mathcal{S}_1$, modulo renaming of free variables, then $\mathit{isJust}\ (\mathit{match}\ \mathcal{S}_0\ \mathcal{S}_1)$. This property is necessary for termination of the supercompiler, as we will discuss in Section 3.7.3.

Naturally, it is desirable for the *match* function to match as many truly equivalent terms as possible while obeying these constraints. We discuss the implementation of a *match* function for a call-by-need language that respects these two properties in Section 5.2.

3.5 The splitter

The job of the splitter is to complete the process of supercompiling a *State* whose reduction is stuck, either because of a lack of information (e.g. if the *State* is blocked on a free variable), or because the termination criterion is preventing us from reducing that *State* further. The splitter has the following type signature[5]:

$$\begin{aligned}\mathit{split} :: \mathit{Monad}\ m \Rightarrow (&\mathit{State} \to m\ \mathit{Term})\\ &\to \mathit{State} \to m\ \mathit{Term}\end{aligned}$$

In general, $(\mathit{split}\ \mathit{opt}\ \mathcal{S})$ identifies some sub-components of the state $\mathcal{S}$, uses *opt* to optimise them, and combines the results into a term whose meaning is the same as $\mathcal{S}$ (assuming, of course, that *opt* preserves meaning), i.e. essentially[6] we expect that:

$$(\forall \mathcal{S}.\mathit{rebuild}\ \mathcal{S} \underset{\sim}{\triangleright} \mathit{opt}\ \mathcal{S}) \implies \forall \mathcal{S}.\mathit{rebuild}\ \mathcal{S} \underset{\sim}{\triangleright} \mathit{split}\ \mathit{opt}\ \mathcal{S}$$

A sound, but feeble, implementation of $\mathit{split}\ \mathit{opt}\ \mathcal{S}$ would be one which *never* recursively invokes *opt*:

$$\mathit{split}\ \_\ \mathcal{S} = \mathit{return}\ (\mathit{rebuild}\ \mathcal{S})$$

Such an implementation is wildly conservative, because not even trivially reducible subexpressions will benefit from supercompilation. A good *split* function will residualise as little of the input as possible, using *opt* to optimise as much as possible.

It turns out that, starting from this sound-but-feeble baseline, there is a rich variety of choices one can make for *split*, and it is difficult to write an implementation that is simultaneously sound and reasonably non-conservative. We discuss the issues surrounding *split*, as well as the particular implementation we use in our call-by-need supercompiler, in much more detail in Chapter 4.

3.6 A complete example

Here is an example of the supercompiler in action. Consider the function *map*, whose tagged definition is as follows:

[5] In fact, we will only instantiate the *Monad m* in the type signature of *split* to *ScpM*, but this signature makes clear that *split* does not make use of any of the monad-carried information.

[6] Similar to the situation in Section 3.4.2 this property is not what we actually expect *split* to satisfy: the true property is discussed in Section 3.8.

$$\begin{array}{l} \mathit{map}\ f\ \mathit{xs} = (\textbf{case}\ \mathit{xs}^{t_1}\ \textbf{of}\ [\,] \to [\,] \\ \qquad\qquad\qquad\qquad\qquad\quad (y : \mathit{ys}) \to ((f\ y)^{t_2} : ((\mathit{map}^{t_3}\ f)^{t_4}\ \mathit{ys})^{t_5})^{t_6})^{t_7} \end{array}$$

Now, suppose we supercompile the call $((\mathit{map}^{t_a}\ \mathit{not})^{t_b}\ \mathit{xs})^{t_c}$, which inverts every element in a list of Booleans, xs. Remember (Section 3.1) that *supercompile* normalises the expression before giving it to *sc*. Normalisation will evaluate the term until it gets stuck, which is nearly immediate, because it needs to inline *map*. So the initial *State* given to *sc* is this:

$$\mathcal{S}_0 = \left\langle \mathit{map} \mapsto (\ldots)^{t_9}, \mathit{not} \mapsto (\ldots)^{t_8} \,\middle|\, \mathit{map}^{t_a} \,\middle|\, \bullet\ \mathit{not}^{t_b}, \bullet\ \mathit{xs}^{t_c} \right\rangle$$

As this is the first invocation of *sc*, there is no way for *memo* to tie back, so *sc′* is called. The history is empty, so the termination check passes—after extending the history with the tag-bag $\{\!\{t_a, t_b, t_c, t_9, t_8\}\!\}$—and so *sc′* calls *reduce* which evaluates the state (including inlining; see Section 3.3) until it gets stuck because it has no binding for xs:

$$\left\langle \mathit{map} \mapsto (\ldots)^{t_9}, \mathit{not} \mapsto (\ldots)^{t_8} \,\middle|\, \mathit{xs}^{t_1} \,\middle|\, \kappa^{t_7} \right\rangle$$

where κ is the stack frame for the case expression:

$$\kappa = \textbf{case} \bullet \textbf{of} \left\{ \begin{array}{l} [\,] \to [\,] \\ (y : \mathit{ys}) \to ((\mathit{not}\ y)^{t_2} : ((\mathit{map}^{t_3}\ \mathit{not})^{t_4}\ \mathit{ys})^{t_5})^{t_6}) \end{array} \right\}$$

Now *split* residualises part of the reduced *State* ($\textbf{case}\ \mathit{xs}\ \textbf{of}\ [\,] \to \ldots; (y : \mathit{ys}) \to \ldots$) and recursively supercompiles the two branches of the case. We concentrate on the (:) branch. Once again the *State* constructed for this branch is normalised before being passed to the recursive invocation of *sc*. The normalised state looks like this:

$$\left\langle \begin{array}{l} \mathit{map} \mapsto (\ldots)^{t_9}, \mathit{not} \mapsto (\ldots)^{t_8}, \\ z \mapsto (\mathit{not}\ y)^{t_2}, \mathit{zs} \mapsto ((\mathit{map}^{t_3}\ \mathit{not})^{t_4}\ \mathit{ys})^{t_5} \end{array} \,\middle|\, (z : \mathit{zs})^{t_6} \,\middle|\, \epsilon \right\rangle$$

We cannot tie back at this point because this state does not match any existing promise. Furthermore, the tag-bag for this state, $\{\!\{t_2, t_5, t_6, t_9, t_8\}\!\}$, is distinct (as a set) from that for the previous *State*, and so supercompilation proceeds in the *Continue* branch of *sc′* by splitting the *State* (*reduce* is the identity function on this *State* as it is already a value). Both the head z and tail zs of the output list are recursively supercompiled, but we focus on the tail. Normalising gives:

$$\mathcal{S}_2 = \left\langle \mathit{map} \mapsto (\ldots)^{t_9}, \mathit{not} \mapsto (\ldots)^{t_8} \,\middle|\, \mathit{map}^{t_3} \,\middle|\, \bullet\ \mathit{not}^{t_4}, \bullet\ \mathit{xs}^{t_5} \right\rangle$$

When *sc* is invoked on $\mathcal{S}_2$, the memoiser notices that it has already supercompiled the α-equivalent state $\mathcal{S}_0$, and so it returns immediately with a *Term* that just invokes the corresponding h-function, *h0*.

The final output program (including the *h3* and *h4* functions generated by invocations of *sc* that we have elided) is thus:

$$\begin{array}{l} \mathit{h0}\ \mathit{xs} = \textbf{case}\ \mathit{xs}\ \textbf{of}\ [\,] \qquad\ \to \mathit{h3} \\ \qquad\qquad\qquad\qquad\ \ (y : \mathit{ys}) \to \mathit{h1}\ y\ \mathit{ys} \\ \mathit{h1}\ y\ \mathit{ys} = \mathit{h4}\ y : \mathit{h0}\ \mathit{ys} \\ \mathit{h3} = [\,] \\ \mathit{h4}\ y = \mathit{not}\ y \end{array}$$

After inlining *h1*, *h2* and *h4* we recognise *h0* as a version of *map* specialised for *not* as its first argument.

3.7 Termination of the supercompiler

The focus of this thesis is not to provide a completely formal proof that the supercompiler we define is terminating or correct. However, we will lay out the properties we expect to hold, will formally prove substantial theorems, and clearly mark conjectures where they are necessary (principally in Chapter 4 and Chapter 5). For the purposes of these proofs, we will often omit tags, since it is only important that there is *some* tag information present: the actual values of the tags themselves are irrelevant for the purposes of the proofs. In this section, we tackle the issue of supercompiler termination.

3.7.1 Two non-termination checks

Although we have been careful to ensure that our evaluation function, *reduce*, is total, it is not so obvious that *sc* itself is terminating. Since *split* may recursively invoke *sc* via its higher order argument, we might get an infinitely deep stack of calls to *sc*.

To rule out this possibility, *sc* carries a history, which—as we saw in Section 3.1—is checked before any reduction is performed. If *terminate* allows the history to be extended, the input *State* is reduced before recursing. Otherwise, the input *State* is fed to *split* unchanged. Note that we do *not* simply stop supercompiling by returning the *rebuild*ing of the input *State* if the termination test fails: by passing the *State* to *split* unchanged instead, we ensure that we have a chance to optimise all the subcomponents of the current state.

It is important to note that the history carried by *sc* is extended entirely independently from the history produced by the *reduce* function (this is similar to the concept "transient reductions" [Sørensen and Glück, 1999]). The two histories deal with different sources of non-termination.

The history carried by *reduce* prevents non-termination due to divergent expressions, such as this one:

$$\textbf{let}\, f\ x = 1 + (f\ x)\ \textbf{in}\, f\ 10$$

In contrast, the history carried by *sc* prevents non-termination that can arise from repeatedly invoking the *split* function—even if every subexpression would, considered in isolation, terminate. This is illustrated in the following program:

$$\textbf{let}\ \mathit{count}\ n = n : \mathit{count}\ (n+1)\ \textbf{in}\ \mathit{count}\ 0$$

Left unchecked, we would repeatedly *reduce* the calls to *count*, yielding a value (a cons-cell) each time. The *split* function would then pick out both the head and tail of the cons cell to be recursively supercompiled, leading to yet another unfolding of *count*, and so on. The resulting (infinite) residual program would look something like:

$$\begin{aligned}
&\textbf{let}\ \mathit{h0} = \mathit{h1} : \mathit{h2}; \mathit{h1} = 0\\
&\quad\ \ \mathit{h2} = \mathit{h3} : \mathit{h4}; \mathit{h3} = 1\\
&\quad\ \ \mathit{h4} = \mathit{h5} : \mathit{h6}; \mathit{h5} = 2\\
&\quad\ \ \ldots
\end{aligned}$$

The check with *terminate* before reduction ensures that instead, one of the applications of *count* is left unreduced. This use of *terminate* ensures that our program remains finite:

$$\begin{aligned}
&\textbf{let}\ \mathit{h0} = \mathit{h1} : \mathit{h2}; \mathit{h1} = 0\\
&\quad\ \ \mathit{h2} = \textbf{let}\ \mathit{count} = \lambda n.\, \mathit{h3}\ n
\end{aligned}$$

```
            in count 1
        h3 n = n : h3 (n + 1)
    in h0
```

3.7.2 Proof of termination without recursive let

In order to be able to prove that the supercompiler terminates, we need some condition on exactly what sort of subcomponents *split opt* invokes *opt* on. (It is also clear that if *opt* terminates on all inputs then *split* itself should not diverge, and hence that *split* should only invoke *opt* finitely many times.)

Let us pretend for a moment that we have no recursive **let** (we consider the case of recursive **let** shortly, in Section 3.7.3). In this scenario, you can show that for our *split* (defined in Chapter 4), split-recursion shrinks:

Definition 3.7.1 (Split-recursion shrinks)**.** We say that split-recursion shrinks if *split opt* $\mathcal{S}$ invokes *opt* $\mathcal{S}'$ only if $\mathcal{S}' \prec \mathcal{S}$. The $\prec$ relation is a well-founded relation defined by $\mathcal{S}' \prec \mathcal{S} \iff \mathit{size}(\mathcal{S}') < \mathit{size}(\mathcal{S})$, where $\mathit{size} : \mathit{State} \to \mathbb{N}$ returns the number of abstract syntax tree nodes in the *State*.

It is possible to prove property 3.7.1 for *split* with non-recursive input terms, and this property is sufficient to ensure termination, as the following argument shows:

Theorem 3.7.1 (Well-foundedness with nonrecursive let)**.** *If split-recursion shrinks, sc always recurses a finite number of times.*

Proof. Proceed by contradiction. If *sc* recursed an infinite number of times, then by definition the call stack would contain infinitely many activations of *sc hist* $\mathcal{S}$ for (possibly repeating) sequences of *hist* and $\mathcal{S}$ values. Denote the infinite chains formed by those values as $\langle \mathit{hist}_0, \mathit{hist}_1, \ldots\rangle$ and $\langle \mathcal{S}_0, \mathcal{S}_1, \ldots\rangle$ respectively.

Now, observe that there it must be the case that there are infinitely many i for which the predicate *isContinue* $(\mathit{terminate}\ \mathit{hist}_i\ \mathcal{S}_i)$ holds. This follows because the only other possibility is that there must be some j such that $\forall l. l \geq j \implies \mathit{isStop}\ (\mathit{terminate}\ \mathit{hist}_l\ \mathcal{S}_l)$. On such a suffix, *sc* is recursing through *split* without any intervening uses of *reduce*. However, by the property we required *split* to have, such a sequence of states must have a strictly decreasing size:

$$\forall l. l > j \implies \mathit{size}(\mathcal{S}_l) < \mathit{size}(\mathcal{S}_j)$$

However, < is a well founded relation, so such a chain cannot be infinite. This contradicts our assumption that this suffix of *sc* calls is infinite, so it must be the case that there are infinitely many i such that *isContinue* $(\mathit{terminate}\ \mathit{hist}_i\ \mathcal{S}_i)$.

Now, form the infinite chain $\langle \mathcal{S}'_1, \mathcal{S}'_2, \ldots\rangle$ consisting of a restriction of $\mathcal{S}_i$ of to those elements for which *isContinue* $(\mathit{terminate}\ \mathit{hist}_i\ \mathcal{S}_i)$ holds. By the properties of *terminate*, it follows that:

$$\forall ij. j < i \implies \neg\left(\mathit{tagBag}\ \mathcal{S}'_j \trianglelefteq \mathit{tagBag}\ \mathcal{S}'_i\right)$$

However, this contradicts the fact that $\trianglelefteq$ is a well-quasi-order. □

Combined with the requirement that *split opt* only calls *opt* finitely many times, the whole supercompilation process must terminate.

It is interesting to note that this argument still holds even if the supercompiler does not use *memo*. Nonetheless, in practice the results of supercompilation will be much better if we use *memo* as in many cases we will be able to avoid failing the termination test (and hence generating bad code) by tying back to a previous state.

3.7.3 Extension to recursive let

So much for the case where there is no recursive **let**. In practice almost all programs we will encounter will make some use of recursive **let**, even if it is only in the definition of library functions such as *map*. It turns out we can repair the argument above to deal with recursive **let** by proving a more complex property about *split*. Unlike our previous argument, this version will make fundamental use of the presence of memoisation.

First, let us understand why the argument above breaks down with recursive **let**. Consider this input to *split*:

$$\mathcal{S} = \langle \mathit{repeat} \mapsto \lambda y.\, \mathbf{let}\ \mathit{ys} = \mathit{repeat}\ y\ \mathbf{in}\ y : \mathit{ys} \,|\, \mathit{repeat} \,|\, \bullet\ y \rangle$$

The splitter defined in Chapter 4 would recursively supercompile this state:

$$\mathcal{S}' = \langle \mathit{repeat} \mapsto \lambda y.\, \mathbf{let}\ \mathit{ys} = \mathit{repeat}\ y\ \mathbf{in}\ y : \mathit{ys}, \mathit{ys} \mapsto \mathit{repeat}\ y \,|\, y : \mathit{ys} \,|\, \epsilon \rangle$$

This state is already larger than the original state by the metric of Definition 3.7.1, so we can see that if we have recursive **let** we certainly can't prove that split-recursion shrinks. In fact, we can go further and observe that *split*ting $\mathcal{S}'$ would cause us to recursively supercompile two states, one of which (arising from the *ys* binding) would be:

$$\langle \mathit{repeat} \mapsto \lambda y.\, \mathbf{let}\ \mathit{ys} = \mathit{repeat}\ y\ \mathbf{in}\ y : \mathit{ys} \,|\, \mathit{repeat} \,|\, \bullet\ y \rangle$$

Note that this forms a loop, so we won't be able to prove that split-recursion shrinks even if we make a different choice of state-size metric than the syntax-node metric of Definition 3.7.1.

In the presence of recursive **let**, we can instead show that for our *split* a different property holds:

Definition 3.7.2 (Split-recursion rearranges)**.** *split opt* $\langle H \,|\, e \,|\, K \rangle$ only invokes *opt* on states $\langle H' \,|\, e' \,|\, K' \rangle$ that satisfy all of these conditions:

1. $H' \subseteq H \cup \mathit{alt\text{-}heap}\,(e, K)$

2. K' *'isInfixOf'* K

3. $e' \in \mathit{subterms} \langle H \,|\, e \,|\, K \rangle$

The $\mathit{subterms}\langle H \,|\, e \,|\, K \rangle$ function returns all expressions that occur syntactically within the heap, stack or focus of the input state. The standard library function *isInfixOf* tests whether the left-hand argument occurs as a contiguous sublist within the right-hand argument. The function $\mathit{alt\text{-}heap}\,(e, K)$ takes the variables bound by update frames in K and, if $e \equiv x$ for some x, the variable x. It then forms the cross product of that set with the values $\mathbf{C}\ \overline{\alpha}\ \overline{x}$ in any $\mathbf{case} \bullet \mathbf{of}\, \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau} \ \to\ e} \in K$. This *subterms* function is required only to deal with those new bindings that arise from positive information propagation (Section 4.2).

Essentially what this property says is that the states recursively supercompiled by *split* can be formed only by "rearranging" the components of the input state: entirely new syntax cannot be generated out of thin air. This will allow us to prove that chains of successive recursive *split* calls can only create a finite number of distinct states.

Theorem 3.7.2. *The split defined in Chapter 4 obeys the property of Definition 3.7.2.*

Proof. Straightforward inspection of the algorithm: in particular, see the definitions in Section 4.3.3. □

With this, we are in a position to repair the earlier proof.

Theorem 3.7.3 (Well-foundedness)**.** *sc always recurses a finite number of times.*

Proof. Proceed by contradiction. If *sc* recursed an infinite number of times, then by definition the call stack would contain infinitely many activations of $sc\ hist\ \mathcal{S}$ for (possibly repeating) sequences of *hist* and $\mathcal{S}$ values. Denote the infinite chains formed by those values as $\langle hist_0, hist_1, \ldots\rangle$ and $\langle \mathcal{S}_0, \mathcal{S}_1, \ldots\rangle$ respectively. We define H_l, K_l and e_l such that $\mathcal{S}_l \equiv \langle H_l \,|\, e_l \,|\, K_l\rangle$.

Now, observe there must be infinitely many i such that $isContinue\ (terminate\ hist_i\ \mathcal{S}_i)$. This follows because the only other possibility is that there must exist some j such that $\forall l.l \geq j \implies isStop\ (terminate\ hist_l\ \mathcal{S}_l)$. On such a suffix, *sc* is recursing through *split* without any intervening uses of *reduce*. By the modified property of *split* and the properties of *alt-heap* and *subterms* we have that

$$\begin{aligned}&\forall l.l \geq j \implies\\ &\quad H_l \subseteq H_j \cup alt\text{-}heap(e_j, K_j)\\ &\wedge\ \ K_l\ \text{`}isInfixOf\text{`}\ K_j\\ &\wedge\ \ e_l \in subterms(\mathcal{S}_j)\end{aligned}$$

We can therefore conclude that the infinite suffix must repeat itself at some point (up to renaming of the free variables): $\exists l.l > j \wedge \mathcal{S}_l = \mathcal{S}_j$. However, we required that *match* always succeeds when matching two terms equivalent up to renaming, which means that $sc\ hist_l\ \mathcal{S}_l$ would have been tied back by *memo* rather than recursing. This contradicts our assumption that this suffix of *sc* calls is infinite, so it must be the case that there are infinitely many i such that $isContinue\ (terminate\ hist_i\ \mathcal{S}_i)$.

Now, form the infinite chain $\langle \mathcal{S}'_1, \mathcal{S}'_2, \ldots\rangle$ consisting of that restriction of $\mathcal{S}_i$ for which $isContinue\ (terminate\ hist_i\ \mathcal{S}_i)$ holds. As in Section 3.7, this contradicts the fact that $\trianglelefteq$ is a well-quasi-order. □

Note that although the termination argument becomes more complex in the recursive-**let** case, the actual supercompilation algorithm remains as simple as ever.

3.7.4 Negative recursion in data constructors

As a nice aside, the rigorous termination criterion gives us a stronger termination guarantee than GHC [Peyton Jones et al., 1992]. Because GHC does not check for recursion through negative positions in data constructors, the following notorious program would (until very recently [Peyton-Jones, 2012]) force GHC into an infinite loop:

```
data U = MkU (U → Bool)
russel u@(MkU p) = not (p u)
x = russel (MkU russel) :: Bool
```

3.8 Correctness of the supercompiler

In the previous section we argued that the supercompiler terminated. In this section we will argue that it is correct—i.e. the supercompiled program has the same meaning as the input program. Since we are working an a call-by-need setting, this entails not only proving that supercompilation preserves the denotational meaning of the program, but also that we don't lose work sharing.

The standard way [Klyuchnikov, 2010b] to prove the correctness of a supercompilation algorithm is to use an argument based on the improvement theory [Sands, 1995] introduced in Section 2.3. The main theorem that is generally proven is that supercompilation is an improvement:

$$e \mathrel{\underset{\sim}{\triangleright}} \mathit{supercompile}\ e$$

I.e. if the input program terminates in a given context, the supercompiled program will terminate in the same context in the same number of steps or less. This theorem is sufficient to prove that supercompilation is correct, as long as you are willing to accept that a supercompiler may sometimes turn a non-terminating program into a terminating one. For conciseness in our proofs, this is the definition of correctness that we will use[7].

The principal reason that we use improvement theory as the tool to prove that our supercompiler is correct, rather than arguing directly at the level of e.g. denotational semantics, is that improvement theory gives a set of tools for reasoning about recursion in our programs. If we were to make a naive argument with denotational semantics, we might look at the program:

$$\textbf{let}\ f\ x = 42\ \textbf{in}\ f\ 10$$

We might claim that since $f\ x$ is denotationally equal to 42 for any x it "must" be the case that we can replace the subterm 42 in the program like so:

$$\textbf{let}\ f\ x = f\ x\ \textbf{in}\ f\ 10$$

As the resulting program loops, our argument clearly was deeply flawed: simple denotional equality does not tell us when we can rewrite subterms of a program.In contrast, improvement theory gives us a simple and clear basis for doing so in a way which will not affect the termination properties the programs we are working on.

A nice benefit of proving that supercompilation is an improvement is that it establishes that supercompilation is actually an optimisation in some sense. Of course, just because the resulting program is faster (or at least no worse) in the abstract sense of "number of steps" does not mean it is faster on actual hardware, but it is a good first step.

Because our supercompiler design memoises more frequently[8] than standard supercompilers we cannot prove the standard theorem. Instead, we will prove the closely related property that supercompilation is an improvement *up to delay*:

$$\mathit{delay}_e(e) \mathrel{\underset{\sim}{\triangleright}} \mathit{supercompile}\ e$$

The *delay* family of functions is defined in Figure 3.3. When reading these functions, it may be helpful to remember that the purpose of the $^{\checkmark}e$ construct (which was introduced in Section 2.3) is to delay evaluation of a term by one non-normalising reduction step.

This improvement-up-to-delay property of *supercompile* says that if a slowed-down version of the input program terminates in some context, the supercompiled program will terminate in the same context in the same number of steps or less. Note that unlike the

[7] It is generally possible to show that supercompilation is in fact a "strong improvement" in that it is both an improvement *and* never changes a non-terminating program into a terminating one. Our supercompiler is no exception in this regard, but we omit the proof.

[8] Other supercompilers typically only memoise when there is a manifest β-reduction in the focus, we do so upon every invocation of *sc*.

previous property, the property we will show does *not* establish that supercompilation is an optimisation in the sense of improvement theory. However, it can be shown that $\mathit{delay}_{\mathsf{e}}(e)$ is only ever a constant multiplicative factor slower than e, i.e.

$$\exists k. e \Downarrow^{n} \implies \mathit{delay}_{\mathsf{e}}(e) \Downarrow^{\leq kn}$$

From this fact we can see that showing improvement-up-to-delay implies that work sharing is not lost by the supercompiler, since loss of work sharing can cause algorithms to become polynomially slower or worse. For example, the following function

$$\begin{array}{l} \mathit{sumLength}\ \mathit{xs} = \mathit{sum}\ (\mathit{replicate}\ n\ n) \\ \quad \textbf{where}\ n = \mathit{length}\ \mathit{xs} \end{array}$$

requires a number of reduction steps proportional to the length of the input list *xs* in call-by-need, but requires a number of reduction steps proportional to the *square* of the length in call-by-name.

In order to show that the supercompiler is correct, we of course need to assume some properties about the various modules of our supercompiler. Concretely, we will assume these properties:

Definition 3.8.1 (Matcher correctness)**.** The *match* function is correct if it succeeds only when the inputs are cost-equivalent *up to delay*:

$$\mathit{match}\ \mathcal{S}_0\ \mathcal{S}_1 \ = \ \mathit{Just}\ \theta \implies (\mathit{delay}\ \mathcal{S}_0)\theta \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} \mathit{delay}\ \mathcal{S}_1$$

Definition 3.8.2 (Splitter correctness)**.** The *split* function is correct if:

$$\begin{array}{l} \forall \mathcal{S}'. {}^{\checkmark}\, \mathit{delay}\ \mathcal{S}' \mathrel{\underset{\sim}{\triangleright}} \mathit{checkScpM}(\overline{\mathrm{p}}, \mathit{opt}\ \mathcal{S}') \\ \implies \mathit{delay}\ \mathcal{S} \mathrel{\underset{\sim}{\triangleright}} \mathit{checkScpM}(\overline{\mathrm{p}}, \mathit{split}\ \mathit{opt}\ \mathcal{S}) \end{array}$$

$$\begin{array}{rcl}
\mathit{delay}\,\langle \overline{x{:}\tau \mapsto e_x} \,|\, e \,|\, \epsilon \rangle & = & \textbf{let}\ \overline{x{:}\tau = \mathit{delay}_h(e_x)}\ \textbf{in}\ \mathit{delay}'_{\mathsf{e}}(e) \\
\mathit{delay}\,\langle H \,|\, e \,|\, \textbf{update}\ y{:}\tau, K \rangle & = & \mathit{delay}\,\langle H, y{:}\tau \mapsto e \,|\, y \,|\, K \rangle \\
\mathit{delay}\,\langle H \,|\, e \,|\, \bullet\ x, K \rangle & = & \mathit{delay}\,\langle H \,|\, e\ x \,|\, K \rangle \\
\mathit{delay}\,\langle H \,|\, e \,|\, \bullet\ \tau, K \rangle & = & \mathit{delay}\,\langle H \,|\, e\ \tau \,|\, K \rangle \\
\mathit{delay}\,\left\langle H \,\middle|\, e \,\middle|\, \textbf{case} \bullet \textbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau}\ \to\ e_{\mathbf{C}}}, K \right\rangle & & \\
& = & \mathit{delay}\,\left\langle H \,\middle|\, \textbf{case}\ e\ \textbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau}\ \to\ e_{\mathbf{C}}} \,\middle|\, K \right\rangle
\end{array}$$

$$\mathit{delay}_h(e) \ = \ \begin{cases} \mathit{delay}'_{\mathsf{e}}(e) & e \equiv v \\ \mathit{delay}_{\mathsf{e}}(e) & \text{otherwise} \end{cases} \qquad \mathit{delay}_{\mathsf{e}}(e) \ = \ {}^{\checkmark}\, \mathit{delay}'_{\mathsf{e}}(e)$$

$$\begin{array}{rcl}
\mathit{delay}'_{\mathsf{e}}(x) & = & x \\
\mathit{delay}'_{\mathsf{e}}(\lambda x{:}\tau.\, e) & = & \lambda x{:}\tau.\, \mathit{delay}_{\mathsf{e}}(e) \\
\mathit{delay}'_{\mathsf{e}}(e\ x) & = & \mathit{delay}'_{\mathsf{e}}(e)\ x \\
\mathit{delay}'_{\mathsf{e}}(\mathbf{C}\ \overline{\tau}\ \overline{x}) & = & \mathbf{C}\ \overline{\tau}\ \overline{x} \\
\mathit{delay}'_{\mathsf{e}}(\textbf{case}\ e\ \textbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau}\ \to\ e_{\mathbf{C}}}) & = & \textbf{case}\ \mathit{delay}'_{\mathsf{e}}(e)\ \textbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau}\ \to\ \mathit{delay}_{\mathsf{e}}(e_{\mathbf{C}})} \\
\mathit{delay}'_{\mathsf{e}}(\textbf{let}\ \overline{x{:}\tau\ =\ e_x}\ \textbf{in} e) & = & \textbf{let}\ \overline{x{:}\tau\ =\ \mathit{delay}_{\mathsf{e}}(e_x)}\ \textbf{in}\ \mathit{delay}'_{\mathsf{e}}(e) \\
\mathit{delay}'_{\mathsf{e}}(\Lambda \alpha{:}\kappa.\, e) & = & \Lambda \alpha{:}\kappa.\, \mathit{delay}_{\mathsf{e}}(e) \\
\mathit{delay}'_{\mathsf{e}}(e\ \tau) & = & \mathit{delay}'_{\mathsf{e}}(e)\ \tau
\end{array}$$

Figure 3.3: Delaying expressions and states

Where *checkScpM* is defined as follows:

$$\begin{array}{lcll}
\mathit{checkScpM} :: ([\,\mathit{Promise}\,], \mathit{ScpM}\ \mathit{Term}) \rightarrow \mathit{Term} & & & \\
\mathit{checkScpM}(\overline{p}, \mathit{me}) & = & \mathit{runScpM}\ \$\ \textbf{do} & \mathit{mapM}\ \mathit{promise}\ \overline{p} \\
 & & & e \leftarrow \mathit{me} \\
 & & & \mathit{return}\ \$\ \mathit{substPromises}\ (\overline{p}, e)
\end{array}$$

$$\begin{array}{rcl}
\multicolumn{3}{l}{\mathit{substPromises} :: ([\,\mathit{Promise}\,], \mathit{Term}) \rightarrow \mathit{Term}} \\
\mathit{substPromises}(\epsilon, e) & = & e \\
\mathit{substPromises}((p, \overline{p}), e) & = & \mathit{substPromises}(\overline{p}, e[e'/h]) \\
\textbf{where} \qquad h & = & \mathit{fun}\ p \\
(\overline{\alpha{:}\kappa}, \overline{x{:}\tau}) & = & \mathit{freeVars}\ (\mathit{meaning}\ p) \\
e' & = & \Lambda\overline{\alpha{:}\kappa}.\, \lambda \mathit{unit}{:}\,\mathit{unitTy}.\, \lambda\overline{x{:}\tau}.\, {}^{\checkmark}(\mathit{delay}\ (\mathit{meaning}\ p))
\end{array}$$

The $\mathit{substPromises}(\overline{p}, e)$ call is designed to eliminate all occurrences in e of the h-functions corresponding to the promises $\overline{p}$. The term e is in ANF, so if a h-function could ever occur as an argument in e we might have to insert a **let** to perform this elimination, but fortunately h-functions never occur in argument positions in the result of supercompilation.

We justify the properties of *match* and *split* as part of the relevant chapters (see Section 5.2.2 and Section 4.5), along with arguments as to why our implementations obey them. For now, we will be content to merely assume that they hold true and use them as part of our proof of correctness.

In order to show correctness, we also need some lemmas. The first lemma we require is the following:

Lemma 3.8.1 (Reduction improves up to delay)**.** *If $\mathcal{S} \rightsquigarrow \mathcal{S}'$, then delay $\mathcal{S} \underset{\sim}{\triangleright}$ delay $\mathcal{S}'$*

Proof. We proceed by cases on the reduction rules. We will only consider a few representative cases, the rest go through similarly.

Case VAR In this case,

$$\langle H, x{:}\tau \mapsto e \,|\, x \,|\, K \rangle \rightsquigarrow \langle H \,|\, e \,|\, \textbf{update}\ x{:}\tau, K \rangle$$

(where $(e \not\equiv v)$). This case follows immediately since:

$$\mathit{delay}\,\langle H \,|\, e \,|\, \textbf{update}\ x{:}\tau, K \rangle \ = \ \mathit{delay}\,\langle H, x{:}\tau \mapsto e \,|\, x \,|\, K \rangle$$

Case UPDATE2 In this case,

$$\langle H[x{:}\tau \mapsto u] \,|\, x \,|\, \textbf{update}\ y{:}\tau, K \rangle \rightsquigarrow \langle H, y{:}\tau \mapsto x \,|\, x \,|\, K \rangle$$

Since $\mathit{delay}\,\langle H \,|\, x \,|\, \textbf{update}\ y{:}\tau, K \rangle \ = \ \mathit{delay}\,\langle H, y{:}\tau \mapsto x \,|\, y \,|\, K \rangle$, this case is proven since it is easy to show that:

$$\left\langle y \mapsto {}^{\checkmark}x \,\middle|\, y \,\middle|\, \epsilon \right\rangle \underset{\sim}{\triangleright} \left\langle y \mapsto {}^{\checkmark}x \,\middle|\, x \,\middle|\, \epsilon \right\rangle$$

Case BETAV In this case,

$$\langle H \,|\, \lambda x{:}\tau.\, e \,|\, \bullet\ x, K \rangle \rightsquigarrow \langle H \,|\, e \,|\, K \rangle$$

Since $\mathit{delay}\,\langle H \,|\, \lambda x{:}\tau.\, e \,|\, \bullet\ x, K \rangle = \mathit{delay}\,\langle H \,|\, (\lambda x{:}\tau.\, e)\ x \,|\, K \rangle$, this case is proven since it is easy to show that:

$$\langle \epsilon \,|\, (\lambda x{:}\tau.\, e)\ x \,|\, \epsilon \rangle \underset{\sim}{\triangleleft\!\triangleright} \langle \epsilon \,|\, e \,|\, \epsilon \rangle$$

Case LETREC In this case,

$$\langle H \,|\, \mathbf{let}\ \overline{x\!:\!\tau\ =\ e_x}\ \mathbf{in}\ e \,|\, K \rangle \rightsquigarrow \langle H, \overline{x\!:\!\tau \mapsto e_x} \,|\, e \,|\, K \rangle$$

This case follows because **let**-floating out of an evaluation context is a cost-equivalence (assuming the variables $\overline{x}$ do not occur free outside the context, as in this case), and for all e, $\mathit{delay}_e(e) \underset{\sim}{\triangleright} \mathit{delay}_h(e)$.

□

We also require this lemma:

Lemma 3.8.2 (Binding and substitution interchangable)**.** *If all occurrences of h in $\hat{e}$ are saturated calls of the form $h\ \overline{\upsilon}_i^n\ (\mathit{unit}, \overline{y}_i^m)$ and $p = P\,\{\,\mathit{name} = h, \mathit{meaning} = \mathcal{S}, \mathit{ftvs} = \overline{\alpha}^n, \mathit{fvs} = \overline{x}^m\,\}$ and $(\overline{\alpha}^n, \overline{x}^m) = \mathit{freeVars}\ \mathcal{S}$. then:*

$$\begin{aligned} \mathit{substPromises}((p, \overline{p}), \hat{e}) \underset{\sim}{\triangleleft\!\triangleright}\ & \mathbf{let}\ h\!:\!\hat{\tau}\ =\ \Lambda\overline{\alpha\!:\!\kappa}.\,\lambda \mathit{unit}\!:\!\mathit{unitTy}.\,\lambda\overline{x\!:\!\tau}.\,\mathit{delay}\ \mathcal{S} \\ & \mathbf{in}\ \mathit{substPromises}(\overline{p}, \hat{e}) \end{aligned}$$

Proof.

$$\begin{aligned} & \mathit{substPromises}((p, \overline{p}), \hat{e}) \\ = \ & \mathit{substPromises}(\overline{p}, \hat{e})[\Lambda\overline{\alpha\!:\!\kappa}.\,\lambda \mathit{unit}\!:\!\mathit{unitTy}.\,\lambda\overline{x\!:\!\tau}.\,{}^{\checkmark}\,\mathit{delay}\ \mathcal{S}/h] \\ = \ & \mathit{substPromises}(\overline{p}, \hat{e}'[\ldots, (h\ \overline{\alpha}\ (\mathit{unit}, \overline{x}))\theta_i/z_i, \ldots]) \\ & \quad [\Lambda\overline{\alpha\!:\!\kappa}.\,\lambda \mathit{unit}\!:\!\mathit{unitTy}.\,\lambda\overline{x\!:\!\tau}.\,{}^{\checkmark}\,\mathit{delay}\ \mathcal{S}/h] \\ \underset{\sim}{\triangleleft\!\triangleright}\ & \mathit{substPromises}(\overline{p}, \hat{e}'[\ldots, ({}^{\checkmark}\,\mathit{delay}\ \mathcal{S})\theta_i/z_i, \ldots]) \\ \underset{\sim}{\triangleleft\!\triangleright}\ & \ \mathbf{let}\ h\!:\!\hat{\tau}\ =\ \Lambda\overline{\alpha\!:\!\kappa}.\,\lambda \mathit{unit}\!:\!\mathit{unitTy}.\,\lambda\overline{x\!:\!\tau}.\,\mathit{delay}\ \mathcal{S} \\ & \ \mathbf{in}\ \mathit{substPromises}(\overline{p}, \hat{e}'[\ldots, ({}^{\checkmark}\,\mathit{delay}\ \mathcal{S})\theta_i/z_i, \ldots]) \\ \underset{\sim}{\triangleleft\!\triangleright}\ & \ \mathbf{let}\ h\!:\!\hat{\tau}\ =\ \Lambda\overline{\alpha\!:\!\kappa}.\,\lambda \mathit{unit}\!:\!\mathit{unitTy}.\,\lambda\overline{x\!:\!\tau}.\,\mathit{delay}\ \mathcal{S} \\ & \ \mathbf{in}\ \mathit{substPromises}(\overline{p}, \hat{e}'[\ldots, (h\ \overline{\alpha}\ (\mathit{unit}, \overline{x}))\theta_i/z_i, \ldots]) \\ = \ & \ \mathbf{let}\ h\!:\!\hat{\tau}\ =\ \Lambda\overline{\alpha\!:\!\kappa}.\,\lambda \mathit{unit}\!:\!\mathit{unitTy}.\,\lambda\overline{x\!:\!\tau}.\,\mathit{delay}\ \mathcal{S} \\ & \ \mathbf{in}\ \mathit{substPromises}(\overline{p}, \hat{e}) \end{aligned}$$

□

We are now in a position to prove the main theorem:

Theorem 3.8.3 (*sc* correctness)**.** *The sc function is correct if:*

$${}^{\checkmark}\,\mathit{delay}\ \mathcal{S} \underset{\sim}{\triangleright} \mathit{checkScpM}(\overline{p}, \mathit{sc}\ \mathit{hist}\ \mathcal{S})$$

Proof. There are three principal cases to consider:

1. The memoiser manages to tie back (the "tieback" case)

2. The memoiser doesn't tie back, and the termination test says *Stop* (the "stop" case)

3. The memoiser doesn't tie back, and the termination test says *Continue* (the "continue" case)

We consider each in order.

Tieback case In this case, we know that there is some $P\{\, name\ =\ h,\ meaning\ = \mathcal{S}',\ ftvs = \overline{\alpha},\ fvs = \overline{x}\,\} \in \overline{p}$ such that:

$$match\ \mathcal{S}'\ \mathcal{S}\ =\ Just\ \theta$$

Therefore, if the matcher is correct, by Definition 3.8.1 it must be the case that:

$$(delay\ \mathcal{S}')\theta\ \underset{\sim}{\lhd\rhd}\ delay\ \mathcal{S}$$

To prove the case, we need to show:

$$\begin{aligned}
{}^{\checkmark}delay\ \mathcal{S}\ &\underset{\sim}{\trianglerighteq}\ checkScpM(\overline{p}, return\ (h\ \overline{\alpha}\ (unit, \overline{x}))\theta)\\
&=\ substPromises(\overline{p}, (h\ \overline{\alpha}\ (unit, \overline{x}))\theta)\\
&=\ {}^{\checkmark}delay\ \mathcal{S}'\theta\\
&\underset{\sim}{\lhd\rhd}\ {}^{\checkmark}delay\ \mathcal{S}
\end{aligned}$$

Which follows trivially because $\underset{\sim}{\trianglerighteq}$ is reflexive.

Stop case Because we know that the recursion of sc is well-founded (see Section 3.7), we can combine induction with Lemma 3.8.2 to show that:

$$\begin{aligned}
delay\ \mathcal{S}\ &\underset{\sim}{\trianglerighteq}\ checkScpM((p, \overline{p}), split\ (sc\ hist)\ \mathcal{S})\\
&=\ checkScpM(\overline{p}, \mathbf{do}\ \ promise\ p\\
&\qquad\qquad\qquad\qquad\quad \hat{e}\ \leftarrow\ split\ (sc\ hist)\ \mathcal{S}\\
&\qquad\qquad\qquad\qquad\quad return\ \$\ substPromises(p, \hat{e}))\\
&=\ substPromises((p, \overline{p}), \hat{e})
\end{aligned}$$

Where $p = P\{\, name = h,\ meaning = \mathcal{S},\ ftvs = \overline{\alpha},\ fvs = \overline{x}\,\}$, $(\overline{\alpha}, \overline{x}) = freeVars\ \mathcal{S}$. So much for what we know inductively. To prove the case, we still need to show that:

$$\begin{aligned}
{}^{\checkmark}delay\ \mathcal{S}\ &\underset{\sim}{\trianglerighteq}\ checkScpM(\overline{p}, sc\ hist\ \mathcal{S})\\
&=\ checkScpM(\overline{p}, \mathbf{do}\ \ promise\ p\\
&\qquad\qquad\qquad\qquad\quad \hat{e}\ \leftarrow\ split\ (sc\ hist)\ \mathcal{S}\\
&\qquad\qquad\qquad\qquad\quad bind\ h\ \hat{\tau}\ (\Lambda\overline{\alpha{:}\kappa}.\,\lambda unit{:}unitTy.\,\lambda\overline{x{:}\tau}.\,\hat{e})\\
&\qquad\qquad\qquad\qquad\quad return\ (h\ \overline{\alpha}\ (unit, \overline{x}))
\end{aligned}$$

Where $\hat{\tau} = \forall\overline{\alpha{:}\kappa}.\ unitTy\ \rightarrow\ \overline{\tau} \rightarrow stateType\ \mathcal{S}$.

We can see that the $\hat{e}$ returned from the call to $split$ is the same in both this derivation and what we derived by induction, since they both originate from making the same recursive call in the same monadic environment of promises. Now, due to the $bind$ call we know that $runScpM$ will create the binding for h at the top level, and since let-floating of values is a cost-equivalence we can continue to simplify the required property as follows:

$$\begin{aligned}
{}^{\checkmark}delay\ \mathcal{S}\ &\underset{\sim}{\trianglerighteq}\ substPromises(\overline{p},\ \mathbf{let}\ h{:}\hat{\tau}\ =\ \Lambda\overline{\alpha{:}\kappa}.\,\lambda unit{:}unitTy.\,\lambda\overline{x{:}\tau}.\,\hat{e}\\
&\qquad\qquad\qquad\qquad\quad \mathbf{in}\ h\ \overline{\alpha}\ (unit, \overline{x}))\\
&=\ \mathbf{let}\ h{:}\hat{\tau}\ =\ \Lambda\overline{\alpha{:}\kappa}.\,\lambda unit{:}unitTy.\,\lambda\overline{x{:}\tau}.\,substPromises(\overline{p}, \hat{e})\\
&\quad\ \ \mathbf{in}\ h\ \overline{\alpha}\ (unit, \overline{x})
\end{aligned}$$

Because we know that:

$${}^{\checkmark}delay\ \mathcal{S} \underset{\sim}{\trianglerighteq} \mathbf{let}\ h{:}\hat{\tau}\ =\ \Lambda\overline{\alpha{:}\kappa}.\,\lambda unit{:}unitTy.\,\lambda\overline{x{:}\tau}.\,delay\ \mathcal{S}\ \mathbf{in}\ h\ \overline{\alpha}\ (unit, \overline{x})$$

Using the improvement theorem (Theorem C.4.2), we simplify our goal to:

$$\begin{aligned}
&\mathbf{let}\ h{:}\hat{\tau}\ =\ \Lambda\overline{\alpha{:}\kappa}.\,\lambda unit{:}unitTy.\,\lambda\overline{x{:}\tau}.\,delay\ \mathcal{S}\\
&\mathbf{in}\ delay\ \mathcal{S}\\
&\ \ \underset{\sim}{\trianglerighteq}\quad \mathbf{let}\ h{:}\hat{\tau}\ =\ \Lambda\overline{\alpha{:}\kappa}.\,\lambda unit{:}unitTy.\,\lambda\overline{x{:}\tau}.\,delay\ \mathcal{S}\\
&\qquad\ \ \mathbf{in}\ substPromises(\overline{p}, \hat{e})
\end{aligned}$$

Discarding the (dead) binding for h on the left, and applying Lemma 3.8.2 on the right, this simplifies to:

$$\mathit{delay}\ \mathcal{S} \quad \mathrel{\underset{\sim}{\triangleright}} \quad \mathit{substPromises}((p, \overline{p}), \hat{e})$$

This is exactly what we earlier showed to be true inductively.

Continue case This case follows similarly to the stop case, except that you need to use Lemma 3.8.1 to show that the use of *reduce* by the continue case is meaning-preserving up to delay.

□

From this theorem, we can immediately obtain correctness of *supercompile* as a corollary:

Corollary 3.8.4 (Correctness of *supercompile*)**.** *If e terminates in some context, then supercompile e terminates in that same context.*

Proof. From the correctness of *sc*, we know that:

$$\begin{array}{rcl}
{}^{\checkmark}\mathit{delay}\,\langle \epsilon \mid e \mid \epsilon \rangle & \mathrel{\underset{\sim}{\triangleright}} & \mathit{checkScpM}(\epsilon, \mathit{sc}\ \mathit{emptyHistory}\ \ \langle \epsilon \mid e \mid \epsilon \rangle) \\
 & = & \mathit{runScpM}\ \$\ \mathit{sc}\ \mathit{emptyHistory}\ \ \langle \epsilon \mid e \mid \epsilon \rangle \\
 & = & \mathit{supercompile}\ e
\end{array}$$

Since $\mathit{delay}_e(e) \mathrel{\underset{\sim}{\triangleright}} {}^{\checkmark}\mathit{delay}\,\langle \epsilon \mid e \mid \epsilon \rangle$, this is sufficient to show that supercompilation is an improvement up to delay:

$$\mathit{delay}_e(e) \mathrel{\underset{\sim}{\triangleright}} \mathit{supercompile}\ e$$

□

3.9 Supercompiling states vs. supercompiling terms

Our supercompiler uses normalised states (Section 2.2.2) ubiquitously throughout: *sc* supercompiles a normalised state, matching compares states and the operational semantics (obviously) operates on states. In this section we review the reasons as to why this is the most convenient definition.

Firstly, the fact that values of the *State* type are always normalised has several benefits:

- Because we know that sequences of normalising reductions are always of finite length, in order to ensure that the *reduce* function always terminates we only have to test the termination criteria when we perform a non-normalising reduction (see Section 3.3). So, normalisation allows us to avoid making some termination tests.

  In our experiments we found that this would reduce the number of termination tests we needed to make in *reduce* by a large factor of as much as 10 times. The exact amount by which this optimisation helps will of course depend on the program being supercompiled: for example, if you are supercompiling a tail recursive loop function which takes n parameters, then you would expect to make n times fewer termination tests in *reduce*.

  After some experience with our supercompiler we discovered that making termination tests infrequent is actually more than a mere optimisation. If we test for termination very frequently (say, after *any* reduction rule is used—not just after non-normalising rules), the successive states will be very similar; and the more similar they are, the greater the danger that the necessarily-conservative termination criterion (Section 3.2) will unnecessarily say *Stop*.

- Programs output by our supercompiler will never contain “obvious” undone **case** discrimination nor manifest β-redexes.

- Normalising states before matching makes matching succeed more often since trivial differences between *State*s cannot prevent tieback from occurring (Section 3.4). For example, early versions of the supercompiler that did not incorporate normalisation suffered from increased output program size because they did not determine that the following *State*s all meant the same thing:

$$\begin{array}{l}\langle \mathit{foldr} \mapsto \ldots \,|\, \mathit{foldr} \,|\, \bullet\ c, \bullet\ n \rangle \\ \langle \mathit{foldr} \mapsto \ldots \,|\, \mathit{foldr}\ c \,|\, \bullet\ n \rangle \\ \langle \mathit{foldr} \mapsto \ldots \,|\, \mathit{foldr}\ c\ n \,|\, \epsilon \rangle \\ \langle x \mapsto \mathit{True}, \mathit{foldr} \mapsto \ldots \,|\, \textbf{case}\ x\ \textbf{of}\ \mathit{True} \to \mathit{foldr}\ c\ n \,|\, \epsilon \rangle\end{array}$$

Other benefits accrue because we operate on *State*s rather than terms:

- It is natural to treat a state’s heap as an unordered map. Our implementation exploits this by providing a *match* function that is insensitive to the exact order of bindings in the heaps of the states it is to match. A contrasting approach would be to treat the heap as an ordered association list or sequence of enclosing **let** expressions, but this would not be as normalising because states such as $\langle x \mapsto \mathit{True}, y \mapsto \mathit{False} \,|\, (x, y) \,|\, \epsilon \rangle$ and $\langle y \mapsto \mathit{False}, x \mapsto \mathit{True} \,|\, (x, y) \,|\, \epsilon \rangle$ might not be considered the same for the purposes of *match*, losing some tieback opportunities.

- When splitting, it can become necessary to perform transformations such as the case-of-case transformation in order to fully optimise certain input terms such as **case** (**case** x **of** $\mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True}$) **of** $\mathit{False} \to \mathit{True}; \mathit{True} \to \mathit{False}$ (see Section 4.2). This family of “stack inlining” transformations are much easier to implement when we have an explicit stack available as one of the components of the state.

- Identifying the stack explicitly is convenient for our call-by-need most-specific-generalisation (Chapter 5) in several ways. We defer discussion of these reasons until Section 5.9.

Chapter 4

The call-by-need splitter

The purpose of the supercompiler's splitter is, when given a (normalised) state, to identify some new states that can be recursively supercompiled, supercompile them, and then reassemble to produce a term with the same meaning as the original state. Recall from Section 3.5 that the type signature of *split* is:

$$\begin{aligned} \mathit{split} :: \mathit{Monad}\ m \Rightarrow &(\mathit{State} \to m\ \mathit{Term}) \\ &\to \mathit{State} \to m\ \mathit{Term} \end{aligned}$$

The *opt* argument to a call *split opt* $\mathcal{S}$ is the function that will be used to perform the recursive supercompilation, while $\mathcal{S}$ is of course the state to be split.

Defining a *split* which achieves a good level of optimisation is straightforward for call-by-name and call-by-value languages. With call-by-need languages the issue of work duplication arises: we wish to propagate as much information as possible (to achieve optimisation) but do not want to risk duplicating work to do so. Achieving the right trade-off here complicates the definition of *split*. When considering a call-by-need language with full recursive **let** then the problem of defining a good *split* becomes harder again: this is the problem we will solve in this chapter. In Section 4.6 we will review the reasons why defining *split* is so particularly difficult in our setting compared to the alternatives.

4.1 The push-recurse framework

To aid exposition, we will think about splitting as a two stage process.

1. In the first stage, which we call the *pusher* (as it inlines and pushes bits of syntax into each other), the input state is transformed into a new "pushed" state where every immediate subterm has been replaced with a nested state. For example, an input state such as

   $$\langle a \mapsto f\ xs, b \mapsto g\ 100, f \mapsto g\ 10 \,|\, (a, b) \,|\, \epsilon \rangle$$

   might be transformed by the pusher into the pushed state

   $$\langle a \mapsto \langle f \mapsto g\ 10 \,|\, f\ xs \,|\, \epsilon \rangle, b \mapsto \langle \epsilon \,|\, g\ 100 \,|\, \epsilon \rangle \ \,|\, \ (a, b) \ \,|\, \ \epsilon \rangle$$

   This stage is by far the most complicated of the two and will be covered in detail in Section 4.3.

2. The second stage, after extracting a pushed state, is to recursively supercompile every nested state in the pushed state (such as $\langle \epsilon \,|\, g\ 100 \,|\, \epsilon \rangle$) and use the results of

that recursion to reassemble the final result term. So, after driving and reassembling the example above might turn into the following final term returned by *split*:

$$\begin{array}{l}\textbf{let}\ a = \mathit{h5}\ g\ \mathit{xs} \\ \quad\ \ b = \mathit{h6}\ g \\ \textbf{in}\ (a, b)\end{array}$$

We will cover this stage in detail in Section 4.4.

Remembering that the states to be split are normalised, and so they may have only a value or variable in the focus, the grammar of pushed states is as you would expect (in this section we will elide all tags for clarity):

Pushed States $\mathring{\mathcal{S}} ::= \left\langle \mathring{H} \,\middle|\, \mathring{e} \,\middle|\, \mathring{K} \right\rangle$

Pushed Heaps

$$\begin{array}{rcll} \mathring{H} & ::= & \overline{\mathring{h}} & \text{Heap} \\ \mathring{h} & ::= & x{:}\tau \mapsto \mathring{e} & \text{Heap binding} \end{array}$$

Pushed Terms

$$\begin{array}{rcll} \mathring{e} & ::= & x & \text{Term variable} \\ & | & \lambda x{:}\tau.\,\mathcal{S} & \text{Term abstraction} \\ & | & \mathbf{C}\ \overline{\tau}\ \overline{x} & \text{Algebraic data} \\ & | & \Lambda\alpha{:}\kappa.\,\mathcal{S} & \text{Type abstraction} \\ & | & \mathcal{S} & \text{Raw state} \end{array}$$

Pushed Stacks

$$\begin{array}{rcll} \mathring{K} & ::= & \overline{\mathring{\kappa}} & \text{Stack} \\ \mathring{\kappa} & ::= & \textbf{update}\ x{:}\tau & \text{Update frame} \\ & | & \bullet\ x & \text{Supply argument to function value} \\ & | & \bullet\ \tau & \text{Instantiate value} \\ & | & \textbf{case} \bullet \textbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau}\ \to\ \mathcal{S}} & \text{Scrutinise value} \end{array}$$

Given this framework, we can write *split* as follows:

$$\begin{array}{l} \mathit{split} :: \mathit{Monad}\ m \Rightarrow (\mathit{State} \to m\ \mathit{Term}) \\ \qquad\qquad\qquad\qquad \to \mathit{State} \to m\ \mathit{Term} \\ \mathit{split}\ \mathit{opt}\ \mathcal{S}\ = \ \mathit{traverse}\mathring{\mathit{State}}\ \mathit{opt}\ \mathring{\mathcal{S}} \\ \quad \textbf{where}\ \mathring{\mathcal{S}}\ =\ \mathit{push}\ \mathcal{S} \end{array}$$

where we use two functions:

$$\begin{array}{l} \mathit{push} :: \mathit{State} \to \mathring{\mathit{State}} \\ \mathit{traverse}\mathring{\mathit{State}} :: \mathit{Monad}\ m \Rightarrow (\mathit{State} \to m\ \mathit{Term}) \\ \qquad\qquad\qquad\qquad\qquad \to \mathring{\mathit{State}} \to m\ \mathit{Term} \end{array}$$

The call $\mathit{traverse}\mathring{\mathit{State}}\ \mathit{opt}\ \mathring{\mathcal{S}}$ is intended to recursively replace all nested states $\mathcal{S}$ in the argument $\mathring{\mathcal{S}}$ with $\mathit{opt}\ \mathcal{S}$ and *rebuild* the result into a term. So for example,

$$\mathit{traverse}\mathring{\mathit{State}}\ (\mathit{return}.\mathit{rebuild})\ \langle a \mapsto \langle f \mapsto g\ 10 \,|\, f\ \mathit{xs} \,|\, \epsilon\rangle, b \mapsto \langle \epsilon \,|\, g\ 100 \,|\, \epsilon\rangle\ \mid\ (a, b)\ \mid\ \epsilon\rangle$$

gives the term:

$$\begin{array}{l}\textbf{let}\ a = \textbf{let}\ f = g\ 10\ \textbf{in}\ f\ \mathit{xs} \\ \quad\ \ b = g\ 100 \\ \textbf{in}\ (a, b)\end{array}$$

Our real *push* function will be defined in Section 4.3. We begin with an extensive introduction to how an ideal *push* function should behave.

4.2 Avoiding information loss in the splitter

Many simple definitions of the splitter are "correct" by the definition of Section 3.8 and so are safe to use in supercompilation, but fail to achieve certain kinds of specialisation.

The rebuild splitter The simplest possible splitter (albeit one which does not fit within the push-recurse framework) is defined by:

$$\mathit{split}\ \mathit{opt}\ \mathcal{S} = \mathit{return}\ (\mathit{rebuild}\ \mathcal{S})$$

This splitter is non-optimal because it fails to achieve any optimisation of any subcomponents of the input state. A supercompiler with a splitter like this would only achieve reduction at the very top level of the input state, and so it would be able to optimise the input term $(\lambda y.\, y)\ a$ to a, but would not be able to optimise $\lambda a.\, (\lambda y.\, y)\ a$ to $\lambda a.\, a$ or $\mathbf{case}\ p\ \mathbf{of}\ (a, b) \to (\lambda y.\, y)\ a$ to $\mathbf{case}\ p\ \mathbf{of}\ (a, b) \to a$.

The subterm splitter A slightly better splitter is the splitter which recursively supercompiles sub*terms* of the input state. This can be defined in the push-recurse framework as a splitter with the following *push* function:

$$\begin{aligned}
\mathit{push}\,\langle H \,|\, e \,|\, K\rangle_{\Sigma|\Gamma} &= \langle \mathit{push}_H(\Sigma, \Gamma'|H) \,|\, \mathit{push}_{\bullet}(\Sigma, \Gamma'|e) \,|\, \mathit{push}_K(\Sigma, \Gamma'|K)\rangle \\
&\qquad \mathbf{where}\ \ \Gamma' = \Gamma \ \ ,\ \ \{x{:}\tau \mid \mathbf{update}\ x{:}\tau \in K\} \\
&\qquad\qquad\qquad\qquad\ ,\ \ \{x{:}\tau \mid x{:}\tau \mapsto e \in H\}
\end{aligned}$$

$$\begin{aligned}
\mathit{push}_H(\Sigma, \Gamma|\epsilon) &= \epsilon \\
\mathit{push}_H(\Sigma, \Gamma|x{:}\tau \mapsto e, H) &= x{:}\tau \mapsto \mathit{push}_{\bullet}(\Sigma, \Gamma|e), \mathit{push}_H(\Sigma, \Gamma|H)
\end{aligned}$$

$$\begin{aligned}
\mathit{push}_{\bullet}(\Sigma, \Gamma|x) &= x \\
\mathit{push}_{\bullet}(\Sigma, \Gamma|\lambda x{:}\tau.\, e) &= \lambda x{:}\tau.\ \langle \epsilon \,|\, e \,|\, \epsilon\rangle_{\Sigma|\Gamma, x:\tau} \\
\mathit{push}_{\bullet}(\Sigma, \Gamma|\Lambda \alpha{:}\kappa.\, e) &= \Lambda \alpha{:}\kappa.\ \langle \epsilon \,|\, e \,|\, \epsilon\rangle_{\Sigma, \alpha:\kappa|\Gamma} \\
\mathit{push}_{\bullet}(\Sigma, \Gamma|\mathbf{C}\ \overline{\tau}\ \overline{x}) &= \mathbf{C}\ \overline{\tau}\ \overline{x} \\
\mathit{push}_{\bullet}(\Sigma, \Gamma|e) &= \langle \epsilon \,|\, e \,|\, \epsilon\rangle_{\Sigma|\Gamma}
\end{aligned}$$

$$\begin{aligned}
\mathit{push}_K(\Sigma, \Gamma|\epsilon) &= \epsilon \\
\mathit{push}_K(\Sigma, \Gamma|\kappa, K) &= \mathit{push}_{\kappa}(\Sigma, \Gamma|\kappa), \mathit{push}_K(\Sigma, \Gamma|K)
\end{aligned}$$

$$\begin{aligned}
\mathit{push}_{\kappa}(\Sigma, \Gamma|\mathbf{update}\ x{:}\tau) &= \mathbf{update}\ x{:}\tau \\
\mathit{push}_{\kappa}(\Sigma, \Gamma|\bullet\ x) &= \bullet\ x \\
\mathit{push}_{\kappa}(\Sigma, \Gamma|\bullet\ \tau) &= \bullet\ \tau \\
\mathit{push}_{\kappa}(\Sigma, \Gamma|\mathbf{case}\bullet\mathbf{of}\,\overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau} \ \to\ e}) &= \mathbf{case}\bullet\mathbf{of}\,\overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau} \ \to\ \langle \epsilon \,|\, e \,|\, \epsilon\rangle_{\Sigma|\Gamma}}
\end{aligned}$$

A supercompiler that uses a push-recurse *split* defined with this *push* function is able to optimise $\lambda a.\, (\lambda y.\, y)\ a$ to $\lambda a.\, a$ or $\mathbf{case}\ p\ \mathbf{of}\ (a, b) \to (\lambda y.\, y)\ a$ to $\mathbf{case}\ p\ \mathbf{of}\ (a, b) \to a$, unlike the simple *rebuild* splitter above. However, this *split* still leaves something to be desired. For example, it will split the state:

$$
\begin{aligned}
&\textbf{let } a = id\ y\\
&\qquad id = \lambda x.\, x\\
&\textbf{in } \mathit{Just}\ a
\end{aligned}
$$

as follows:

$$\langle a \mapsto \langle \epsilon \,|\, id\ y \,|\, \epsilon \rangle, id \mapsto \lambda x.\ \langle \epsilon \mid x \mid \epsilon \rangle \ \ |\ \ \mathit{Just}\ a \ \ |\ \ \epsilon \rangle$$

This splitting would mean that our supercompiler couldn't reduce the call *id y* at compile time, since the definition of the *id* function is not present in the appropriate nested heap.

Heap pushing in the subterm splitter To fix the immediate problem, you might modify the *push* function so that all heap bindings in the heap passed to *push* are copied to all of the nested heaps in the pushed state returned by *push* (if this happens, we say that those heap bindings have been "pushed down"). So for our example above, the pushed state would be:

$$\langle a \mapsto \langle a \mapsto id\ y, id \mapsto \lambda x.\, x \,|\, id\ y \,|\, \epsilon \rangle, id \mapsto \lambda x.\ \langle a \mapsto id\ y, id \mapsto \lambda x.\, x \,|\, x \,|\, \epsilon \rangle \ \ |\ \ \mathit{Just}\ a \ \ |\ \ \epsilon \rangle$$

This does allow the supercompiler to statically reduce the *id* application in the example. However, such a splitter is unsuitable for our purposes as it may duplicate work, such as when asked to split this state:

$$
\begin{aligned}
&\textbf{let } n = \mathit{fib}\ 100\\
&\qquad b = n + 1\\
&\qquad c = n + 2\\
&\textbf{in } (b, c)
\end{aligned}
$$

In which case the pushed state (ignoring dead heap bindings) would be something like:

$$\langle n \mapsto \langle \epsilon \,|\, \mathit{fib}\ 100 \,|\, \epsilon \rangle, b \mapsto \langle n \mapsto \mathit{fib}\ 100 \,|\, n + 1 \,|\, \epsilon \rangle, c \mapsto \langle n \mapsto \mathit{fib}\ 100 \,|\, n + 2 \,|\, \epsilon \rangle \ \ |\ \ (b, c) \ \ |\ \ \epsilon \rangle$$

One solution to this problem of potential work duplication would be to only push down that portion of the input heap which is manifestly *cheap* (i.e. a value or variable). This heuristic is sufficient to statically reduce the *id* application in our example as it leads to the following pushed state (again ignoring dead heap bindings):

$$\langle a \mapsto \langle id \mapsto \lambda x.\, x \,|\, id\ y \,|\, \epsilon \rangle, id \mapsto \lambda x.\ \langle \epsilon \mid x \mid \epsilon \rangle \ \ |\ \ \mathit{Just}\ a \ \ |\ \ \epsilon \rangle$$

However, this simple heuristic fails to achieve as much optimisation as we would like with examples such as the following:

$$
\begin{aligned}
&\textbf{let } \mathit{map} = \ldots\\
&\qquad \mathit{ys} = \mathit{map}\ f\ \mathit{zs}\\
&\qquad \mathit{xs} = \mathit{map}\ g\ \mathit{ys}\\
&\textbf{in } \mathit{Just}\ \mathit{xs}
\end{aligned}
$$

which would be split as follows:

$$\langle \mathit{ys} \mapsto \langle \mathit{map} \mapsto \ldots \,|\, \mathit{map}\ f\ \mathit{zs} \,|\, \epsilon \rangle, \mathit{xs} \mapsto \langle \mathit{map} \mapsto \ldots \,|\, \mathit{map}\ g\ \mathit{ys} \,|\, \epsilon \rangle \ \ |\ \ \mathit{Just}\ \mathit{xs} \ \ |\ \ \epsilon \rangle$$

This splitting means we cannot achieve deforestation of the two nested *map* calls because they are independently supercompiled. We would prefer a splitting where *ys* is instead pushed down into the heap binding for *xs*:

$$\langle \mathit{xs} \mapsto \langle \mathit{map} \mapsto \ldots, \mathit{ys} \mapsto \mathit{map}\ f\ \mathit{zs} \,|\, \mathit{map}\ g\ \mathit{ys} \,|\, \epsilon \rangle \ \ |\ \ \mathit{Just}\ \mathit{xs} \ \ |\ \ \epsilon \rangle$$

In general we want to push down as much syntax as possible, so that our recursive invocations of the supercompiler have as much information as possible to work with. With this in mind, our real *push* function will in fact push down heap bindings if they are manifestly *either* cheap *or* used linearly in a certain sense (as in this *map*-composition example). In either case, there is no risk of work duplication.

Stack pushing in the subterm splitter We have concentrated a lot on pushing down heap bindings, but we have not yet said much about how the stack component of the input state should be handled. Returning to our original simple subterm splitter, we find that we would split the state

$$\langle \epsilon \mid \mathit{unk} \mid \textbf{case} \bullet \textbf{of } \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True}, \textbf{case} \bullet \textbf{of } \mathit{True} \to x; \mathit{False} \to y \rangle$$

as follows:

$$\left\langle \epsilon \middle| \mathit{unk} \middle| \begin{array}{r} \textbf{case} \bullet \textbf{of } \mathit{True} \to \langle \epsilon \mid \mathit{False} \mid \epsilon \rangle \\ \mathit{False} \to \langle \epsilon \mid \mathit{True} \mid \epsilon \rangle \end{array}, \begin{array}{r} \textbf{case} \bullet \textbf{of } \mathit{True} \to \langle \epsilon \mid x \mid \epsilon \rangle \\ \mathit{False} \to \langle \epsilon \mid y \mid \epsilon \rangle \end{array} \right\rangle$$

We would really prefer the following alternative splitting where the tail of the stack is pushed into the two branches of the first **case**, which allows the intermediate scrutinisation of a *Bool* to be performed at compile time:

$$\left\langle \epsilon \middle| \mathit{unk} \middle| \begin{array}{r} \textbf{case} \bullet \textbf{of } \mathit{True} \to \langle \epsilon \mid \mathit{False} \mid \textbf{case} \bullet \textbf{of } \mathit{True} \to x; \mathit{False} \to y \rangle \\ \mathit{False} \to \langle \epsilon \mid \mathit{True} \mid \textbf{case} \bullet \textbf{of } \mathit{True} \to x; \mathit{False} \to y \rangle \end{array} \right\rangle$$

Our real *push* function will indeed produce this pushed state, in line with the general principle that in order to achieve maximum specialisation we wish to push down as much syntax as possible without duplicating work.

Positive information The last important point to bear in mind when defining *push* is that we gain information about a free variable when it is scrutinised by a residual **case**. This extra "positive information" can be represented by adding additional heap bindings to the nested states we are to supercompile. Thus, given the state

$$\langle \mathit{not} \mapsto \ldots, \mathit{xor} \mapsto \ldots \mid x \mid \textbf{case} \bullet \textbf{of } \mathit{True} \to \mathit{not}\ x; \mathit{False} \to \mathit{xor}\ x\ \mathit{True} \rangle$$

we should split as follows:

$$\left\langle \epsilon \middle| x \middle| \begin{array}{r} \textbf{case} \bullet \textbf{of } \mathit{True} \to \langle x \mapsto \mathit{True}, \mathit{not} \mapsto \ldots \mid \mathit{not}\ x \mid \epsilon \rangle \phantom{\ \mathit{True}} \\ \mathit{False} \to \langle x \mapsto \mathit{False}, \mathit{xor} \mapsto \ldots \mid \mathit{xor}\ x\ \mathit{True} \mid \epsilon \rangle \end{array} \right\rangle$$

By making use of the values of x we have learnt from the **case** alternatives, we are able to statically reduce the *not* and *xor* operations in each branch.

In the next section we describe in detail our *push* function that constructs pushed states. In Section 4.4, we discuss issues relating to the process whereby the pushed states are recursively driven and then "pasted" back together.

4.3 Our pushing algorithm

The *push* function, of type $\mathit{State} \to \overset{\circ}{\mathit{State}}$, can be described at a high level as consisting of a number of stages:

1. A "sharing-graph" is extracted from the input state, where:

    - Each graph node c corresponds to either a single stack frame in the input state (denoted by the 0-based index of that stack frame, n), a single heap binding in the input state (denoted by the corresponding bound variable x), or the focus of the term (denoted by $\bullet$)

$$\begin{array}{rcll} c & ::= & \bullet & \text{Focus of input} \\ & | & x & \text{Heap binding for } x \text{ in input} \\ & | & n & \text{Stack frame at index } n \text{ in input} \\ o & ::= & ✓ & \text{Used in at most one context} \\ & | & ✗ & \text{Used by multiple contexts} \end{array}$$

Figure 4.1: Grammar of sharing-graph nodes and edges

- A (directed) edge from one node to another indicates that the former makes reference to the latter. These edges make explicit the sharing relationship between the parts of the input state. These edges will be labelled, where the labels o are either ✓ or ✗: we will explain the meaning of these labels in Section 4.3.1.

For reference, the grammar of nodes and edges is given formally in Figure 4.1.

2. This sharing-graph is solved to extract the set of *marked* nodes M. The intention is that any node which is marked will have its corresponding syntax pushed down into the state subcomponents that we will recursively supercompile, and furthermore it is usually true that any node which is marked will not be residualised. In order to propagate as much information as possible to our children, we desire that M should be as large as possible. At the same time, we must avoid marking nodes in such a way that pushing down will cause work duplication.

   Striking the right balance between work duplication and information propagation is hard, and is why we need to explicitly track sharing relationships via the sharing graph data structure.

3. The final pushed state has one nested state for every direct recursive invocation of the supercompiler we intend to make. Each of these nested states will contain heaps and stacks which consist *only* of those portions of the input heap and stack which have been marked by the previous step.

We discuss each of these stages in detail below. For clarity, all of our definitions assume that terms are untagged, though they extend in an obvious manner to tagged terms.

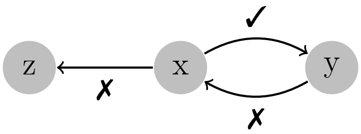
This method of splitting a state may appear overelaborate at first blush. However, the algorithm presented in this chapter is the last in a long line of our attempts to solve the splitting problem, and as such is the result of considerable experimentation with different approaches of varying complexities. We found that in practice it was very easy to define an algorithm which worked acceptably well for 90% of cases, but devising an approach that both avoided work duplication and allowed information propagation in all programs of interest was very difficult, particularly where those programs involved **letrec**.

In particular, we found that the identification of sharing graphs and their solution procedure as a separable problem was a great help in the process of devising a correct algorithm.

4.3.1 Sharing-graphs

In the previous section, we described sharing graphs as being (possibly-cyclic) directed graphs with labelled edges. The grammar of nodes c and edge labels o was given in Figure 4.1.

In what follows, we will represent sharing-graphs mathematically as a doubly nested mapping. To be precise, the graph will be a mapping from the nodes of the graph to maps from *successors* of that node to the label. For example, the object $\{x \mapsto \{y \mapsto \checkmark, z \mapsto \text{✗}\}, y \mapsto \{x \mapsto \text{✗}\}, z \mapsto \varnothing\}$ represents the following sharing-graph:



Note that this representation enforces the invariant that there is at most one edge between any pair of nodes. It also allows for graphs containing edges to nonexistent nodes, such as the node y in $\{x \mapsto \{y \mapsto \checkmark\}\}$. This is a useful property of the representation when we are compositionally forming graphs and do not yet know what nodes will be present in the complete graph. In the case where a "dead" edge like this remains even once the complete sharing-graph has been constructed (which can happen), it will be harmless to our splitting algorithms—they simply treat such edges as nonexistent.

Sharing-graph construction We have so far deferred discussion of the meaning of edge labels o. A vague but perhaps helpful explanation is that if we have an edge $c_0 \overset{o}{\to} c_1$ then o is ✓ if and only if we are able and motivated to push the syntax corresponding to c_1 into a residualised version of the syntax corresponding to c_0. If this is not the case then the edge is labelled instead with ✗.

- To be "able" to do the pushing means that if c_0 is the *sole reference to* c_1 then by pushing c_1 into a residualised version of c_0 we would not lose work sharing.
- To be "motivated" to push down means that pushing the syntax for c_1 down has some chance of achieving useful simplification from recursive supercompiler invocations.

Note that both of these tests are done only with information local to c_0: in order to decide whether we are able and motivated we do not need to do a complex work duplication check involving other potential users of c_1. This means that the sharing graph has the property that the labelled edges outgoing from a node c can be computed entirely from the syntax corresponding to c, which in turn allows our implementation to construct the sharing-graph compositionally from the different components of the state.

Actual consideration of the full work duplication consequences of making nodes available for pushing is left entirely to the next stage of the pushing algorithm: the sharing-graph solver (Section 4.3.2).

The principles embodied by the "able and motivated" test should become clearer later when we consider a number of examples of states and their corresponding sharing-graphs.

Sharing-graphs are constructed by the *graph* family of functions given in Figure 4.2. The graph is constructed compositionally, with roughly one *graph* function for each syntactic entity (heap binding, stack frame, focus) in a state. A rough overview is that:

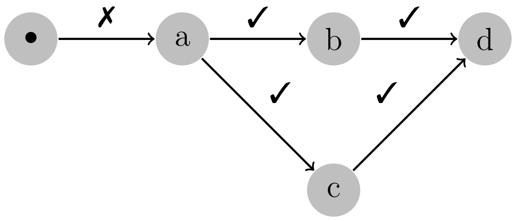
- The graph node for each syntactic entity has outgoing edges to the heap binding nodes for each of the syntactic entity's free variables.
- The graph node corresponding to the focus has an outgoing edge to the first stack frame's node.
- The graph nodes corresponding to each stack frame each have one outgoing edge to the subsequent stack frame's node.

$$
\begin{aligned}
\mathit{graph}_{\bullet}(e) &= \{\bullet \mapsto (\{0 \mapsto \text{✗}\} \cup \widehat{\mathit{graph}}_{\bullet}(e))\} \\
\widehat{\mathit{graph}}_{\bullet}(x) &= \{x \mapsto \text{✗}\} \\
\widehat{\mathit{graph}}_{\bullet}(\lambda x{:}\tau.\, e) &= \widehat{\mathit{graph}}_{e}(e)(\text{✗}) \setminus \{x\} \\
\widehat{\mathit{graph}}_{\bullet}(\Lambda \alpha{:}\kappa.\, e) &= \widehat{\mathit{graph}}_{e}(e)(\checkmark) \\
\widehat{\mathit{graph}}_{\bullet}(\mathbf{C}\ \overline{\tau}\ \overline{x}) &= \{x \mapsto \text{✗} \mid x \in \overline{x}\} \\
\widehat{\mathit{graph}}_{\bullet}(e) &= \widehat{\mathit{graph}}_{e}(e)(\checkmark) \\
\\
\mathit{graph}_{K}(K) &= \bigcup\{\mathit{graph}_{\kappa}(\kappa)(i) \mid K[i] = \kappa\} \\
\mathit{graph}_{\kappa}(\textbf{update}\ x{:}\tau)(i) &= \{i \mapsto \{i+1 \mapsto \text{✗}\}, x \mapsto \{i \mapsto \text{✗}\}\} \\
\mathit{graph}_{\kappa}(\bullet\ x)(i) &= \{i \mapsto \{i+1 \mapsto \text{✗}, x \mapsto \text{✗}\}\} \\
\mathit{graph}_{\kappa}(\bullet\ \tau)(i) &= \{i \mapsto \{i+1 \mapsto \text{✗}\}\} \\
\mathit{graph}_{\kappa}(\textbf{case} \bullet \textbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau} \to e})(i) &= \{i \mapsto (\{i+1 \mapsto \checkmark\} \cup \{y \mapsto \checkmark \mid y \in \bigcup \overline{\mathrm{fvs}\,(e) \setminus \overline{x}}\})\} \\
\\
\mathit{graph}_{H}(\epsilon) &= \varnothing \\
\mathit{graph}_{H}(x{:}\tau \mapsto e, H) &= \{x \mapsto \widehat{\mathit{graph}}_{\bullet}(e)\} \cup \mathit{graph}_{H}(H) \\
\\
\widehat{\mathit{graph}}_{e}(e)(c) &= \{y \mapsto c \mid y \in \mathrm{fvs}\,(e)\}
\end{aligned}
$$

Figure 4.2: Construction of sharing-graphs

Example 1: a simple heap

$$\langle a \mapsto f\ b\ c, b \mapsto g\ d, c \mapsto h\ d, d \mapsto \mathit{fib}\ x \mid \mathit{Just}\ a \mid \epsilon\rangle$$



All of the edges in this graph which originate from heap binding nodes are labelled with ✓. This reflects the fact that:

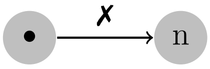
- We are able to float a **let**-binding for x into a **let**-binding which references x without duplicating the work embodied by x, (assuming that x is not in fact referenced anywhere else).
- Our sharing-graph construction algorithm is designed with the assumption that it could potentially improve optimisation if we had access to the definitions of the free variables of a heap binding when supercompiling that binding (such as if we were pushing $\mathit{xs} \mapsto \mathit{map}\ f\ \mathit{ys}$ into another binding $\mathit{ys} \mapsto \mathit{map}\ g\ \mathit{xs}$). Therefore, we are motivated to do such pushing.

  As a human we might look at this example state and see that since f g and h are all unknown functions, nothing can come of pushing down the definitions of b, c or d. However, remember that the construction of sharing-graphs makes use of only information *local* to nodes, and by looking at a binding like $a \mapsto f\ b\ c$ in isolation we cannot say whether the definition of f is available or not, so we must optimistically say that any of the free variables f, b and c are all motivational.

The final edge $\bullet \overset{\text{✗}}{\rightarrow} a$ indicates that although pushing a into the argument of *Just* wouldn't duplicate any work, the data constructor context is certainly boring and so we have no motivation to do so.

Example 2: a work-duplicating context

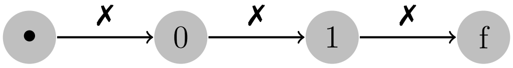
$$\langle n \mapsto \mathit{fib}\ x \,|\, \lambda y.\, n + y \,|\, \epsilon \rangle$$



The solitary edge in this graph is marked ✗ to reflect the fact that pushing the definition of n beneath the λ would cause work duplication, and so we are not able to push it down.

Example 3: a simple stack

$$\langle f \mapsto \lambda x.\, \textbf{case}\ x\ \textbf{of}\ \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True} \,|\, g \,|\, \textbf{0:}\bullet\ \ \mathit{xs}, \textbf{1:}\bullet\ \ f \rangle$$



This example is the first to include a stack frame. In order to aid exposition we make the index i of each stack frame κ in the state explicit using the notation $\mathbf{i{:}}\kappa$.

When the sharing graph is constructed for a non-empty stack, the focus node • has an edge to the first stack frame node 0, and then each stack frame node has an edge to the node of the following frame, such as the edge $0 \overset{\text{✗}}{\rightarrow} 1$ in this graph. When defining sharing-graphs, we said that there would be an edge between nodes if the predecessor node made use of the successor. When it comes to stack frame references, this use relationship is less explicit than with heap binding references, but it is still present: it is the use relationship you would see if you were to transform the state into continuation-passing style.

In this graph, all edges have been annotated with ✗:

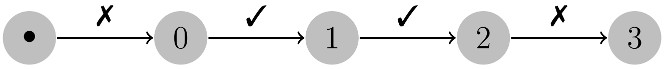
- For the edge $\bullet \overset{\text{✗}}{\rightarrow} 0$, we choose ✗ since we have (usually) already attempted to reduce the input to the splitter, and so there are probably no remaining opportunities for the focus to react with the first stack frame 0: as a result we are unmotivated to push it into a residualised version of the focus.

- For the edge $0 \overset{\text{✗}}{\rightarrow} 1$, we choose ✗ for similar reasons: there is no motivation to push the second stack frame into a residualised version of the first stack frame. Indeed, in this case it is unclear as to exactly what pushing a stack frame into the residualised stack from $\bullet\ \mathit{xs}$ would mean given the restrictive grammar of stack frames. (We will shortly see an example where the act of pushing a stack frame within another does make sense.)

- Finally, for the edge $1 \overset{\text{✗}}{\rightarrow} f$ we choose ✗, as a residualised version of the second stack frame $\bullet\ f$ is an entirely boring context into which to push the definition of f: there is no extra context you can glean from being the argument of an unknown function application.

Example 4: scrutinisation

$$\left\langle \epsilon \,\middle|\, \mathit{unk} \,\middle|\, \begin{array}{l}\textbf{0:case} \bullet \textbf{of}\ \mathit{True} \to \mathit{False} \\ \qquad\qquad\ \ \mathit{False} \to \mathit{True}\end{array}, \begin{array}{l}\textbf{1:case} \bullet \textbf{of}\ \mathit{True} \to f \\ \qquad\qquad\ \ \mathit{False} \to g\end{array}, \textbf{2:}\bullet\ \ x, \textbf{3:}\bullet\ \ y \right\rangle$$

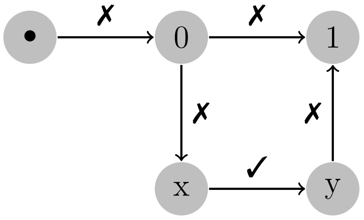


This example demonstrates a case where edges pointing to stack frames may usefully be marked ✓ rather than ✗. The edges $0 \overset{\checkmark}{\to} 1$ and $1 \overset{\checkmark}{\to} 2$ indicate that we are motivated to push the tail of the stack into either of the two **case** frames, since we might potentially be able to interact a data constructor within a **case** branch with the stack tail. Furthermore, doing so would not be work duplicating.

Just like our previous example, the edge $\bullet \overset{\text{✗}}{\to} 0$ indicates that we should not attempt to absorb the initial stack frame into the focus: we would not be able to make progress by doing so since that stack frame would only be able to interact with the same in-focus information that we have already simplified and normalised it with regards to. A similar argument applies for the edge $2 \overset{\text{✗}}{\to} 3$: the supercompiler has already failed to discover the result of the application with argument x, and so it is unmotivated to try to absorb the application with argument y because the function we are calling is unknown.

Example 5: update frames

$$\langle x \mapsto \mathit{fib}\ y \mid f \mid \mathbf{0{:}}\,\bullet\ \ x, \mathbf{1{:}update}\ y \rangle$$



Update frames are unique amongst the stack frames in that the graph generated for them includes nodes not only for the frame itself but also for the variable they bind, such as the edge $y \overset{\text{✗}}{\to} 1$ in the example. The purpose of these edges is to ensure that we avoid pushing down any update frame that binds a variable that is used elsewhere.

Because these edges are essentially "tricks" meant to ensure that references to variables bound by the stack transitively reference the corresponding stack frame in the sharing-graph, the annotation on the edges $y \overset{\text{✗}}{\to} 1$ is actually irrelevant to what follows: we simply make the arbitrary choice of ✗.

Without these edges, we would *always* push all trailing stack frames into the first **case** frame on the stack, thus *erroneously* turning

$$\langle y \mapsto \mathit{fib}\ x \mid f \mid \bullet\ y, \mathbf{case} \bullet \mathbf{of}\ \mathit{True} \to e_1; \mathit{False} \to e_2, \mathbf{update}\ x \rangle$$

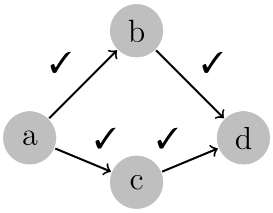
into a pushed state of the form:

$$\langle y \mapsto \langle \epsilon \mid \mathit{fib}\ x \mid \epsilon \rangle \mid f \mid \bullet\ y, \mathbf{case} \bullet \mathbf{of}\ \mathit{True} \to \langle \epsilon \mid e_1 \mid \mathbf{update}\ x \rangle; \mathit{False} \to \langle \epsilon \mid e_2 \mid \mathbf{update}\ x \rangle \rangle$$

4.3.2 Solving sharing-graphs

Once we have constructed the sharing-graph for a state, we need to solve it to determine a set M of marked nodes suitable for pushing down that is as large as possible while avoiding duplicating work through that pushing.

Solving sharing-graphs is not entirely straightforward, because deciding if marking a node may cause a loss of work sharing cannot be done by a test which only has access to information local to a node such as the number of incident edges. For example, with the following sharing graph:

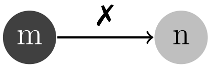


A naive syntactic test that looked at incident incoming edges would say that d could not be marked as it is used by two other nodes. However, in fact it is acceptable to mark d since both b and c can be marked *and* will be pushed into the common residualised (unmarked) node a.

The acceptability of marking d depends delicately on non-local information. For example, if the edge between a and c had been $a \overset{✗}{\rightarrow} c$ instead then c could not have been marked, which would have led to d not being markable (pushing d down would copy it into both the c and a contexts, duplicating work in the process). Likewise, if the edge $a \overset{✓}{\rightarrow} c$ had been replaced with an edge $e \overset{✓}{\rightarrow} c$ with e another unmarked node, then we would not be able to mark d even though it would be permissible to mark both of its predecessors b and c (pushing d down would copy it into both the a and e contexts, again duplicating work).

The situation becomes even more complicated when you consider that graphs can have arbitrary cyclic structure. For this reason we will spend this section giving the matter of graph marking a comprehensive treatment.

We will often wish to draw graphs which have corresponding marked-node sets. For such graphs, marked nodes will be drawn with a darker background. For example, the node m below is marked but n is not:



In our description of the marking process, the first thing we can do is give a property of marked sets which, if it holds for a set, means that constructing a pushed state with that set will never cause work duplication. We call this property admissability. Informally, it is only admissable for a node to be marked if either that node is cheap (i.e. it corresponds to a heap binding for a cheap term), or if marking it would result in it being pushed into a *unique*, non-work-duplicating, context.

Our notion of term cheapness is captured by a predicate which identifies terms which can certainly be freely duplicated without risking work duplication:

$$cheap(e) = e \equiv x \vee e \equiv v$$

Before we formally define admissability, we need to define the notion of a *pushing path*.

Definition 4.3.1 (Pushing path)**.** For a set of marked nodes M, a pushing path $c \underset{M}{\overset{o}{\twoheadrightarrow}} c'$ is a path in a sharing graph of the form:

$$c \overset{o}{\rightarrow} c_0 \overset{o_0}{\rightarrow} \ldots c_i \overset{o_i}{\rightarrow} \ldots \overset{o_n}{\rightarrow} c'$$

Such that:

- The path begins with the unmarked node $c \notin M$
- It travels through zero or more nodes c_i all of which are marked: $c_i \in M$

- The path ends at the marked node $c' \in M$

The relevance of this definition is that if there is a pushing path from $c \overset{o}{\underset{M}{\twoheadrightarrow}} c'$ then the pushed state constructed from this marking will have pushed the syntax corresponding to c' into the residual syntax corresponding to c.

We will sometimes omit the marking M from a pushing path (writing $c \overset{o}{\twoheadrightarrow} c'$) where it is clear from context.

Definition 4.3.2 (Admissable marked sets)**.** A set M of marked nodes is admissable for some sharing-graph G if:

1. The focus node is unmarked: $\bullet \notin M$

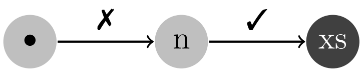
2. For every node c, at least one of the following is true:

   - The node is unmarked: $c \notin M$, OR
   - The node is unreachable from $\bullet$, OR
   - The node is cheap: $c \equiv x \wedge \mathit{cheap}(H(x))$ for the heap H in the corresponding state, OR
   - The node is marked ($c \in M$), and there is a *unique* unmarked node $c_{\mathit{root}} \notin M$ such that all pushing paths in G that end at c have the form $c_{\mathit{root}} \overset{\checkmark}{\underset{M}{\twoheadrightarrow}} c$

In order to solidify our intuition about admissability, we now consider some examples of admissable markings.

Example 1: work-duplicating contexts

$$\langle xs \mapsto \mathit{enumFromTo}\ a\ b, n \mapsto \mathit{length}\ xs \,|\, \lambda y.\, n + y \,|\, \epsilon \rangle$$

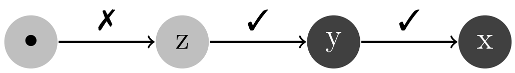


In this example, it would be inadmissable to mark the node n because then there would be a pushing path $\bullet \overset{✗}{\twoheadrightarrow} n$. Indeed, marking n would be dangerous for work duplication reasons because it would cause the expensive n binding to be pushed underneath the λ.

However, it is admissable to mark xs because if we do there is only a single pushing path ending at xs, namely $n \overset{\checkmark}{\twoheadrightarrow} xs$. This reflects the fact that as long as n is residualised (i.e. not marked and thus pushed under the lambda), it is safe to inline arbitrary **let**-bindings into it.

Example 2: transitive uses

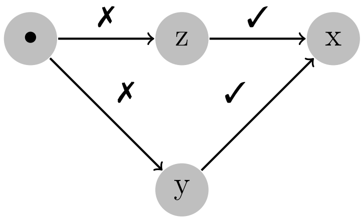
$$\langle x \mapsto \mathit{enumFromTo}\ a\ b, y \mapsto \mathit{length}\ xs, z \mapsto \mathit{fib}\ y \,|\, \mathit{Just}\ z \,|\, \epsilon \rangle$$



Similarly to the previous example, it is admissable to mark y and x because there is a single pushing path ending at each of them (both of which begin at z). As expected, this admissable marking does not lead to work duplication because it corresponds to pushing the y and x bindings into the **let**-binding for z.

Example 3: multiple uses

$$\langle x \mapsto \mathit{fib}\ y, y \mapsto f\ x, z \mapsto g\ x \,|\, (y, z) \,|\, \epsilon \rangle$$

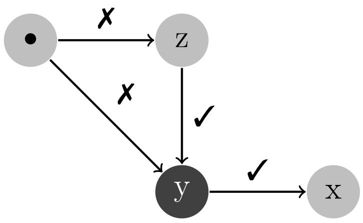


In this example, it is inadmissable to mark z or else we would have a pushing path $\bullet \overset{\text{✗}}{\twoheadrightarrow} z$ (similarly, marking y is inadmissable). As a consequence, it is inadmissable to mark x because if we did we would have *two* pushing paths $z \overset{\text{✓}}{\twoheadrightarrow} x$ and $y \overset{\text{✓}}{\twoheadrightarrow} x$.

The inadmissability of x reflects the fact that pushing it into both y and z would cause the work memoised by x to be duplicated.

Example 4: cheap bindings

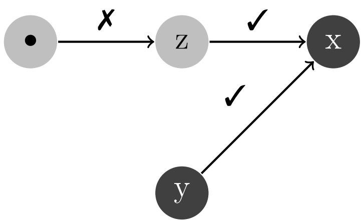
$$\langle x \mapsto \mathit{fib}\ n, y \mapsto \mathit{Just}\ x, z \mapsto f\ y \,|\, (y, z) \,|\, \epsilon \rangle$$



If y were not *cheap* in this example, it would be inadmissable to mark it. As it is cheap, however, y may be marked. However, it is still inadmissable to mark x because doing so would give rise to the two pushing paths $\bullet \overset{\text{✗}}{\twoheadrightarrow} x$ and $z \overset{\text{✓}}{\twoheadrightarrow} x$.

Example 5: unreachable nodes

$$\langle x \mapsto \mathit{enumFromTo}\ 1\ 100, y \mapsto \mathit{length}\ x, z \mapsto \mathit{sum}\ x \,|\, \mathit{Just}\ z \,|\, \epsilon \rangle$$



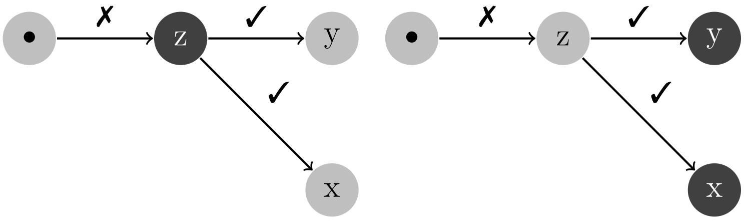
In this example, it is admissable to mark y as it is unreachable from the focus node $\bullet$. Because we are able to mark y, it is admissable to mark x as well. Marking any nodes which are not reachable from the focus node has a twofold effect:

- It ensures that we do not consider dead heap bindings such as y above to use their free variables. If we did not drop such nodes then we might risk identifying x as a shared heap binding used by both y and z which we could not then inline into the residual heap binding for z.

- Similarly, it ensures that dead bindings originating from update frames, such as y in $\langle \epsilon \,|\, x \,|\, \textbf{0:case} \bullet \textbf{of } \mathit{True} \to e_1; \mathit{False} \to e_2, \textbf{1:update } y, \textbf{2:} \bullet\ \ z \rangle$, do not count as uses of those update frames. Without dropping such nodes, we risk that in the above state we would not be able to inline the trailing stack frames $\textbf{update}\ y, \bullet\ z$ into the branches of the enclosed **case** as the update frame for y would be forced to be residualised due to apparent sharing of the node 1 between the nodes 0 and y.

Maximum markings A natural question to ask is whether for some particular graph there exists a *maximum marking*: i.e. an admissable marking such that any other admissable marking is a subset of it. If such a maximum existed, we would certainly want to use it to solve a sharing-graph for the splitter to ensure that as much information as possible is propagated. In fact, such a maximum does not necessarily exist. Consider:

$$\langle x \mapsto \mathit{fib}\ n, y \mapsto \mathit{fib}\ m, z \mapsto (x, y) \,|\, \lambda a. f\ z \,|\, \epsilon \rangle$$



Either of the two admissable markings suggested above are *maximal* in the sense that not even a single additional node can be marked without the marking becoming inadmissable, but they are not equivalent. Therefore, a unique maximum does not exist.

Note, however, that one of the two markings is preferable to the other: we should prefer the marking $\{z\}$ to $\{y, x\}$. The reason is that z is a simple value (x, y), and pushing the definitions of x and y is therefore pointless since they cannot interact further with the boring "value context" in which they are used. However, pushing z under the λ could potentially lead to simplification because the function f might scrutinise its argument.

Because of this observation, when solving a sharing-graph we will *always* prefer to mark cheap nodes if we have a choice[1]. Under this constraint, a unique maximum marking *does* in fact exist. When we talk about finding the maximum marking later in this document, we mean that we find it under this constraint.

To prove that a maximum marking exists, we first need a lemma that shows that marking groups of nodes which are admissable because they have unique pushing paths does not prevent later marking of *other* groups of nodes which are admissable for the same reason. Essentially, this lemma states that it does not matter in which order we mark nodes when building a marking by iterated set union.

Lemma 4.3.1 (Admissability preservation)**.** *For a sharing-graph G, if there exist markings M, M' and M'' such that:*

- *All of M, $M \cup M'$ and $M \cup M''$ are admissable*
- *If $\mathit{cheap}(H(x))$ then $x \in M$*
- *If c is unreachable from $\bullet$ in G then $c \in M$*

Then $M \cup M' \cup M''$ is also admissable.

[1] The strategy of always marking cheap nodes can give non-optimal results, but only in the unusual situation where we have a type-abstraction Λ over a non-value, such as $\textbf{let}\ y = \mathit{not}\ z; x = \Lambda\alpha. \textbf{case}\ y\ \textbf{of}\ \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True}\ \textbf{in}\ (\mathit{id}\ x, \mathit{id}\ x)$. An optimal pushed state for this would be $\langle x \mapsto \Lambda\alpha.\ \langle y \mapsto \mathit{not}\ z \,|\, y \,|\, \textbf{case} \bullet \textbf{of}\ \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True} \rangle \,|\, (\langle \epsilon \,|\, \mathit{id} \,|\, \bullet\ x \rangle, \langle \epsilon \,|\, \mathit{id} \,|\, \bullet\ x \rangle) \,|\, \epsilon \rangle$, which allows deforestation of the result of *not*. However if we always mark cheap nodes then we will create the pushed state $\langle y \mapsto \mathit{not}\ z \,|\, (\langle x \mapsto \ldots \,|\, \mathit{id} \,|\, \bullet\ x \rangle, \langle x \mapsto \ldots \,|\, \mathit{id} \,|\, \bullet\ x \rangle) \,|\, \epsilon \rangle$ where the call to *not* is residualised.

Proof. Our proposition is that, given node $c \in (M \cup M' \cup M'')$ but $c \notin (M \cup M') \vee c \notin (M \cup M'')$, there must exist c' such that there is a unique pushing path $c' \underset{M \cup M' \cup M''}{\overset{\checkmark}{\twoheadrightarrow}} c$. We proceed by induction on the minimum distance (in number of edges) between c and any node $\notin (M \cup M' \cup M'')$.

In the base case, the minimum distance is 0, and hence $c \notin (M \cup M' \cup M'')$. However, this directly contradicts one of our assumptions, and so we are done.

In the inductive case, we consider only the case $c \notin (M \cup M'')$ (the other case follows symmetrically). We can immediately deduce that $c \in M'$. Because of our assumptions about cheapness and unreachability, it must be the case therefore that there exists a unique pushing path $c' \underset{M \cup M'}{\overset{\checkmark}{\twoheadrightarrow}} c$, where $c' \notin (M \cup M')$.

If $c' \notin M''$ then the proposition follows immediately since we would have a unique pushing path $c' \underset{M \cup M' \cup M''}{\overset{\checkmark}{\twoheadrightarrow}} c$. In the other case ($c' \in M''$), we can induct to deduce that there must exist c'' such that there is a unique pushing path $c'' \underset{M \cup M' \cup M''}{\overset{\checkmark}{\twoheadrightarrow}} c'$. (Induction is valid because $c' \neq c$ and so it must be strictly closer to a node $\notin (M \cup M' \cup M'')$).

Armed with the result of induction, we know that the result of joining the two paths must yield a unique pushing path $c'' \underset{M \cup M' \cup M''}{\overset{\checkmark}{\twoheadrightarrow}} c$, as required. □

Given this lemma the main theorem is easy to prove:

Theorem 4.3.2 (Maximum marking exists)**.** *For a graph G, a maximum marking M exists given that if cheap$(H(x))$ then $x \in M$.*

Proof. Assume that this was not the case: i.e. for some graph G there were two markings M_1 and M_2 such that $M_1 \neq M_2$ and both markings are maximal (i.e. there is no admissable $M_1' \supset M_1$ or $M_2' \subset M_2$).

Let M_0 be the smallest marking such that if *cheap*$(H(x))$ then $x \in M_0$ and if c is unreachable from $\bullet$ in G then $c \in M_0$. By maximality of both markings, we can immediately see that $M_0 \subset M_1$ and $M_0 \subset M_2$. Therefore, since M_0 is always admissable, by Lemma 4.3.1 we know that $M_0 \cup M_1 \cup M_2 = M_1 \cup M_2$ must be admissable. However, this contradicts maximality of M_1 and M_2. □

Finding the maximum marking A simple way to find the maximum marking is to exhaustively generate all possible markings for a graph and then select the one with the largest number of elements that passes an admissability test. However, for graphs with n nodes there will be 2^n possible markings, so this may be inefficient, particularly since we observed that in practice sharing graphs can grow to contain hundreds of nodes. In practice we prefer the polynomial-time algorithm of Figure 4.3.

The predicate *reachable*(c, c', G) holds when c' is reachable from c in the graph G. The function *sccs*(G) returns a topologically-ordered list of (graph, edge map) pairs. Each graph in the list corresponds to a single strongly-connected-component, containing the nodes within the SCC as well as the edges between them (note that all nodes in an SCC are guaranteed to be reachable from each other). Each graph is paired with a mapping from successor SCCs to another mapping *Map* $(c,\ c)\ o$ which maps pairs of nodes (c, c') where c is in the current SCC and c' is in a successor SCC to the edge annotation o that

$$
\begin{array}{rcl}
\mathit{mark}(N,G) & = & \widehat{\mathit{mark}}(N,\varnothing,\mathit{sccs}(G))
\end{array}
$$

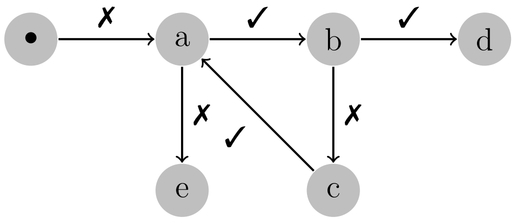
$$
\begin{array}{rcl}
\widehat{\mathit{mark}} & :: & (\mathit{Set}\ c, \mathit{Map}\ c\ (\mathit{Maybe}\ c), [(\mathit{Graph}\ c\ o,\ \mathit{Map}\ (\mathit{Graph}\ c\ o)\ (\mathit{Map}\ (c,\ c)\ o))]) \\
& \rightarrow & \mathit{Map}\ c\ c \\
\widehat{\mathit{mark}}(N,P,\epsilon) & = & \varnothing \\
\widehat{\mathit{mark}}(N,P,((G,E),\overline{(G,E)})) & = & M \cup \widehat{\mathit{mark}}(N,P',\overline{(G,E)}) \\
\textbf{where}\quad P_G & = & \{c \mapsto P_c \mid c \mapsto P_c \in P, c \in \mathit{dom}(G)\} \\
M & = & \begin{cases} \varnothing & P_G = \varnothing \\ \{c \mapsto c_{\mathit{trg}} \mid c \in \mathit{dom}(P_G)\} & \mathit{dom}(G) \cap N \equiv \varnothing \wedge (c_{\mathit{trg}} \equiv \boxplus\, \mathit{rng}(P_G)) \neq \boxslash \\ \mathit{mark}(N, G_{\mathit{filt}}) & \text{otherwise} \end{cases} \\
G_{\mathit{filt}} & = & \{c \mapsto \{c' \mapsto o \mid c' \mapsto o \in G_c, c' \notin \mathit{dom}(P_G)\} \mid c \mapsto G_c \in G\} \\
P' & = & \begin{cases} P \cup_{\boxplus} \bigcup_{\boxplus} \{\{c' \mapsto \begin{cases} c_{\mathit{trg}} & c \mapsto c_{\mathit{trg}} \in M \\ \boxslash & o \equiv \text{✗} \\ c & \text{otherwise} \end{cases} \} \mid G' \mapsto E_{G'} \in E, (c,c') \mapsto o \in E_{G'}\} \end{cases}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{contract}(\epsilon,G) & = & G \\
\mathit{contract}((c,M),G) & = & \mathit{contract}(M, \mathit{contract1}(c,G)) \\
\mathit{contract1}(c,G) & = & \{\quad c_0 \mapsto \qquad \{c'_1 \mapsto o \mid c \mapsto o \in G_{c_0}, c'_1 \mapsto o' \in (G(c) \setminus \{c\})\} \\
& & \qquad\qquad \cup_{\oplus} \quad (G_{c_0} \setminus \{c\}) \\
& & \mid \quad c_0 \mapsto G_{c_0} \in (G \setminus \{c\})\}
\end{array}
$$

$$
\begin{array}{cccl}
c_0 & \boxplus & c_1 & = \begin{cases} c_0 & c_0 \equiv c_1 \\ \boxslash & \text{otherwise} \end{cases} \\
o_1 & \oplus & o_2 & = \begin{cases} \checkmark & o_1 = \checkmark \wedge o_2 = \checkmark \\ \text{✗} & \text{otherwise} \end{cases} \\
C_0 & \cup_\star & C_1 & = \{c \mapsto o_0 \mid c \mapsto o_0 \in C_0, c \notin \mathit{dom}(C_1)\} \\
& & & \cup \ \{c \mapsto o_1 \mid c \mapsto o_1 \in C_1, c \notin \mathit{dom}(C_0)\} \\
& & & \cup \ \{c \mapsto o_0 \star o_1 \mid c \mapsto o_0 \in C_0, c \mapsto o_1 \in C_1\}
\end{array}
$$

Figure 4.3: Solving a sharing-graph for the marked set

was on the edge $c \overset{o}{\rightarrow} c'$ in G. So for example for the following graph:



One acceptable result (the relative ordering of e and d in the output is unimportant) of *sccs* would be:

$$
\begin{array}{lll}
[& (\{\bullet \mapsto \varnothing\}, & \{\{a \mapsto \{b \mapsto \checkmark\}, b \mapsto \{c \mapsto \text{✗}\}, c \mapsto \{a \mapsto \checkmark\}\} \\
& & \quad \mapsto \{(\bullet, a) \mapsto \text{✗}\}\}) \\
, & (\{a \mapsto \{b \mapsto \checkmark\}, b \mapsto \{c \mapsto \text{✗}\}, c \mapsto \{a \mapsto \checkmark\}\}, & \{\{e \mapsto \varnothing\} \mapsto \{(a,e) \mapsto \text{✗}\}, \\
& & \ \{d \mapsto \varnothing\} \mapsto \{(b,d) \mapsto \checkmark\}\}) \\
, & (\{e \mapsto \varnothing\}, & \varnothing) \\
, & (\{d \mapsto \varnothing\}, & \varnothing)]
\end{array}
$$

The set N that appears as an argument to *mark* is the so-called generalisation set used to prevent certain pushings from occurring. At present, we will always assume that $N = \{\bullet\}$, but in Section 6.2.2 we will make more use of this parameter.

We conjecture (but have not proved in detail) that *mark* can be used to compute a maximum marking:

Conjecture 4.3.3. *The marking* $M_{cheap} \cup dom(mark(\{\bullet\}, contract(M_{cheap}, G_{orig})))$ *is a maximum marking for* G_{orig}*, where* $M_{cheap} = \{c \mid c \in dom(G_{orig}), \neg reachable(\bullet, c, G_{orig}) \vee (c \equiv x \wedge cheap(H(x)))\}$.

The reason we have for believing this is true is that for every SCC that $\widehat{mark}$ considers, the maximum admissable number of nodes in that SCC are marked. We only definitively stop attempting to mark a node when we have detected that it acts as a SCC "entry node" (i.e. the node has a predecessor in another SCC) for a SCC whose entry nodes do not have a common target, in which case it is clearly inadmissable (see the section below) to mark any of the entry nodes.

Furthermore, by Lemma 4.3.1, we know that marking fewer nodes in an earlier SCC cannot possibly allow us to mark more nodes in a later one. Therefore, the overall marking set must be maximal.

Because of Theorem 4.3.2, if the algorithm constructed a maximal marking, it would also be a maximum marking.

Although this conjecture does not say anything about the case where the generalisation set N supplied to *mark* is not $\{\bullet\}$, we expect that for general N a version of the conjecture holds true where the computed marked set is the maximum marking "with respect to N" i.e. it is the maximal marking that does not mark any nodes of N. This property becomes important when *split* is used for generalisation (Section 6.2.2)

An explanation of *mark* We begin by ensuring that all cheap and unreachable nodes in G_{orig} are marked, and *contract* the graph to reflect that. The essential feature of $contract(c, G_{orig})$ is that in the output the node c will be missing, and the old predecessors of c will gain all of the old successors of c as new successors (using $\oplus$ to join any annotations on edges which would otherwise go between the same pairs of nodes).

With this dealt with, the rest of the algorithm is free to attempt to mark only those nodes which satisfy the unique-pushing-path admissability criterion. The $\widehat{mark}$ function considers the graph one SCC at a time in topological order. As it goes along the SCCs it accumulates a predecessor information mapping P from nodes c to either:

- $\boxslash$, if there certainly cannot be a unique pushing path to c.
- A "target" node c_{trg} if—from the edges encountered so far—it appears that there is a unique pushing path ending at c, $c_{trg} \overset{\checkmark}{\twoheadrightarrow} c$.

For each SCC it constructs the map P_G, mapping entry nodes (i.e. those which have incoming edges from some preceding SCC) to their predecessor information. Because we consider nodes in topological order, at the point we reach a SCC the predecessor information has been updated with information from all predecessors to nodes in the current SCC G, except for any predecessors which are in G itself.

Now we determine the contribution M to the final graph marking from this SCC. In fact, within *mark* we temporarily redefine markings to be not a set, but rather a mapping from marked nodes to their target nodes: so if $c \mapsto c' \in M$ then in $contract(M_{cheap}, G_{orig})$ there will be a pushing path $c \underset{M_{final}}{\overset{\checkmark}{\twoheadrightarrow}} c'$ for $M_{final} = dom(mark(\{\bullet\}, contract(M_{cheap}, G_{orig})))$. The domain of M will always be a subset of the nodes in the current SCC graph G.

In the case that the entire SCC has no predecessors (i.e. $P_G \equiv \varnothing$) we force M to be empty. Because the graph input to *mark* has no unreachable nodes (they were contracted away), this will only occur in two cases:

1. If G is the acyclic SCC containing only the focus node $\bullet$.
2. If G is an acyclic SCC containing only a node which was a SCC entry node for an earlier use of $\widehat{mark}$ which is recursively calling this instance of $\widehat{mark}$ (this case is discussed later).

In the case that all of the entry nodes of G have a common target node recorded in P_G, it is admissable to mark the whole SCC, and we do so.

If the entry nodes of G do *not* have a common target node, we can deduce that it inadmissable to mark *any* of the entry nodes. The only entry nodes for which this is not obvious are those entry nodes c which have a common target recorded in P_G i.e. $P_G(c) = c'$. However, think about what would happen if we marked c in isolation. If we did, it would have at least two distinct pushing paths terminating at it: one path from c' and another from one of its predecessors in G. To avoid inadmissability, all predecessors of c in G would have to be marked as well, and (by the same argument) all of *their* predecessors in G, until the *entire* SCC G would have to be marked. But this is clearly inadmissable as we already assumed that the entry nodes (a subset of the nodes of G) do not have a common target.

In this case, even though we cannot mark any of the entry nodes, we might be able to mark some of the interior nodes of the SCC which are not successors of nodes in earlier SCCs. We recursively use *mark* to determine the maximum set of nodes of G that are suitable for marking (and their target nodes), recursing with G_{filt}—a version of G which ensures that entry nodes do not have any predecessors and hence will never be marked.

When we recurse, we do so with a graph, G_{filt}, which has a strictly smaller number of edges that the graph originally given to *mark*, so it is easy to show that termination is guaranteed.

After M has been determined, we update P with predecessor information gleaned from G. In particular, if a node c in G has been marked and has target node c_{trg} (i.e. $c \mapsto c_{trg} \in M$), then we claim that any successor c' of c will (in the absence of other predecessors of c') have the same target node c_{trg}. Unmarked nodes are either their own targets to their successors, or cannot act as targets because doing so would mean pushing along a ✗-marked edge.

4.3.3 Creating the pushed state

Now that we have defined sharing-graphs as well as how to create and solve them, we are in a position to straightforwardly define the *push* function used to create a pushed state suitable for recursive supercompilation from a standard state. In fact, we will define *push* in terms of an auxiliary function $\widehat{push}$ which takes a generalisation set N similar to the generalisation set argument to the *mark* function. The definition of *push* is simple:

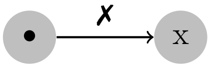
$$\mathit{push}\ \mathcal{S} \quad = \quad \widehat{\mathit{push}}(\{\bullet\})(\mathcal{S})$$

For now this will be the only use of $\widehat{push}$. It will be used in its full generality when we come to define generalisation in Section 6.2.

In order to solidify intuition about the operation of *push* we consider a number of examples of inputs to *push*, their sharing graphs (with corresponding maximum markings), and the optimal (most information propagated) pushed states that will be produced as a result of *push*. Note that for readability in each case we will omit any dead heap bindings from both the pushed state heap and the nested state heaps.

Example 1: no pushing

$$\langle x \mapsto \mathit{fib}\ 100 \mid \lambda y.\, x + 1 \mid \epsilon \rangle$$

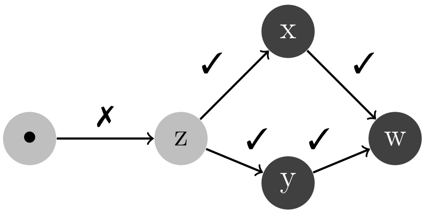


$$\langle x \mapsto \langle \epsilon \mid \mathit{fib}\ 100 \mid \epsilon \rangle \mid \lambda y.\, \langle \epsilon \mid x + 1 \mid \epsilon \rangle \mid \epsilon \rangle$$

In this simple example no information can be propagated and so we end up simply recursively supercompiling some subexpressions of the input state.

Example 2: heap pushing

$$\langle w \mapsto \mathit{fib}\ 100, x \mapsto \mathit{fib}\ w, y \mapsto \mathit{fib}\ w, z \mapsto \mathit{fib}\ x + \mathit{fib}\ y \mid \mathit{Just}\ z \mid \epsilon \rangle$$

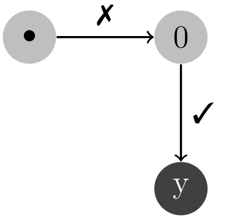


$$\langle z \mapsto \langle w \mapsto \mathit{fib}\ 100, x \mapsto \mathit{fib}\ w, y \mapsto \mathit{fib}\ w \mid \mathit{fib}\ x + \mathit{fib}\ y \mid \epsilon \rangle \mid \mathit{Just}\ z \mid \epsilon \rangle$$

This illustrates how marked heap bindings can be pushed into the nested states that we supercompile, potentially propagating useful information we can use to simplify the program at a later date.

Example 3: heap pushing into the stack

$$\langle y \mapsto \mathit{fib}\ 100 \mid x \mid \mathbf{0{:}case} \bullet \mathbf{of}\ \mathit{True} \to y + 1; \mathit{False} \to y + 2 \rangle$$

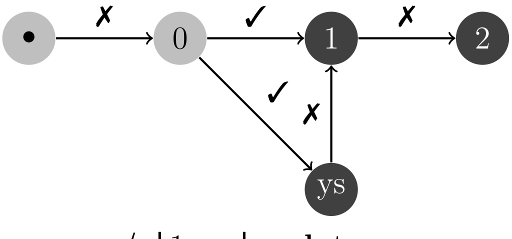


$$\langle \epsilon \mid x \mid \mathbf{case} \bullet \mathbf{of}\ \mathit{True} \to \langle y \mapsto \mathit{fib}\ 100 \mid y + 1 \mid \epsilon \rangle ; \mathit{False} \to \langle y \mapsto \mathit{fib}\ 100 \mid y + 2 \mid \epsilon \rangle \rangle$$

It is admissable to mark heap bindings for pushing into *any* context if doing so is admissable, and hence the pushing would not cause work duplication. In this example, we inline a heap binding into the branch of a stack frame.

Example 4: stack pushing

$$\langle \epsilon \mid \mathit{unk} \mid \mathbf{0{:}case} \bullet \mathbf{of}\ \mathit{True} \to 1 : \mathit{ys}; \mathit{False} \to 2 : \mathit{ys}, \mathbf{1{:}update}\ \mathit{ys}, \mathbf{2{:}case} \bullet \mathbf{of}\ x{:}\,\_ \to x; [\,] \to 0 \rangle$$

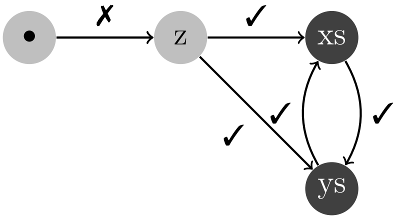


$$\left\langle \epsilon \,\middle|\, \mathit{unk} \,\middle|\, \begin{array}{rcl} \mathbf{case} \bullet \mathbf{of}\ \mathit{True} & \to & \langle \epsilon \mid 1 : \mathit{ys} \mid \mathbf{update}\ \mathit{ys}, \mathbf{case} \bullet \mathbf{of}\ x{:}\,\_ \to x; [\,] \to 0 \rangle \\ \mathit{False} & \to & \langle \epsilon \mid 2 : \mathit{ys} \mid \mathbf{update}\ \mathit{ys}, \mathbf{case} \bullet \mathbf{of}\ x{:}\,\_ \to x; [\,] \to 0 \rangle \end{array} \right\rangle$$

In the same way that we can inline marked heap bindings into their use sites, we can inline marked stack frames into the contexts that the frames receive values from. In practice, this means that we can push later stack frames into the branches of a **case** frame.

Example 5: mutually-recursive heap pushing

$$\langle xs \mapsto f\ x\ ys, ys \mapsto f\ x\ xs, z \mapsto head\ xs + head\ ys \mid Just\ z \mid \epsilon\rangle$$

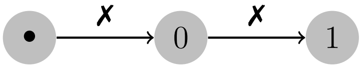


$$\langle z \mapsto \langle xs \mapsto f\ x\ ys, ys \mapsto f\ x\ xs \mid head\ xs + head\ ys \mid \epsilon\rangle \mid Just\ z \mid \epsilon\rangle$$

Just like standard non-recursive heap bindings, our framework allows suitable recursive and even mutually-recursive heap bindings to be marked and hence pushed.

Example 6: positive information propagation

$$\langle \epsilon \mid unk \mid \mathbf{0{:}update}\ ys, \mathbf{1{:}case} \bullet \mathbf{of}\ True \to f\ unk\ ys; False \to True\rangle$$

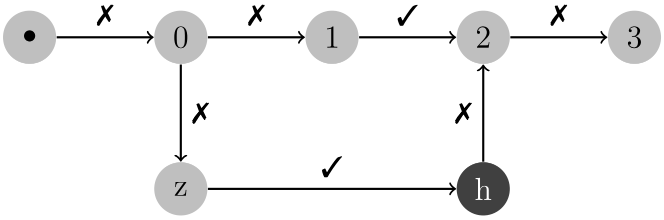


$$\left\langle \epsilon \middle| unk \middle| \begin{array}{l} \mathbf{0{:}update}\ ys, \mathbf{1{:}case} \bullet \mathbf{of}\ True \to \langle unk \mapsto True, ys \mapsto True \mid f\ unk\ ys \mid \epsilon\rangle \\ \qquad\qquad\qquad\qquad\qquad\quad False \to \langle unk \mapsto False, ys \mapsto False \mid True \mid \epsilon\rangle \end{array} \right\rangle$$

In supercompilation, we can learn about the structure of a hitherto-unknown value from scrutinisation. This learning process is implemented by *push*.

Example 7: sharing through the stack

$$\langle z \mapsto filter\ h \mid unk \mid \mathbf{0{:}} \bullet\ \ z, \mathbf{1{:}case} \bullet \mathbf{of}\ True \to f; False \to g, \mathbf{2{:}update}\ h, \mathbf{3{:}} \bullet\ \ x\rangle$$



$$\langle z \mapsto \langle \epsilon \mid filter\ h \mid \epsilon\rangle \mid unk \mid \bullet\ z, \mathbf{case} \bullet \mathbf{of}\ True \to \langle \epsilon \mid f \mid \epsilon\rangle; False \to \langle \epsilon \mid g \mid \epsilon\rangle, \mathbf{update}\ h, \bullet\ x\rangle$$

If your language has no general recursive heap bindings, there can never be any paths in the sharing graph from heap nodes to stack nodes (though paths from the stack to the heap *can* occur). In this case, the original program used general recursion to build a graph such that there is a path from the heap to a stack frame, and that graph was such that it prevented the admissable marking of all but one node in the sharing-graph.

This example also shows that it is irrelevant whether the "heap binding node" for a variable bound by an update frame (in this case h) is marked if the corresponding update frame (in this case 2) is not also marked: the update frame will only be pushed down if the stack frame itself is marked.

$$
\begin{aligned}
\widehat{\mathit{push}}(N)\,\langle H \,|\, e \,|\, K\rangle_{\Sigma|\Gamma} \quad &= \quad \left\langle \mathring{H} \,\middle|\, \mathring{e} \,\middle|\, \mathring{K} \right\rangle \\
\textbf{where} \quad (\hat{\Gamma}_H, \hat{H}) \quad &= \quad \mathit{prep}_H(M, H) \\
\hat{e} \quad &= \quad \mathit{prep}_\bullet(e) \\
(\hat{K}, \hat{\Gamma}, \mathring{K}) \quad &= \quad \mathit{prepresid}_K(M|\hat{\Sigma}, \hat{\Gamma}_H, \hat{H}, \hat{e}|0, K) \\
\mathring{e} \quad &= \quad \begin{cases} \mathit{resid}_\bullet(\hat{\Sigma}, \hat{\Gamma}, \hat{H}|e) & \bullet \in N \\ \left\langle \hat{H} \,\middle|\, e \,\middle|\, \hat{K} \right\rangle_{\hat{\Sigma}|\hat{\Gamma}} & \text{otherwise} \end{cases} \\
\mathring{H} \quad &= \quad \mathit{resid}_H(\hat{\Sigma}, \hat{\Gamma}, \hat{H}|H) \\
\\
M_{\mathit{cheap}} \quad &= \quad (\{x \mid x{:}\tau \mapsto e \in H, \mathit{cheap}(e)\} \cup \\
&\qquad \{c \mid c \in \mathit{dom}(G), \neg \mathit{reachable}(\bullet, c, G)\}) \setminus N \\
M \quad &= \quad M_{\mathit{cheap}} \cup \mathit{dom}\,(\mathit{mark}(N, \mathit{contract}(M_{\mathit{cheap}}, G))) \\
\\
G_H \quad &= \quad \mathit{graph}_H(H) \\
G_e \quad &= \quad \begin{cases} \mathit{graph}_\bullet(e) & \bullet \in N \\ \{\bullet \mapsto (\{0 \mapsto \checkmark\} \cup \widehat{\mathit{graph}}_e(e)(\checkmark))\} & \text{otherwise} \end{cases} \\
G_K \quad &= \quad \mathit{graph}_K(K) \\
G \quad &= \quad G_H \cup G_e \cup G_K
\end{aligned}
$$

Figure 4.4: Creating the pushed state

$$
\mathit{prep}_\bullet(e) \quad = \quad \begin{cases} x & e \equiv x \\ \epsilon & \text{otherwise} \end{cases}
\qquad
\begin{aligned}
\mathit{resid}_\bullet(\hat{\Sigma}, \hat{\Gamma}, \hat{H}|x) \quad &= \quad x \\
\mathit{resid}_\bullet(\hat{\Sigma}, \hat{\Gamma}, \hat{H}|\lambda x{:}\tau.\, e) \quad &= \quad \lambda x{:}\tau.\, \left\langle \hat{H} \,\middle|\, e \,\middle|\, \epsilon \right\rangle_{\hat{\Sigma}|\hat{\Gamma}, x:\tau} \\
\mathit{resid}_\bullet(\hat{\Sigma}, \hat{\Gamma}, \hat{H}|\Lambda \alpha{:}\kappa.\, e) \quad &= \quad \Lambda \alpha{:}\kappa.\, \left\langle \hat{H} \,\middle|\, e \,\middle|\, \epsilon \right\rangle_{\hat{\Sigma}, \alpha:\kappa|\hat{\Gamma}} \\
\mathit{resid}_\bullet(\hat{\Sigma}, \hat{\Gamma}, \hat{H}|\mathbf{C}\ \overline{\tau}\ \overline{x}) \quad &= \quad \mathbf{C}\ \overline{\tau}\ \overline{x} \\
\mathit{resid}_\bullet(\hat{\Sigma}, \hat{\Gamma}, \hat{H}|e) \quad &= \quad \left\langle \hat{H} \,\middle|\, e \,\middle|\, \epsilon \right\rangle_{\hat{\Sigma}|\hat{\Gamma}}
\end{aligned}
$$

$$
\mathit{prep}_H(M, H) \quad = \quad (\{x{:}\tau \mid x{:}\tau \mapsto e \in H, x \notin M\}, \{x{:}\tau \mid x{:}\tau \mapsto e \in H, x \in M\})
$$

$$
\mathit{resid}_H(\hat{\Sigma}, \hat{\Gamma}, \hat{H}|\overline{h}) \quad = \quad \{x{:}\tau \mapsto \mathit{resid}_\bullet(\hat{\Sigma}, \hat{\Gamma}, \hat{H}|e) \mid x{:}\tau \mapsto e \in H, x \notin \hat{H} \vee \mathit{cheap}(e)\}
$$

$$
\begin{aligned}
\mathit{prepresid}_K(M|\hat{\Sigma}, \hat{\Gamma}, \hat{H}, \hat{e}|i, \epsilon) \quad &= \quad (\epsilon, \hat{\Gamma}, \epsilon) \\
\mathit{prepresid}_K(M|\hat{\Sigma}, \hat{\Gamma}, \hat{H}, \hat{e}|i, (\kappa, K)) \quad &= \quad \begin{cases} ((\kappa, \hat{K}), \hat{\Gamma}'', K') & i \in M \\ (\epsilon, \hat{\Gamma}'', (\mathit{resid}_\kappa(\hat{\Sigma}, \hat{\Gamma}'', \hat{H}, \hat{e}, \hat{K}|\kappa), K')) & \text{otherwise} \end{cases} \\
\textbf{where} \quad (\hat{K}, \hat{\Gamma}'', K') \quad &= \quad \mathit{prepresid}_K(M|\hat{\Sigma}, \hat{\Gamma}', \hat{H}, \hat{e}'|i+1, K) \\
(\hat{e}', \hat{\Gamma}') \quad &= \quad \begin{cases} ((x, \hat{e}), (x{:}\tau, \hat{\Gamma})) & \kappa \equiv \textbf{update}\ x{:}\tau \wedge i \notin M \\ (\epsilon, \hat{\Gamma}) & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{resid}_\kappa(\hat{\Sigma}, \hat{\Gamma}, \hat{H}, \hat{e}, \hat{K}|\textbf{update}\ x{:}\tau) \quad &= \quad \textbf{update}\ x{:}\tau \\
\mathit{resid}_\kappa(\hat{\Sigma}, \hat{\Gamma}, \hat{H}, \hat{e}, \hat{K}|\bullet\ \ x) \quad &= \quad \bullet\ x \\
\mathit{resid}_\kappa(\hat{\Sigma}, \hat{\Gamma}, \hat{H}, \hat{e}, \hat{K}|\bullet\ \ \tau) \quad &= \quad \bullet\ \tau \\
\mathit{resid}_\kappa(\hat{\Sigma}, (\hat{\Gamma}, \overline{y{:}\mathbf{T}\ \overline{\upsilon}}), \hat{H}, \overline{y}, \hat{K}|\textbf{case} \bullet \textbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau}\ \ \rightarrow\ \ e}) & \\
= \quad \overline{\textbf{case} \bullet \textbf{of}\ \mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau}\ \ \rightarrow\ \ \left\langle \hat{H}, \overline{y{:}\mathbf{T}\ \overline{\upsilon} \mapsto \mathbf{C}\ \overline{\upsilon}, \overline{\alpha}\ \overline{x}} \,\middle|\, e \,\middle|\, \hat{K} \right\rangle_{\hat{\Sigma}, \overline{\alpha:\kappa}|\hat{\Gamma}, \overline{x:\tau}}} &
\end{aligned}
$$

Figure 4.5: Preparing and residualising syntax for pushed states

Defining $\widehat{push}$ Figure 4.4 shows the definition of the $\widehat{push}$ function, which makes use of the auxiliary functions defined in Figure 4.5.

The essential action of $\widehat{push}$ is to construct a sharing-graph from the input using the *graph* functions, solve it using *mark*, and then use the *prep* and *resid* family of functions to construct a pushed state from the input state and the marking.

- The *prep* family of functions is (loosely speaking) designed to prepare syntax for pushing into the nested states within the final pushed state.
- The *resid* family of functions is designed to produce the syntax (heap, stack and focus) that belongs to the pushed state itself (rather than any nested state).

Interesting features of the definitions are:

- The prepared version of the focus $\hat{e}$ is not a term per-se, but rather a list of variables that the focus is certainly equal to. This is used to implement positive information propagation.
- Cheap heap bindings are included in both the prepared and residualised versions of the heap. This is because we both want to inline them into any use site, but also make them available for residualisation in the output in case they are required. You can think of this as if cheap bindings are *copied* down into nested states, whereas non-cheap bindings are *moved* down.
- We inline the *entire* prepared heap into each nested state with no regard for which of the heap bindings are actually required by the rest of that state. This might seem reckless, but in fact it does not lead to work duplication because any of the pushed bindings which are not required will of course not be evaluated and will eventually be dropped as dead code.
- In the definition of $resid_\kappa$, for anything other than a **case** frame we appear to throw away the portion $\hat{K}$ of the prepared stack passed as an argument, without including it in the result. In fact, because of the fact that all these stack frames have an ✗-edge to the following stack frame in the sharing-graph, $\hat{K}$ will always be empty.

4.3.4 The pushing algorithm does not duplicate work

Armed with our previous work on sharing-graphs and admissability, we can confidently state the following conjecture:

Conjecture 4.3.4 (Pushing is an improvement)**.** *For all N and $\mathcal{S}$, $\mathcal{S} \gtrsim \widehat{push}(N)(\mathcal{S})$*

As the pushing algorithm essentially only does **let**-floating and **case**-of-**case** transformation, it is easy to believe that the pushed state has the same meaning *denotationally* as the input. What is less clear is that it has the same meaning *intensionally* i.e. with regard to work sharing.

The reason we have for believing that pushing is actually an improvement is because we only inline (and hence risk work duplication for) those nodes which are marked by our use of the *mark* function, which by Conjecture 4.3.3 returns an marking which is admissable. Thus, every marked node c will be admissable, and so at least one of three situations will apply:

1. The node c will be unreachable in the sharing graph. In this case, we know that it must correspond to either:

  - A dead heap binding, in which case $c \equiv x$ and it cannot pose a problem since it must still be dead in the output of $\widehat{push}$ (whether it is bound in the heap of the pushed state or in the heaps of one or more nested states), and if $x \notin \mathrm{fvs}\,(H \setminus \{x\}) \cup \mathrm{fvs}\,(K) \cup \mathrm{fvs}\,(e)$ then $\langle H \,|\, e \,|\, K \rangle \mathrel{\underset{\sim}{\lhd\!\rhd}} \langle H \setminus \{x\} \,|\, e \,|\, K \rangle$.
  - A dead variable bound by an update frame, in which case (once again) it must still be dead in the output of $\widehat{push}$, in which case whether it is marked or not does not affect the pushed state in any way.

2. The node c corresponds to a cheap heap binding, in which case pushing it into any context (even inside a λ) is justified as an improvement by Theorem C.3.1 (if the bound term is a value) and by a similar argument if the bound term is a variable.

3. The node c has a unique pushing path terminating at it of the form $c' \overset{\checkmark}{\twoheadrightarrow} c$. In this case, we can see by inspection of $\widehat{push}$ that the prepared syntax corresponding to c will occur *only* as part of (one or more) nested states within the residualised syntax corresponding to c'.

   By considering the possible cases for the original syntax corresponding to c' one at a time we can see that either:

   - The graph generated from the original syntax could not have given rise to a pushing path rooted at c' starting with ✓, and so c' cannot have originated from that syntax. This case applies when c' corresponds to a λ in the focus of the input state
   - Or else moving the syntax for c down to be totally enclosed by the residualised syntax for c' does not cause work duplication. This is easy to see where c' corresponds to something like a heap binding (in which case we can justify the transformation by a rule of improvement theory that **let**-of-**let** transformation is a cost equivalence). It is only slightly harder to see where c' corresponds to something like a **case** frame, where the prepared syntax for c will occur syntactically once inside each of the branches of the residualised **case** frame in the pushed state.

4.4 Recursing to drive the pushed state

The simplest splitter would take the pushed state constructed by $\widehat{push}(\{\bullet\})(\mathcal{S})$, drive all of the nested states therein, and then rebuild a final term by replacing those nested states with the result of supercompiling them. This is the model we described in Section 4.1 with the *traverseState* function.

Dead heap bindings In practice, we use a slight refinement of this scheme which allows us to avoid supercompiling some of the nested states. As an example, consider the splitting of the following state:

$$\langle x \mapsto \mathit{fib}\ n \,|\, \lambda z.\, (\lambda y.\, 1)\ x \,|\, \epsilon \rangle$$

The corresponding pushed state will be:

$$\langle x \mapsto \langle \epsilon \,|\, \mathit{fib}\ n \,|\, \epsilon \rangle \;\;|\;\; \lambda z.\, \langle \epsilon \,|\, (\lambda y.\, 1)\ x \,|\, \epsilon \rangle \;\;|\;\; \epsilon \rangle$$

In the simple scheme, we would recursively drive both nested states. However, if we were to drive the state in the focus first we would find that the variable x is not mentioned in the supercompiled version of $\langle \epsilon \,|\, (\lambda y.\, 1)\ x \,|\, \epsilon \rangle$. This proves that the heap binding for x is in fact dead, and so it is not in fact necessary to drive it. Our refined scheme exploits this by driving the focus first, and then driving only those heap bindings which we discover to be used either by the focus or something (heap binding or stack frame) that the focus references[2].

This same refinement can also help when the pushed state contains manifestly dead heap bindings even *before* driving. For example, consider the following input state:

$$\langle f \mapsto \lambda x.\, \mathit{fib}\ (x+1) \,|\, \lambda y.\, f\ y + f\ y \,|\, \epsilon \rangle$$

Previously, when presenting the output of *push* we have implicitly omitted dead bindings from the pushed state. Temporarily returning to showing these dead bindings, we can write the pushed state for this input state as follows (remember that cheap heap bindings like f will be copied into all nested states as well as residualised in place):

$$\langle f \mapsto \langle f \mapsto \lambda x.\, \mathit{fib}\ (x+1) \,|\, \lambda x.\, \mathit{fib}\ (x+1) \,|\, \epsilon \rangle \;\;|\;\; \lambda y.\, \langle f \mapsto \lambda x.\, \mathit{fib}\ (x+1) \,|\, f\ y + f\ y \,|\, \epsilon \rangle \;\;|\;\; \epsilon \rangle$$

Our refined scheme will avoid driving the dead binding for f in the pushed state. Dead bindings are endemic in output of the *push* function defined in Section 4.3, so our refined driving scheme saves a lot of supercompilation work.

Dead stack frames In just the same way that some heap bindings are either manifestly dead or can become provably dead as a result of driving, and so need not be optimised by supercompilation, there may be cases in which *stack frames* become dead and hence discardable. Consider:

$$\langle \epsilon \,|\, \mathit{error} \,|\, \bullet\ \mathit{Bool}, \bullet\ \mathit{msg}, \mathbf{case} \bullet \mathbf{of}\ \mathit{True} \to \mathit{fib}\ n; \mathit{False} \to \mathit{fact}\ n \rangle$$

The *error* function is a Haskell function defined in the standard libraries which never returns directly[3], and so any stack frames following a saturated call to *error* will never be entered and so need not be optimised.

It is straightforward to define the splitter in such a way that it detects residual calls to these known-bottoming functions. Any stack suffix that occurs after a saturated call to such a function can be replaced with a trivial residual stack of the form $\mathbf{case} \bullet \mathbf{of}\ \epsilon$ which does not require any recursive invocation of the supercompiler. With this change, our final driven term would be

$$\mathbf{case}\ \mathit{error}\ \mathit{Bool}\ \mathit{msg}\ \mathbf{of}\ \epsilon$$

Note that we need to replace the discarded stack suffix with a nullary **case** stack frame rather than removing it entirely. This is purely for type compatibility reasons: if we used an empty stack instead, then our final driven term would have type *Bool* rather than the required type *Int*.[4]

[2] In order to make this work effectively, we need to modify the memoiser so that if driving a state $\mathcal{S}$ yields a term e for a promise hn, then it returns a call to hn for which only those variables free in e are free (which are a subset of those variables free in $\mathcal{S}$). So if the state $\mathcal{S}$ for promise $h1$ is $(\lambda y.\, z)\ x$ and e is z then the call returned is $\mathbf{let}\ x = x\ \mathbf{in}\ h1\ x\ z$. Ideally the memoiser will also mutate the old promise to record which variables were dead so that later tiebacks to the promise will be able to use this same trick.

[3] If normal Haskell functions can be thought of as having a single implicit continuation argument in addition to their normal value argument, then *error* can be thought of as being unusual in having a value argument—the error message *String*—but no continuation argument.

[4] In order to be able to infer the type of possibly-nullary **case** expressions in our implementation, we made use of the fact that in GHC's abstract syntax trees, each **case** is already annotated with the type of its branches. An alternative solution to preserve type compatibility is to use the coercions and casts of System FC instead of a nullary **case**.

Another thing to look out for with this trick is that the dead stack suffix might contain some update frames, which cannot be dropped entirely without potentially causing some variables to be unbound. Thus, the replacement trivial stack should preserve any existing update frames, and replace all other types of frame with a nullary **case**. So for example, if we were driving the pushed state

$$\left\langle \mathit{msg} \mapsto \langle \epsilon \,|\, f\ n \,|\, \epsilon \rangle \,\middle|\, \mathit{error} \,\middle|\, \begin{array}{l} \bullet\ \mathit{Bool}, \bullet\ \mathit{msg}, \mathbf{case} \bullet \mathbf{of}\ \begin{array}{l} \mathit{True} \to \langle \epsilon \,|\, \mathit{fib}\ 100 \,|\, \epsilon \rangle \\ \mathit{False} \to \langle \epsilon \,|\, \mathit{fact}\ 100 \,|\, \epsilon \rangle \end{array}, \mathbf{update}\ n \end{array} \right\rangle$$

Then the final driven term would be

$$\begin{array}{l} \mathbf{let}\ n = \mathbf{case}\ \mathit{error}\ \mathit{Bool}\ \mathit{msg}\ \mathbf{of}\ \epsilon \\ \mathbf{in}\ n \end{array}$$

In practice, we found that exploiting known-bottoming functions in this way can save the supercompiler a lot of work: in particular, the size of supercompiler output for the *bernoulli* benchmark in the Nofib benchmark suite [Partain, 1993] fell by 16% when this optimisation was implemented.

Detecting more dead stack frames As described so far, the splitter is only able to determine that a stack suffix is dead if it occurs directly after a residualised saturated call to a bottoming function such as *error*. An extension of this idea, analogous to the treatment of dead heap bindings, allows more dead stack frames to be detected.

If we arrange that supercompilation of a state returns not only an optimised term but also an optional natural number indicating how many arguments need to be applied to the term before it diverges (zero if the term is already $\bot$), then when driving a pushed state such as

$$\left\langle \epsilon \,\middle|\, f \,\middle|\, \bullet\ x, \mathbf{case} \bullet \mathbf{of}\ \begin{array}{l} \mathit{True} \to \langle \epsilon \,|\, \mathit{error}\ \texttt{"True"} \,|\, \epsilon \rangle \\ \mathit{False} \to \langle \epsilon \,|\, \mathit{error}\ \texttt{"False"} \,|\, \epsilon \rangle \end{array}, \mathbf{update}\ x, \mathbf{case} \bullet \mathbf{of}\ (a, b) \to \langle \epsilon \,|\, e \,|\, \epsilon \rangle \right\rangle$$

We will be able to detect that both nested states in the residual **case** diverge, and thus that all stack frames after the **case** frame are dead. Assuming that the two nested states turn into calls to *h*-functions named *h1* and *h2*, the resulting driven term would be

$$\begin{array}{l} \mathbf{let}\ x = \mathbf{case}\ f\ x\ \mathbf{of}\ \mathit{True} \to \mathit{h1}; \mathit{False} \to \mathit{h2} \\ \mathbf{in\ case}\ x\ \mathbf{of}\ \epsilon \end{array}$$

In practice, the number of programs that benefit from propagating information about bottomness out of recursive invocations of the supercompiler is very small, so our implementation does not use this extension to the scheme.

4.5 Correctness of the splitter

Recall Definition 3.8.2, which defined what it meant for *split* to be correct:

Definition 3.8.2 (Splitter correctness). *The split function is correct if:*

$$\begin{array}{l} \forall \mathcal{S}'. {}^{\checkmark}\mathit{delay}\ \mathcal{S}' \mathrel{\underset{\sim}{\triangleright}} \mathit{checkScpM}(\overline{p}, \mathit{opt}\ \mathcal{S}') \\ \implies \mathit{delay}\ \mathcal{S} \mathrel{\underset{\sim}{\triangleright}} \mathit{checkScpM}(\overline{p}, \mathit{split}\ \mathit{opt}\ \mathcal{S}) \end{array}$$

It is straightforward to prove that *split* obeys this property as long as the following conjecture about $\widehat{\mathit{push}}$ is true:

Conjecture 4.5.1 (Pushing is an improvement up to delay). *For all N, $\mathcal{S}$, $\mathit{delay}\ \mathcal{S} \underset{\sim}{\triangleright} \mathit{de\mathring{l}ay}(\widehat{\mathit{push}}(\{\bullet\} \cup N)(\mathcal{S}))$, where:*

$$\mathit{de\mathring{l}ay}\left\langle \overline{x{:}\tau \mapsto \mathring{e}_x} \,\middle|\, \mathring{e} \,\middle|\, \overline{\mathring{\kappa}} \right\rangle \;=\; \left\langle\!\!\left\langle \overline{x{:}\tau \mapsto \mathit{de\mathring{l}ay}_e(\mathring{e}_x)} \,\middle|\, \mathit{de\mathring{l}ay}_e(\mathring{e}) \,\middle|\, \overline{\mathit{de\mathring{l}ay}_\kappa(\mathring{\kappa})} \right\rangle\!\!\right\rangle$$

$$\begin{aligned}
\mathit{de\mathring{l}ay}_e(x) &= x \\
\mathit{de\mathring{l}ay}_e(\lambda x{:}\tau.\, \mathcal{S}) &= \lambda x{:}\tau.\, {}^{\checkmark}\mathit{delay}\ \mathcal{S} \\
\mathit{de\mathring{l}ay}_e(\mathbf{C}\ \overline{\tau}\ \overline{x}) &= \mathbf{C}\ \overline{\tau}\ \overline{x} \\
\mathit{de\mathring{l}ay}_e(\Lambda\alpha{:}\kappa.\, \mathcal{S}) &= \Lambda\alpha{:}\kappa.\, {}^{\checkmark}\mathit{delay}\ \mathcal{S} \\
\mathit{de\mathring{l}ay}_e(\mathcal{S}) &= {}^{\checkmark}\mathit{delay}\ \mathcal{S}
\end{aligned}$$

$$\begin{aligned}
\mathit{de\mathring{l}ay}_\kappa(\mathbf{update}\ x{:}\tau) &= \mathbf{update}\ x{:}\tau \\
\mathit{de\mathring{l}ay}_\kappa(\bullet\ x) &= \bullet\ x \\
\mathit{de\mathring{l}ay}_\kappa(\bullet\ \tau) &= \bullet\ \tau \\
\mathit{de\mathring{l}ay}_\kappa(\mathbf{case} \bullet \mathbf{of}\, \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau} \to \mathcal{S}}) &= \mathbf{case} \bullet \mathbf{of}\, \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau} \to {}^{\checkmark}\mathit{delay}\ \mathcal{S}}
\end{aligned}$$

We conjecture that our $\widehat{\mathit{push}}$ function does obey this property. We already argued that $\widehat{\mathit{push}}$ was an improvement (Section 4.3.4), but this is the first time we have considered the effect of delays. To justify that this is a plausible conjecture, we will prove the definition holds for one particular input state which exercises most interesting features of $\widehat{\mathit{push}}$, which we will call $\mathcal{S} \equiv \langle H \,|\, e \,|\, K \rangle$:

$$\left\langle a \mapsto \mathit{Just}\ b, b \mapsto \mathbf{let}\ d = \lambda x.\, x\ \mathbf{in}\ (\lambda d.\, d)\ d, c \mapsto f\ b \,\middle|\, f \,\middle|\, \bullet\ a, \mathbf{case} \bullet \mathbf{of}\ \begin{aligned}&\mathit{True} \to (\lambda y.\, y)\\ &\mathit{False} \to (\lambda z.\, x)\end{aligned}, \bullet\ c \right\rangle$$

We can see that:

$$\mathit{delay}\,\langle H \,|\, e \,|\, K \rangle \;=\; \begin{array}{l}
\mathbf{let}\ a = \mathit{Just}\ b \\
\quad\ b = {}^{\checkmark}(\mathbf{let}\ d = {}^{\checkmark}(\lambda x.\, x)\ \mathbf{in}\ (\lambda d.\, {}^{\checkmark}d)\ x) \\
\quad\ c = {}^{\checkmark}(f\ b) \\
\mathbf{in}\ (\mathbf{case}\, f\ a\ \mathbf{of}\ \begin{array}[t]{l} \mathit{True} \to {}^{\checkmark}(\lambda y.\, {}^{\checkmark}y) \\ \mathit{False} \to {}^{\checkmark}(\lambda z.\, {}^{\checkmark}x))\ c \end{array}
\end{array}$$

The pushed version of this state, $\mathring{S} \equiv \left\langle \mathring{H} \,\middle|\, \mathring{e} \,\middle|\, \mathring{K} \right\rangle$, is:

$$\left\langle a \mapsto \mathit{Just}\ b, b \mapsto \left\langle \epsilon \,\middle|\, \begin{array}{l}\mathbf{let}\ d = \lambda x.\, x \\ \mathbf{in}\ (\lambda d.\, d)\ d\end{array} \,\middle|\, \epsilon \right\rangle \,\middle|\, f \,\middle|\, \bullet\ a, \mathbf{case} \bullet \mathbf{of}\ \begin{array}[t]{l}\mathit{True} \to \langle c \mapsto f\ b \,|\, \lambda y.\, y \,|\, \bullet\ c \rangle \\ \mathit{False} \to \langle c \mapsto f\ b \,|\, \lambda z.\, x \,|\, \bullet\ c \rangle\end{array} \right\rangle$$

and therefore:

$$\mathit{de\mathring{l}ay}\left\langle \mathring{H} \,\middle|\, \mathring{e} \,\middle|\, \mathring{K} \right\rangle \;=\; \begin{array}{l}
\mathbf{let}\ a = \mathit{Just}\ b \\
\qquad b = {}^{\checkmark}(\mathbf{let}\ d = {}^{\checkmark}(\lambda x.\, x)\ \mathbf{in}\ (\lambda d.\, {}^{\checkmark}d)\ d) \\
\mathbf{in\ case}\, f\ a\ \mathbf{of}\ \begin{array}[t]{l} \mathit{True} \to {}^{\checkmark}(\mathbf{let}\ c = {}^{\checkmark}(f\ b)\ \mathbf{in}\ (\lambda y.\, {}^{\checkmark}y)\ c) \\ \mathit{False} \to {}^{\checkmark}(\mathbf{let}\ c = {}^{\checkmark}(f\ b)\ \mathbf{in}\ (\lambda z.\, {}^{\checkmark}x)\ c) \end{array}
\end{array}$$

Using the facts that transformations such as let-floating into a linear context and case-of-case are cost-equivalences, it is straightforward to see that the delayed, pushed state $\mathring{(\mathit{delay})}\left\langle \mathring{H} \,\middle|\, \mathring{e} \,\middle|\, \mathring{K} \right\rangle$ is an improvement on the delayed input $\mathit{delay}\,\langle H \,|\, e \,|\, K \rangle$.

There are two principal subtleties in this conjecture, which we discuss below.

Generalisation sets that do not include • Notice that Conjecture 4.5.1 requires that the generalisation set passed to $\widehat{push}$ contains •. If the set does not contain this then the conjecture is not true. For example, consider the generalisation set $N = \{x\}$ and the state $\langle x \mapsto \mathit{True} \,|\, f\ x \,|\, \epsilon \rangle$, which is pushed to $\langle x \mapsto \mathit{True} \,|\, \langle \epsilon \,|\, f\ x \,|\, \epsilon \rangle \,|\, \epsilon \rangle$. The corresponding delayed versions are:

$$\begin{aligned} \mathit{delay}\, \langle x \mapsto \mathit{True} \,|\, f\ x \,|\, \epsilon \rangle &= \textbf{let}\ x = \mathit{True}\ \textbf{in}\ f\ x \\ \overset{\circ}{\mathit{delay}}\, \langle x \mapsto \mathit{True} \,|\, \langle \epsilon \,|\, f\ x \,|\, \epsilon \rangle \,|\, \epsilon \rangle &= \textbf{let}\ x = \mathit{True}\ \textbf{in}\ {}^{\checkmark}(f\ x) \end{aligned}$$

It is clear that the pushed, delayed state is not an improvement of the input.

This restriction of the conjecture does not affect our proof of the correctness of *split*, since *split* always uses the generalisation set {•}, but it does mean that we cannot use our conjecture unmodified to prove facts about uses of $\widehat{push}$ for generalisation purposes in Section 6.2.2.

Heap-bound values Because of the fact that $\mathit{delay}_\bullet$ does not delay values, the splitter has to "eagerly" split heap-bound values. For example, given the input state

$$\langle x \mapsto \mathit{Just}\ y, y \mapsto \mathit{fib}\ 100 \,|\, x \,|\, \epsilon \rangle$$

We cannot return the pushed state $\langle x \mapsto \langle y \mapsto \mathit{fib}\ 100 \,|\, \mathit{Just}\ y \,|\, \epsilon \rangle \,|\, x \,|\, \epsilon \rangle$ from $\widehat{push}$, since:

$$\begin{aligned} \mathit{delay}\, \langle x \mapsto \mathit{Just}\ y, y \mapsto \mathit{fib}\ 100 \,|\, x \,|\, \epsilon \rangle &= \textbf{let}\ x = \mathit{Just}\ y; y = {}^{\checkmark}(\mathit{fib}\ 100)\ \textbf{in}\ x \\ \overset{\circ}{\mathit{delay}}\, \langle x \mapsto \langle y \mapsto \mathit{fib}\ 100 \,|\, \mathit{Just}\ y \,|\, \epsilon \rangle \,|\, x \,|\, \epsilon \rangle &= \textbf{let}\ x = {}^{\checkmark}(\textbf{let}\ y = {}^{\checkmark}(\mathit{fib}\ 100)\ \textbf{in}\ \mathit{Just}\ y)\ \textbf{in}\ x \end{aligned}$$

Once again it is clear that the pushed, delayed state is not an improvement of the input. Instead, our $\widehat{push}$ function eagerly splits the heap-bound *Just y* value and returns the pushed state $\langle x \mapsto \mathit{True}, y \mapsto \langle \epsilon \,|\, \mathit{fib}\ 100 \,|\, \epsilon \rangle \,|\, x \,|\, \epsilon \rangle$.

If we did not eagerly split heap-bound values in the splitter then the supercompiler would be incorrect. For example, starting with the state $\mathcal{S}_0 = \langle x \mapsto \mathit{True} \,|\, x \,|\, \epsilon \rangle$ the splitter without eager value splitting could recursively drive $\mathcal{S}_1 = \langle \epsilon \,|\, \mathit{True} \,|\, \epsilon \rangle$. Because $\mathcal{S}_0$ is cost equivalent to $\mathcal{S}_1$ up to delay, the *match* function would be justified in tying back the supercompilation process for $\mathcal{S}_1$ to the *h*-function for $\mathcal{S}_0$[5], like so:

$$\begin{aligned} &\textbf{let}\ \mathit{h0} = \textbf{let}\ x = h\ \textbf{in}\ x \\ &\textbf{in}\ \mathit{h0} \end{aligned}$$

4.6 The interaction of call-by-need and recursive let

The combination of call-by-need and recursive **let** considerably complicates the implementation of *split*. In the absence of either one of these features, if the stack contains a **case** frame then the splitter may unconditionally push the entire tail of the stack after that frame into the residualised version of the **case** frame. For example, the input:

$$\langle \epsilon \,|\, h \,|\, \bullet\ z, \textbf{case} \bullet \textbf{of}\ \mathit{True} \to f; \mathit{False} \to g, K \rangle$$

May be transformed to the pushed state:

$$\langle \epsilon \ \,|\, \ h \ \,|\, \ \bullet\ z, \textbf{case} \bullet \textbf{of}\ \mathit{True} \to \langle \epsilon \,|\, f \,|\, K \rangle ; \mathit{False} \to \langle \epsilon \,|\, g \,|\, K \rangle \rangle$$

[5] The *match* function we define in this thesis is not in fact powerful enough to achieve this tieback, but it is easier to make the splitter eagerly split heap-bound values than it is to find an alternative property for *match* which is strong enough to rule out these sorts of tiebacks.

This will never duplicate work or leave any variables unbound regardless of the form of K or whether it contains any update frames. If we are working with a language with recursive **let** then this is no longer true because we cannot transform the input state:

$$\langle \epsilon \,|\, h \,|\, \bullet\ z, \mathbf{case} \bullet \mathbf{of}\ \mathit{True} \to f; \mathit{False} \to g, \mathbf{update}\ z \rangle$$

Into the pushed state:

$$\langle \epsilon \mid h \mid \bullet\ z, \mathbf{case} \bullet \mathbf{of}\ \mathit{True} \to \langle \epsilon \,|\, f \,|\, \mathbf{update}\ z \rangle ; \mathit{False} \to \langle \epsilon \,|\, g \,|\, \mathbf{update}\ z \rangle \rangle$$

Since it leaves the occurrence of z in $\bullet\ z$ unbound at the top level. The only obvious way to bind it at the top level would be to duplicate work:

$$\left\langle z \mapsto \mathbf{case}\ h\ z\ \mathbf{of}\ \mathit{True} \to f; \mathit{False} \to g \;\middle|\; h \;\middle|\; \begin{array}{r} \bullet\ z, \mathbf{case} \bullet \mathbf{of}\ \mathit{True} \to \langle \epsilon \,|\, f \,|\, \mathbf{update}\ z \rangle \\ \mathit{False} \to \langle \epsilon \,|\, g \,|\, \mathbf{update}\ z \rangle \end{array} \right\rangle$$

This problem does not arise in a call-by-name or call-by-value programming language because such operational semantics for such languages have no need to ever create update frames, which are the source of the problem. Furthermore, the problem does not even arise in a call-by-need without recursive **let** but *with* negative recursion in data constructors (which allows the formation of fixed points). The reason for this is that fixed points created via negative recursion do not share work: i.e. if the fixed point $\mathit{fibs} = 1 : 1 : \mathit{zipWith}\ (+)\ \mathit{fibs}\ (\mathit{tail}\ \mathit{fibs})$ is created via negative recursion then it will do as much work as the fixed point $\mathit{fibs} = \lambda eta.\, 1 : 1 : \mathit{zipWith}\ (+)\ (\mathit{fibs}\ ())\ (\mathit{tail}\ (\mathit{fibs}\ ()))$.

If we were not working with a call-by-need language with recursive **let** then we could make two key simplifications to the algorithms above:

- The *mark* function would need only to consider acyclic sharing-graphs, simplifying the code considerably.
- We can assume that the stack frames with indexes $j \geq i$ will all be in the maximum marking as long as there is an edge $i_{\mathit{pred}} \overset{\checkmark}{\to} i$, where i_{pred} is unmarked and:

  $$i_{\mathit{pred}} = \begin{cases} \bullet & i = 0 \\ i - 1 & \text{otherwise} \end{cases}$$

  This invariant is sufficiently informative that it becomes convenient to omit stack frames entirely from the sharing graph and instead just perform a simple test in $\mathit{prepresid}_K$ whether to push or residualise stack frames (stack frames should be pushed if the previous stack frame was, or if the previous stack frame was a residualised **case** frame).

Furthermore, if we are working in a call-by-name language then there is no work sharing to be preserved and the *mark* function can be replaced by the trivial function which marks all nodes in the sharing graph. Likewise, if we are working in a call-by-value language then the heap is guaranteed to contain only cheap terms, and *mark* can be replaced by the same trivial function.

Chapter 5

Call-by-need matching

In this section we will define the *match* function used in the supercompiler's memoisation module (Section 3.4) for testing the syntactic equivalence of states (up to renaming of free variables). As we are supercompiling a call-by-need language, our *match* function needs to be aware of work sharing. For example, *match* should not claim that the following two states are equivalent:

$$\mathcal{S}_0 = \textbf{let}\ a = \mathit{fib}\ y; b = \mathit{fib}\ y\ \textbf{in}\ (a, b) \quad \mathcal{S}_1 = \textbf{let}\ a = \mathit{fib}\ y\ \textbf{in}\ (a, a)$$

If *match* did make such a claim, the supercompiler might tie back the newly-driven state $\mathcal{S}_1$ to an existing promise for $\mathcal{S}_0$, thus duplicating work in the supercompiler's output. Defining a sharing-aware *match* function is surprisingly tricky.

The approach we take to defining *match* is to define it in terms of another function, *msg*, which implements a so-called most-specific generalisation (MSG). The principal advantage of this approach is that the more-general *msg* function is independently useful for implementing generalisation in our supercompiler (Section 6.2).

By using *msg* to implement both *match* and generalisation in the supercompiler, we reduce the amount of code in the supercompiler, and hence the number of potential bugs and supercompiler implementation effort. We will also be able to reuse work done to prove correctness of the MSG in order to prove the correctness of the matcher.

5.1 Most-specific generalisation

The concept of MSG is a familiar one in the supercompilation literature [Mitchell and Runciman, 2008; Leuschel and Martens, 1996; Sørensen and Glück, 1995], and has its roots in some of the earliest work on machine deduction [Plotkin, 1969; Reynolds, 1969; Robinson, 1965]. The idea is that the MSG of two objects (typically terms, but in our supercompiler we will consider the MSG of states) is another object that incorporates all of the structure that is common to the two input terms, but remaining sufficiently general such that the object can be instantiated to yield either of the two inputs.

More concretely, in an untyped call-by-name language, the most specific generalisation of two terms e_0 and e_1 would be another term e such that there exist (variable to term) substitutions θ_0 and θ_1 such that $e\theta_0 = e_0$ and $e\theta_1 = e_1$, where furthermore the substitutions are as small as possible—i.e. as much common information as possible has been incorporated into the common term e rather than the individual substitutions.

Some examples of MSG in this setting would be:

e_0	e_1	θ_0	e	θ_1
$\mathit{map}\ f\ \mathit{xs}$	$\mathit{map}\ f\ (\mathit{filter}\ g\ \mathit{xs})$	$\{h \mapsto f, \mathit{ys} \mapsto \mathit{xs}\}$	$\mathit{map}\ h\ \mathit{ys}$	$\{h \mapsto f, \mathit{ys} \mapsto \mathit{filter}\ g\ \mathit{xs}\}$
$\lambda x.\, \mathit{fib}\ x$	$\lambda x.\, \mathit{plus}\ z\ x$	$\{f \mapsto \mathit{fib}\}$	$\lambda y.\, f\ y$	$\{f \mapsto \mathit{plus}\ z\}$
$\mathit{map}\ f\ \mathit{xs}$	$\begin{array}{r}\textbf{case}\ x\ \textbf{of}\ \mathit{True} \to \mathit{False}\\ \mathit{False} \to \mathit{True}\end{array}$	$\{y \mapsto \mathit{map}\ f\ \mathit{xs}\}$	y	$\left\{y \mapsto \begin{array}{r}\textbf{case}\ x\ \textbf{of}\ \mathit{True} \to \mathit{False}\\ \mathit{False} \to \mathit{True}\end{array}\right\}$

In the call-by-need setting, we need a more complex MSG definition than we had in the call-by-name case. Precisely, we wish to define a function *msg* which obeys the following property:

Definition 5.1.1 (MSG correctness).

$$\begin{array}{l} \mathit{msg}(\mathcal{S}_0, \mathcal{S}_1) = (\langle {H_0}' \,|\, \theta_0 \,|\, {K_0}' \rangle, \mathcal{S}, \langle {H_1}' \,|\, \theta_1 \,|\, {K_1}' \rangle) \\ \quad \Longrightarrow \quad \forall j. \mathit{rebuild}\ \mathcal{S}_j \mathrel{\overset{\lhd\rhd}{\sim}} \langle\!| {H_j}' \,|\, (\mathit{rebuild}\ \mathcal{S})\theta_j \,|\, {K_j}' |\!\rangle \end{array}$$

Furthermore, we expect that all of ${H_j}'$ and ${K_j}'$ are as small as possible given this definition, in order that this defines a *most-specific* generalisation. Note that we can only make a "best effort" towards minimising the individual heap/stack, because deciding cost equivalence in general requires solving the halting problem: consider what common heap should be returned by $\mathit{msg}(\langle x \mapsto \mathit{tauto1}, \mathit{id} \mapsto \lambda x.\, x \,|\, \mathit{id} \,|\, \bullet\ x \rangle, \langle x \mapsto \mathit{tauto2}, \mathit{id} \mapsto \lambda x.\, x \,|\, \mathit{id} \,|\, \bullet\ x \rangle)$, where *tauto1* and *tauto2* encode arbitrary tautologies.

Also worth noting are that:

- The abuse of notation $\langle H \,|\, \theta \,|\, K \rangle$ places a substitution where you would normally expect an expression.
- Substitutions θ map type variables to types and term variables to *variables* (this is different from our opening example with a call-by-name MSG, but consistent with our treatement of substitutions everywhere else in the thesis).

As a simple example, consider the two states

$$\langle f \mapsto \lambda a.\, (a, a), x \mapsto 2 \,|\, f \,|\, \bullet\ x \rangle \quad \langle g \mapsto \lambda b.\, (b, b), c \mapsto 2 \,|\, g \,|\, \bullet\ y, \textbf{case} \bullet \textbf{of}\ (a, b) \to h\ a\ b\ c \rangle$$

We expect the MSGed common state to be

$$\langle h \mapsto \lambda a.\, (a, a) \,|\, h \,|\, \bullet\ z \rangle$$

Where furthermore:

$$\begin{array}{rclrcl} {H_0}' & = & x \mapsto 2 & {H_1}' & = & c \mapsto 2 \\ {K_0}' & = & \epsilon & {K_1}' & = & \textbf{case} \bullet \textbf{of}\ (a, b) \to h\ a\ b\ c \\ {\theta_0}' & = & \{z \mapsto x\} & {\theta_1}' & = & \{z \mapsto y\} \end{array}$$

Notice that matching these two states as *terms* would have failed because the rebuilding of the left hand state has an application as the outermost syntax node (excluding the enclosing **let**), whereas the right hand state has a **case** as the outermost syntax node of its rebuilding.

This example already shows one of the reasons why we insist on cost equivalence instead of syntactic equivalence between the input states and the instantiated, generalised versions discovered by *msg*: we expect that syntactic equivalence will hardly ever hold. For our example, we can see that

$$\langle\!\langle f \mapsto \lambda a.\,(a,a), x \mapsto 2 \,|\, f \,|\, \bullet\ x \rangle\!\rangle \;=\; \begin{array}{l} \textbf{let}\, f = \lambda a.\,(a,a) \\ \quad\;\; x = 2 \\ \textbf{in}\, f\ x \end{array}$$

$$\langle\!\langle H_0{}' \,|\, \langle\!\langle h \mapsto \lambda a.\,(a,a) \,|\, h \,|\, \bullet\ z \rangle\!\rangle\, \theta_0{}' \,|\, K_0{}' \rangle\!\rangle \;=\; \begin{array}{l} \textbf{let}\ x = 2 \\ \textbf{in}\ \textbf{let}\ h = \lambda a.\,(a,a) \\ \quad\ \textbf{in}\ h\ x \end{array}$$

and

$$\langle\!\langle g \mapsto \lambda b.\,(b,b), c \mapsto 2 \,|\, g \,|\, \bullet\ y, \textbf{case} \bullet \textbf{of}\ (a,b) \to h\ a\ b\ c \rangle\!\rangle \;=\; \begin{array}{l} \textbf{let}\ g = \lambda b.\,(b,b) \\ \quad\;\; c = 2 \\ \textbf{in}\ \textbf{case}\ g\ y\ \textbf{of}\ (a,b) \to h\ ab\ c \end{array}$$

$$\langle\!\langle H_1{}' \,|\, \langle\!\langle h \mapsto \lambda a.\,(a,a) \,|\, h \,|\, \bullet\ z \rangle\!\rangle\, \theta_1{}' \,|\, K_1{}' \rangle\!\rangle \;=\; \begin{array}{l} \textbf{let}\ c = 2 \\ \textbf{in}\ \textbf{case}\ (\textbf{let}\ h = \lambda a.\,(a,a) \\ \qquad\qquad\ \textbf{in}\ h\ y)\ \textbf{of}\ (a,b) \to h\ a\ b\ c \end{array}$$

Note that in each case the instantiated, generalised states are cost-equivalent to their respective input states, but not syntactically equivalent.

There are two further subtleties to our definition of *msg*:

- Variable occurrences in the generalised state $\mathcal{S}$ may be bound by the individual heaps/stacks $H_j{}'$ and $K_j{}'$. In contrast, occurrences of variables in the individual heap/stacks may *not* be bound by binders in the common heap/stack.
- Our insistence on cost equivalence means that our MSG must take into account the call-by-need cost model.

The combination of these factors leads to some consequences that may be unintuitive for those used to a standard call-by-name MSG. Consider the two input states, which will refer to the left and right-hand states respectively:

$$\langle a \mapsto f\ y \,|\, (a,a) \,|\, \epsilon \rangle \quad \langle b \mapsto f\ y, c \mapsto f\ y \,|\, (b,c) \,|\, \epsilon \rangle$$

Note that the components of each pair are all exactly the same: a call to an unknown function f. A call-by-name MSG could yield either one of these common generalised states

$$\langle d \mapsto f\ y, e \mapsto f\ y \,|\, (d,e) \,|\, \epsilon \rangle \quad \langle d \mapsto f\ y \,|\, (d,d) \,|\, \epsilon \rangle$$

However, neither of these answers are appropriate for a call-by-need setting. The first answer is unsuitable because it has less work sharing than the left-hand input due to its duplication of the computation of the a heap binding. The second answer is unsuitable because it has more work sharing than the right-hand input due to it commoning up the work of the b and c heap bindings. A call-by-need MSG algorithm must instead return the less-generalised state

$$\langle \epsilon \,|\, (d,e) \,|\, \epsilon \rangle$$

with individual structure respectively

$$\langle a \mapsto f\ y \,|\, \{d \mapsto a, e \mapsto a\} \,|\, \epsilon \rangle \quad \langle a \mapsto f\ y, b \mapsto f\ y \,|\, \{d \mapsto a, e \mapsto b\} \,|\, \epsilon \rangle$$

This shows that the choice of call-by-need has real consequences for the definition of the MSG.

For the purposes of defining the MSG algorithm, we ignore all tags. When extending the algorithm to deal with tags, we have to confront the fact that there is no mechanism for abstracting over the tags of a term, and so the common information returned by *msg* has to be built with either the left tags *or* the right tags. In our implementation, we always use the tags from the right argument and then ensure that whenever we call *msg* we do so with the state whose tags we wish to preserve in the right argument.

5.2 Implementing a matcher from MSG

Recall our description (Section 3.8) of what it means for *match* to be correct:

Definition 3.8.1 (Matcher correctness)**.** *The match function is correct if it succeeds only when the inputs are cost-equivalent* up to delay*:*

$$\mathit{match}\ \mathcal{S}_0\ \mathcal{S}_1\ =\ \mathit{Just}\ \theta \implies (\mathit{delay}\ \mathcal{S}_0)\theta \underset{\sim}{\lhd\!\rhd} \mathit{delay}\ \mathcal{S}_1$$

Given *msg*, we can now define a suitable *match* function using a suitable *msg* function:

$$\begin{aligned} \mathit{match}(\mathcal{S}_0, \mathcal{S}_1) &= \theta \\ \textbf{where}\ \ \langle \epsilon \,|\, \theta \,|\, \epsilon \rangle &= \mathit{instanceMatch}(\mathcal{S}_0, \mathcal{S}_1) \end{aligned}$$

$$\begin{aligned} \mathit{instanceMatch}(\mathcal{S}_0, \mathcal{S}_1) &= \langle {H_1}' \,|\, \theta \,|\, {K_1}' \rangle \\ \textbf{where}\ \ (\langle \epsilon \,|\, \theta_0 \,|\, \epsilon \rangle, \_, \langle {H_1}' \,|\, \theta_1 \,|\, {K_1}' \rangle) &= \mathit{msg}(\mathcal{S}_0, \mathcal{S}_1) \\ \theta &= {\theta_0}^{-1} \circ \theta_1 \end{aligned}$$

Where we make use of substitution inversion, θ^{-1}:

$$\theta^{-1} = \begin{cases} \{y \mapsto x \,|\, x \mapsto y \in \theta\} \cup \{\beta \mapsto \alpha \,|\, \alpha \mapsto \beta \in \theta\} & \theta \text{ injective } \wedge \forall \alpha \mapsto \tau \in \theta. \exists \beta. \tau \equiv \beta \\ \textbf{fail} & \text{otherwise} \end{cases}$$

Note that since both *msg* and substitution inversion are partial functions, *match* is also partial and fails in the case where the second input state cannot be matched against the first. The fact that this *match* function succeeds as often as you would expect it to (in particular, it always succeeds if the two argument states are α-equivalent) depends crucially on the fact that *msg* implements a *most specific* generalisation.

Note also that substitution inversion checks that the input substitution is injective. In fact, in our application the input substitution θ_0 is always surjective in that the range exhaustively covers the (reachable) free variables of the input state $\mathcal{S}_0$, so this check is sufficient to establish bijectivity.

An example where θ_0 is not injective is if we were matching the two states $\langle \epsilon \,|\, (a, a) \,|\, \epsilon \rangle$ and $\langle \epsilon \,|\, (b, c) \,|\, \epsilon \rangle$, in which case we would have the non-injective $\theta_0 = \{d \mapsto a, e \mapsto a\}$ (and $\theta_1 = \{d \mapsto b, e \mapsto c\}$). Our refusal to invert these non-injective renamings reflects the fact that there is *no* way to rename the term (a, a) to obtain the term (b, c).

It is important to note that as *msg* is formulated in Section 5.3, the substitution returned by *msg* will never contain type/term variables in its domain which are *not* free in the common state returned by *msg*. It is important that it doesn't, since such "dead" substitutions can cause the invertability check used by *match* to fail. For example, if we were to find the MSG of $\langle x \mapsto \mathit{True} \,|\, (x, x) \,|\, \epsilon \rangle$ and $\langle y \mapsto \mathit{True}, z \mapsto \mathit{True} \,|\, (y, z) \,|\, \epsilon \rangle$ one answer which satisifies Definition 5.1.1 is $\theta_0 = \{a \mapsto x, b \mapsto x\}$ and $\theta_1 = \{a \mapsto y, b \mapsto z\}$, with common state $\langle a \mapsto \mathit{True}, b \mapsto \mathit{True} \,|\, (a, b) \,|\, \epsilon\rangle$ (note that a and b are not free variables of this state). In this example, θ_0 *is* non-injective, but only due to the dead substitutions for a and b. To avoid spurious failures of this kind, our definition of *msg* does not return substitutions for dead variables in either θ_0 or θ_1.

5.2.1 Positive information and non-injective substitutions

Although our *match* fails if θ_0 is non-injective, it *can* succeed if θ_1 is non-injective on term[1] variables, which may be surprising. This means that *match* succeeds in matching the terms $\mathit{xor}\ a\ b$ and $\mathit{xor}\ c\ c$ (returning a non-injective $\theta = \{d \mapsto c, e \mapsto c\}$) where:

$$\begin{aligned} \mathit{xor}\ x\ y = \mathbf{case}\ x\ \mathbf{of}\ & \mathit{True} \to \mathbf{case}\ y\ \mathbf{of}\ \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True} \\ & \mathit{False} \to \mathbf{case}\ y\ \mathbf{of}\ \mathit{True} \to \mathit{True}; \mathit{False} \to \mathit{False} \end{aligned}$$

If we were attempting to make maximum use of positive information propagation, we should not allow the supercompiler's memoiser to make use of the result of such a *match*. If we do not allow such a non-injective tieback, supercompiling $(\mathit{xor}\ a\ b, \mathit{xor}\ c\ c)$ produces supercompiled output isomorphic to

$$\begin{aligned} & \mathbf{let}\ \mathit{h0}\ a\ b = \mathbf{case}\ a\ \mathbf{of}\ \mathit{True} \to \mathbf{case}\ b\ \mathbf{of}\ \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True} \\ & \qquad\qquad\qquad\qquad\quad \mathit{False} \to \mathbf{case}\ b\ \mathbf{of}\ \mathit{True} \to \mathit{True}; \mathit{False} \to \mathit{False} \\ & \quad\ \ \mathit{h1}\ c \quad = \mathbf{case}\ c\ \mathbf{of}\ \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{False} \\ & \mathbf{in}\ (\mathit{h0}\ a\ b, \mathit{h1}\ c) \end{aligned}$$

whereas if we allow non-injective tiebacks, supercompiling the same term will (depending on the order of supercompilation of child states) either yield the above output, or the suboptimal output

$$\begin{aligned} & \mathbf{let}\ \mathit{h0}\ a\ b = \mathbf{case}\ a\ \mathbf{of}\ \mathit{True} \to \mathbf{case}\ b\ \mathbf{of}\ \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True} \\ & \qquad\qquad\qquad\qquad\quad \mathit{False} \to \mathbf{case}\ b\ \mathbf{of}\ \mathit{True} \to \mathit{True}; \mathit{False} \to \mathit{False} \\ & \mathbf{in}\ (\mathit{h0}\ a\ b, \mathit{h0}\ c\ c) \end{aligned}$$

Our implementation does not in fact reject non-injective tiebacks, for two reasons:

1. Positive information propagation is not important to most of the optimisations that the supercompiler achieves (Section 7.1.2), so losing positive information occasionally is not a particular problem.

2. When there exists a non-injective substitution from a previously promised state to the current state, it is often the case that the previous and the new state have the same tag-bags.

   In this situation, failing to tie back in the memoiser is disastrous because it causes the termination test to fail (assuming the previous state is still on the *sc* stack) and the supercompiler has to use *split* to throw information away. Tying back a bit too eagerly in some cases is well worth it if we can avoid such *split*s.

   This reason would not be so important if either one of these two changes were made:

   (a) If the termination criteria were stronger, so that it would not trigger on pairs of states which have a renaming relationship between them — even if their tag-bags would normally cause the termination check to fail.

[1] None of the discussion about injectivity in this section applies to type variables, since positive information propagation does not apply to them.

(b) If the memoiser only failed to tie back using a non-injective substitution if the promise it was tying back to was not an ancestor (i.e. had already been fulfilled), and thus would not be present in the current *History*.

Our implementation does not incorporate either of these suggestions, but it does incorporate a generalisation-based mechanism to avoid such *split*s (Section 6.2.5).

5.2.2 Correctness of *match*

It is straightforward to see that if the property we required of *msg* (Definition 5.1.1) holds, then *match* has the property:

$$\mathit{match}\ \mathcal{S}_0\ \mathcal{S}_1\ =\ \mathit{Just}\ \theta \implies (\mathit{rebuild}\ \mathcal{S}_0)\theta \mathrel{\overset{\lhd\!\rhd}{\sim}} \mathit{rebuild}\ \mathcal{S}_1$$

However, this is not quite the same as the property the supercompiler requires of *match*. We conjecture (but do not prove) that for the particular implementation of *msg* described in this chapter, *match* does indeed meet the stricter requirements of Definition 3.8.1. An interesting technical point here is that *match* is the whole reason that delay_h (Figure 3.3) treats values specially. If it did not treat values specially, so $\mathit{delay}_h(e) = \mathit{delay}_e(e)$, then we would have that

$$\begin{aligned} \mathit{delay}\,\langle x \mapsto \mathit{True} \,|\, (x,x) \,|\, \epsilon\rangle &= \langle x \mapsto^{\checkmark} \mathit{True} \,|\, (x,x) \,|\, \epsilon\rangle \\ \mathit{delay}\,\langle y \mapsto \mathit{True}, z \mapsto \mathit{True} \,|\, (y,z) \,|\, \epsilon\rangle &= \langle x \mapsto^{\checkmark} \mathit{True}, z \mapsto^{\checkmark} \mathit{True} \,|\, (y,z) \,|\, \epsilon\rangle \end{aligned}$$

and therefore $\mathit{delay}\,\langle x \mapsto \mathit{True} \,|\, (x,x) \,|\, \epsilon\rangle \mathrel{\not\overset{\lhd\!\rhd}{\sim}} \mathit{delay}\,\langle y \mapsto \mathit{True}, z \mapsto \mathit{True} \,|\, (y,z) \,|\, \epsilon\rangle$, so to have the required property we would need that:

$$\mathit{match}\ \langle x \mapsto \mathit{True} \,|\, (x,x) \,|\, \epsilon\rangle\ \langle y \mapsto \mathit{True}, z \mapsto \mathit{True} \,|\, (y,z) \,|\, \epsilon\rangle = \mathit{Nothing}$$

In order to avoid this pessimisation and allow *match* to "reuse" values, we take care to define *delay* to use delay_h on heap bindings, so that we never delay heap-bound values.

It is important to note that even slightly more powerful matchers do not meet the requirements of the definition, and that violating the definition can easily cause the supercompiler to become incorrect. We will demonstrate this through two examples.

Denotational equality Firstly, consider what would happen if the matcher was allowed to relate any two terms which are *denotationally* equal, instead of insisting on terms which are cost-equivalent up to delay. This is perhaps the property that you would expect *match* to obey at first glance.

Now, imagine supercompiling the state $\mathcal{S}_0 = \langle \mathit{id} \mapsto \lambda x.\, x \,|\, \mathit{id} \,|\, \bullet\ x\rangle$, where for some reason the termination test in sc' has been triggered, so we don't *reduce* the obvious β-redex. Given this state, the splitter of Chapter 4 would recursively drive the state $\mathcal{S}_1 = \langle \epsilon \,|\, x \,|\, \epsilon\rangle$ (corresponding to the body of the λ). However, $\mathcal{S}_0$ and $\mathcal{S}_1$ have exactly the same (denotational) meaning, so a *match* that obeyed the proposed property would cause the memoiser to tie back to the promise corresponding to $\mathcal{S}_0$ when asked to supercompile $\mathcal{S}_1$. The resulting optimised program might look like this:

```
let h0 x = let f = λx. h0 x
           in f x
in h0
```

Clearly, this optimised term does not have the same meaning as the input: all it does is loop! So a *match* function whose only constraint on success is that the two inputs must be denotationally equivalent is clearly unsuitable.

$$\boxed{msg(\langle H_0 \,|\, e_0 \,|\, K_0 \rangle_{\Sigma_0|\Gamma_0}, \langle H_1 \,|\, e_1 \,|\, K_1 \rangle_{\Sigma_1|\Gamma_1}) = (\langle H_0 \,|\, \theta_0 \,|\, K_0 \rangle_{\Sigma_0|\Gamma_0}, \langle H \,|\, e \,|\, K \rangle_{\Sigma|\Gamma}, \langle H_1 \,|\, \theta_1 \,|\, K_1 \rangle_{\Sigma_1|\Gamma_1})}$$

```
$msg(\langle H_0 \,|\, e_0 \,|\, K_0 \rangle_{\Sigma_0|\Gamma_0}, \langle H_1 \,|\, e_1 \,|\, K_1 \rangle_{\Sigma_1|\Gamma_1}) \;=\; \mathbf{do}$
    $\mathrm{r}\boldsymbol{\theta_j} := \varnothing$
    $\mathrm{r}\boldsymbol{\Sigma} := \epsilon$
    $\mathrm{r}\boldsymbol{\Gamma} := \epsilon$
    $\mathrm{r}\boldsymbol{H} := \epsilon$
    $e \leftarrow msg_e(e_0, e_1)(\varnothing, \varnothing)$
    $\mathrm{r}\boldsymbol{K} := msg_K(K_0, K_1)$
    $\mathit{fixup}(e)(\varnothing, \varnothing)$
```

Figure 5.1: Call-by-need MSG

Cost equivalence Now consider what would happen if we required that the *match* function should only succeed if its two inputs are cost-equivalent:

$$match\ \mathcal{S}_0\ \mathcal{S}_1\ =\ Just\ \theta \implies (rebuild\ \mathcal{S}_0)\theta \mathrel{\underset{\sim}{\lhd\!\rhd}} rebuild\ \mathcal{S}_1$$

This is perhaps the second most intuitive property we could require of *match*, and it prevents obvious abuses where the supercompiler can tie back a reduced version of a term to an earlier, "more expensive" term. This property is in fact enough to prevent the previous error from occurring, because even though $\mathcal{S}_0$ and $\mathcal{S}_1$ are denotationally equivalent, they are not *cost*-equivalent and so will not be *match*ed, as can be seen by the following reduction sequences:

$$\begin{aligned} \langle x \mapsto True, id \mapsto \lambda x.\, x \,|\, id \,|\, \bullet\ x \rangle &\rightsquigarrow^1 \langle id \mapsto \lambda x.\, x, x \mapsto True \,|\, x \,|\, \epsilon \rangle \\ \langle x \mapsto True \,|\, x \,|\, \epsilon \rangle &\rightsquigarrow^0 \langle x \mapsto True \,|\, x \,|\, \epsilon \rangle \end{aligned}$$

However, as we hinted at in Section 3.4, even insisting on cost-equivalence is not enough to ensure that matching is safe. To see this, consider the supercompilation of $\mathcal{S}_2 = \langle x \mapsto True, y \mapsto x \,|\, y \,|\, \epsilon \rangle$. A sensible thing for *split* to do could be to recursively drive the state $\mathcal{S}_3 = \langle y \mapsto True \,|\, y \,|\, \epsilon \rangle$. Unfortunately, it can be shown that $\mathcal{S}_2$ is cost-equivalent to $\mathcal{S}_3$[2], so the supercompilation of $\mathcal{S}_3$ is apparently justified to tying back to the promise corresponding to $\mathcal{S}_2$, resulting in this output program:

let $h2 =$ **let** $y = h2$ **in** x **in** $h2$

Again, this is clearly not a correct result! We need an even stronger property on *match* to ensure correctness of supercompiler memoisation, which motivates the use of cost-equivalence up to delay.

5.3 MSG algorithm overview

We are now in a position to show how the *msg* function is actually defined. The top level of our MSG algorithm is presented in Figure 5.1. Our algorithm makes use of mutable state and as such is presented in a notation akin to Haskell's monadic **do** syntax that makes sequencing explicit. We make use of pattern matches and case analysis that is done

[2] To see this intuitively, observe that both DATAV and DATA are normalising reduction rules that are "free" for the purposes of cost-equivalence. We would not have a cost-equivalence in this example if we replaced *True* with $\lambda x.\, x$ because BETA-N is free but BETA is not.

top to bottom, and will also at times make use of implicit quantification over $j \in \{0,1\}$. For example, if we write:

$$\begin{aligned}\textbf{do } & \mathrm{r}\boldsymbol{X_j} := \varnothing \\ & \mathrm{r}\boldsymbol{\theta_j} := \mathrm{r}\boldsymbol{\theta_j} \cup \{x \mapsto x_j\}\end{aligned}$$

We really mean:

$$\begin{aligned}\textbf{do } & \mathrm{r}\boldsymbol{X_0} := \varnothing \\ & \mathrm{r}\boldsymbol{X_1} := \varnothing \\ & \mathrm{r}\boldsymbol{\theta_0} := \mathrm{r}\boldsymbol{\theta_0} \cup \{x \mapsto x_0\} \\ & \mathrm{r}\boldsymbol{\theta_1} := \mathrm{r}\boldsymbol{\theta_1} \cup \{x \mapsto x_1\}\end{aligned}$$

The mutable state we use in the algorithm consists of mutable cells $\mathrm{r}\boldsymbol{X}$ that can occur either as a "lvalue" to the left of the assignment operator := or as a "rvalue" reference in any other position.

The mutable cells we use are as follows:

- The substitution cells $\mathrm{r}\boldsymbol{\theta_0}$ and $\mathrm{r}\boldsymbol{\theta_1}$, which accumulate the substitutions with which the generalised state will be instantiated.

- The common environment/heap/stack cells $\mathrm{r}\boldsymbol{\Sigma}$, $\mathrm{r}\boldsymbol{\Gamma}$, $\mathrm{r}\boldsymbol{H}$ and $\mathrm{r}\boldsymbol{K}$ which accumulate the kinding/typing environment, heap bindings and stack frames (respectively) which are common to both of the inputs.

The code in *msg* is concerned with initialising these mutable cells, and then using their final contents to extract the final MSG. The calls to msg_K and msg_e do the actual work of computing an MSG, but the MSG they produce completely ignores any work duplication issues. Our final call to *fixup* prunes down the MSG so that it does not duplicate work, and constructs suitable instance environments/heaps/stacks to be returned from *msg* with the final common environments/heap/stack.

Note that many of the functions in our algorithm may fail, written in code as **fail**. Unless otherwise specified, failure propagates strictly through the computation of the MSG, and may even propagate as far upwards as the initial call to *msg*, which is why *msg* is only defined to be a partial function.

We assume that the original environments/heaps/stacks Σ_j, Γ_j, H_j and K_j are implicitly passed unchanged to all the functions recursively invoked by *msg*. This simply reduces syntactic noise as we can avoid explicitly passing the input heap/stack around as arguments to every other function.

5.4 Computing an initial MSG

The algorithm for term MSG is defined in Figure 5.2. The call $\mathit{msg}_e(e_0, e_1)(A, X)$ is responsible for finding the MSG of the two input terms under the assumption that the type variables in the set A and the term variables in the set X are "rigid" i.e. they may not be mentioned as a free variable in the range of the instantiating substitution/heap/stack.

As a simple example of why the rigid sets are useful, consider the MSG of $\lambda(x :: \mathit{Int}).\, x$ and $\lambda(x :: \mathit{Int}).\, y$. There is no non-trivial generalisation of these two terms because the there is no way for the non-capturing substitutions we work with to instantiate e.g. the free variable y in $\lambda(x :: \mathit{Int}).\, y$ to the bound variable x. This is reflected in our algorithm msg_e because the recursive call to MSG the terms under the λ-binders will take the form $\mathit{msg}_e(x, y)(\varnothing, \{x\})$ where the bound variable x is supplied in the rigid set X.

This issue will affect us when we are dealing with both type and term variable substitutions. In preparation, we define the notion of a non-capturing substitution:

$$\boxed{\mathit{msg}_e(e_0,e_1) = M(e)}$$

$$\begin{array}{rcl}
\mathit{msg}_e(x_0,x_1)(A,X) & = & \mathit{msg}_x(x_0,x_1)(X) \\
\mathit{msg}_e(\lambda x{:}\tau_0.\,e_0,\lambda x{:}\tau_1.\,e_1)(A,X) & = & \lambda x{:}\mathit{msg}_\tau(\tau_0,\tau_1)(A).\,\mathit{msg}_e(e_0,e_1)(A,X\cup\{x\}) \\
\mathit{msg}_e(e_0\ x_0,e_1\ x_1)(A,X) & = & \mathit{msg}_e(e_0,e_1)(X,A)\ \mathit{msg}_x(x_0,x_1)(X) \\
\mathit{msg}_e(\Lambda\alpha{:}\kappa.\,e_0,\Lambda\alpha{:}\kappa.\,e_1)(A,X) & = & \Lambda\alpha{:}\kappa.\,\mathit{msg}_e(e_0,e_1)(A\cup\{\alpha\},X) \\
\mathit{msg}_e(e_0\ \tau_0,e_1\ \tau_1)(A,X) & = & \mathit{msg}_e(e_0,e_1)(X,A)\ \mathit{msg}_\tau(\tau_0,\tau_1)(A) \\
\mathit{msg}_e(\mathbf{C}\ \overline{\tau_0}^n\ \overline{x_0}^m,\mathbf{C}\ \overline{\tau_1}^n\ \overline{x_1}^m)(A,X) & = & \mathbf{C}\ \overline{\mathit{msg}_\tau(\tau_0,\tau_1)(A)}^n\ \overline{\mathit{msg}_x(x_0,x_1)(X)}^m
\end{array}$$

$$\begin{array}{l}
\mathit{msg}_e(\textbf{case}\ e_0\ \textbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau_0}\ \rightarrow\ e_{\mathbf{C}0}}^n,\textbf{case}\ e_1\ \textbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau_1}\ \rightarrow\ e_{\mathbf{C}1}}^n)(A,X) \\
\qquad = \ \textbf{case}\ \mathit{msg}_e(e_0,e_1)(A,X)\ \textbf{of} \\
\qquad\qquad \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\mathit{msg}_\tau(\tau_0,\tau_1)(A\cup\overline{\alpha})}\ \rightarrow\ \mathit{msg}_e(e_{\mathbf{C}0},e_{\mathbf{C}1})(A\cup\overline{\alpha},X\cup\overline{x})}^n \\
\mathit{msg}_e(\textbf{let}\ \overline{x{:}\tau_0\ =\ e_{x0}}^n\ \textbf{in}\ e_0,\textbf{let}\ \overline{x{:}\tau_1\ =\ e_{x1}}^n\ \textbf{in}\ e_1)(A,X) \\
\qquad = \ \textbf{let}\ \overline{x{:}\mathit{msg}_\tau(\tau_0,\tau_1)(A)\ =\ \mathit{msg}_e(e_{x0},e_{x1})(A,X\cup\overline{x}))}\ \textbf{in}\ \mathit{msg}_e(e_0,e_1)(A,X\cup\overline{x}) \\
\qquad\qquad\qquad \mathit{msg}_e(e_0,e_1) \ = \ \textbf{fail}
\end{array}$$

$$\boxed{\mathit{msg}_x(x_0,x_1) = M(x)}$$

$$\mathit{msg}_x(x_0,x_1)(X) \ = \ \begin{cases}
x & x = x_0 = x_1 \wedge x \in X \\
\textbf{fail} & x_0 \in X \vee x_1 \in X \\
x & x \mapsto x_0 \in \mathrm{r}\boldsymbol{\theta_0} \wedge x \mapsto x_1 \in \mathrm{r}\boldsymbol{\theta_1} \\
\textbf{do}\ \ x \leftarrow \mathit{fresh} & \text{otherwise} \\
\qquad \tau \leftarrow \mathit{msg}_\tau(\mathit{typ}(x_0),\mathit{typ}(x_1))(\varnothing) & \\
\qquad \mathrm{r}\boldsymbol{\Sigma} := \mathrm{r}\boldsymbol{\Sigma}, x{:}\tau & \\
\qquad \mathrm{r}\boldsymbol{\theta_j} := \mathrm{r}\boldsymbol{\theta_j} \cup \{x \mapsto x_j\} & \\
\qquad \mathit{msg}'_x(x_0,x_1)(x{:}\tau) & \\
\qquad x &
\end{cases}$$

$$\boxed{\mathit{typ}(x_j) = \tau_j}$$

$$\mathit{typ}(x_j) \ = \ \begin{cases}
\tau_j & x_j{:}\tau_j \mapsto e_j \in H_j \\
\tau_j & \textbf{update}\ x_j{:}\tau_j \in K_j \\
\tau_j & x_j{:}\tau_j \in \Gamma_j
\end{cases}$$

$$\boxed{\mathit{msg}'_x(x_0,x_1)(x{:}\tau) = M()}$$

$$\mathit{msg}'_x(x_0,x_1)(x{:}\tau) \ = \ \begin{cases}
\mathrm{r}\boldsymbol{H} := \mathrm{r}\boldsymbol{H} \cup \{x{:}\tau \mapsto e\} & x_0{:}\tau_0 \mapsto e_0 \in H_0 \wedge x_1{:}\tau_1 \mapsto e_1 \in H_1 \\
 & \wedge e \leftarrow \mathit{msg}_e(e_0,e_1)(\varnothing,\varnothing) \\
() & \text{otherwise}
\end{cases}$$

Figure 5.2: Call-by-need term MSG

Definition 5.4.1 (Non-capturing substitutions)**.** A substitution θ is non-capturing for the set of type variables A if for all $\alpha \mapsto \tau \in \theta$, $\mathrm{ftvs}(\tau) \cap A = 0$. Likewise, a substitution θ is non-capturing for the set of term variables X if for all $x \mapsto y \in \theta$, $y \notin X$.

All of the substitutions in our *msg* algorithm are non-capturing with respect to the relevant X and A sets.

The function msg_e principally consists of a number of simultaneous case matches on its two arguments. Whenever we match binders together in the two arguments to msg_e, we implicitly α-rename both of the two binders to a single variable, which is fresh in the output state. (This same thing happens whenever matching two binders in any part of the MSG algorithm, including when matching $\forall$-bound type variables in the type MSG.)

You might expect that when the two arguments to msg_e cannot be matched then msg_e would attempt to generalise rather than failing (by calling **fail**), so that the MSG of e.g. $f\ x$ and $\lambda y.\, e$ would be a, where:

$$\begin{array}{lllllll} \theta_0 & = & \{a \mapsto a_0\} & H_0 & = & \{a_0 \mapsto f\ x\} \\ \theta_1 & = & \{a \mapsto a_1\} & H_1 & = & \{a_1 \mapsto \lambda y.\, e\} \end{array}$$

However, such a MSG does not in fact meet the criteria for MSG we set out because the instantiated, generalised terms are not cost-equivalent to the original terms. This can be seen from our example because:

$$\langle a_0 \mapsto f\ x \,|\, a_0 \,|\, \epsilon \rangle \not\lesssim \langle \epsilon \,|\, f\ x \,|\, \epsilon \rangle$$

(Similarly for the other input to MSG.) Intuitively, the reason is that the instantiated, generalised term introduces an additional layer of indirection (via a_0) and hence "slows down" the term. More formally, this fact can be seen as a consequence of the fact that Theorem C.3.1 cannot be used to prove a cost equivalence.

Variable MSG The rest of msg_e is straightforward: the interesting work of term MSG is done by variable MSG, implemented by msg_x and msg'_x.

The function msg_x is memoised by mutating θ_0 and θ_1. This is absolutely essential for a number of reasons:

- Firstly (and least importantly) it saves computation time by avoiding recomputing the result of MSGing together the same binder pairs multiple times

- Secondly, it avoids non-termination of msg_e when the terms under consideration make recursive reference to each other. For example, without memoisation, computing the MSG of $\langle \mathit{xs} \mapsto x : \mathit{xs} \,|\, \mathit{xs} \,|\, \epsilon \rangle$ and $\langle \mathit{ys} \mapsto y : \mathit{ys} \,|\, \mathit{ys} \,|\, \epsilon \rangle$ would not terminate.

- Thirdly, it allows us to later assume (in *fixup*, Section 5.5) that if a non-cheap heap binding is referred to twice in the range of a θ_j substitution, then work would be duplicated by allowing that binding to be part of the common output heap.

Having checked that msg_x cannot be satisfied from the memoised information, and that neither of the two argument variables are rigid, msg'_x is invoked. Since we know that neither argument is rigid, they must both be bound by the respective input heap/stack, or be free in the input. The call msg'_x has the job of MSGing together heap bindings to extend the common heap $\mathrm{r}\boldsymbol{H}$. Therefore, if both variables are heap-bound at this point we attempt to MSG together the corresponding heap bindings by making a recursive call to msg_e.

If the two input variables to msg'_x do not refer to heap bindings for which an MSG exists, we cannot hope to extend the common heap. However, in either case we do extend the environment of variables $\mathrm{r}\boldsymbol{\Gamma}$ in which the common state will typecheck with the newly allocated variable and its generalised type. Note that at this point $\mathrm{r}\boldsymbol{\Gamma}$ does not contain just *free* variables of the common state—it might contain some of the variables bound by $\mathrm{r}\boldsymbol{K}$ or $\mathrm{r}\boldsymbol{H}$. As we will see, the later call to *fixup* will prune out any entries in $\mathrm{r}\boldsymbol{\Gamma}$ which end up being bound in the common heap, along with creating suitable individual heaps containing the bindings that do not get commoned up into $\mathrm{r}\boldsymbol{H}$ by msg'_x.

$$\boxed{msg_K(K_0, K_1) = M(K)}$$

$$msg_K(K_0, K_1) = \begin{cases} \kappa, msg_K({K_0}', {K_1}') & K_0 \equiv \kappa_0, {K_0}' \wedge K_1 \equiv \kappa_1, {K_1}' \\ & \wedge \kappa \leftarrow msg_\kappa(\kappa_0, \kappa_1) \\ \epsilon & \text{otherwise} \end{cases}$$

$$\boxed{msg_\kappa(\kappa_0, \kappa_1) = M(\kappa)}$$

$$\begin{aligned}
msg_\kappa(\mathbf{update}\ x_0 {:} \tau_0, \mathbf{update}\ x_1 {:} \tau_1) &= \mathbf{update}\ msg_x(x_0, x_1)(\varnothing) {:} msg_\tau(\tau_0, \tau_1)(\varnothing) \\
msg_\kappa(\bullet\ x_0, \bullet\ x_1) &= \bullet\ msg_x(x_0, x_1)(\varnothing) \\
msg_\kappa(\bullet\ \tau_0, \bullet\ \tau_1) &= \bullet\ msg_\tau(\tau_0, \tau_1)(\varnothing) \\
msg_\kappa(\mathbf{case} \bullet \mathbf{of}\ \overline{\mathbf{C}\ \overline{\alpha {:} \kappa}\ \overline{x {:} \tau_0} \to e_{\mathbf{C}0}}^{n}, \mathbf{case} \bullet \mathbf{of}\ \overline{\mathbf{C}\ \overline{\alpha {:} \kappa}\ \overline{x {:} \tau_1} \to e_{\mathbf{C}1}}^{n}) & \\
= \mathbf{case} \bullet \mathbf{of}\ \overline{\mathbf{C}\ \overline{\alpha {:} \kappa}\ \overline{x {:} msg_\tau(\tau_0, \tau_1)(\overline{\alpha})} \to msg_e(e_{\mathbf{C}0}, e_{\mathbf{C}1})(\overline{\alpha}, \overline{x})}^{n} & \\
msg_\kappa(\kappa_0, \kappa_1) &= \mathbf{fail}
\end{aligned}$$

Figure 5.3: Call-by-need stack MSG

Stack MSG The MSG for stacks is defined in Figure 5.3. The job of msg_K is to attempt to MSG together as many consecutive pairs of stack frames from the two input states as possible. If the stack frames we wish to common up have different structure (i.e. their MSG fails), or either stack has no more frames available for commoning up, we simply return a truncated common stack. Our later call to *fixup* will deal with creating appropriate individual stacks.

The stack frame MSG, msg_κ, is uninteresting except insofar that we treat the binding variable occurrences in update frames exactly as if they were simple variable *occurrences* and MSG them with msg_x.

5.5 Removing MSG work-duplication in *fixup*

The last piece of the MSG algorithm is the *fixup* family of functions, defined in Figure 5.4, which is the part of MSG algorithm responsible for ensuring there is no work duplication and creating the individual heap/stack.

The core of *fixup* is a loop where we find the sets of variables Y_j which should be bound by the jth individual heap/stack. The reasons for which a variable x should be forced to be bound by the individual heap/stack are:

- Because x is referred to directly by the current common state $\langle \mathrm{r}\boldsymbol{H} \,|\, e \,|\, \mathrm{r}\boldsymbol{K} \rangle$ (this case applies if x is one of the variables you obtain after renaming the free variables of that state by $\mathrm{r}\boldsymbol{\theta_j}$).

- Because x is an expensive heap binding that msg'_x MSGed against more than one other corresponding expensive heap binding on the other side.

- Because x is referred to by one of the parts of the current individual heap/stack: i.e. x is required transitively.

The *fixup* algorithm continually iterates until all variables that *should* be bound by the individual heap stack *are* so bound. When iteration finishes, the final result of *msg* is constructed and returned.

To illustrate the *msg* algorithm overall and *fixup* in particular, we consider a number of examples.

$$\boxed{\mathit{fixup}(e)(X_0, X_1) = M(\langle H_0 \mid \theta_0 \mid K_0 \rangle_{\Sigma_0|\Gamma_0}, \langle H \mid e \mid K \rangle_{\Sigma|\Gamma}, \langle H_1 \mid \theta_1 \mid K_1 \rangle_{\Sigma_1|\Gamma_1})}$$

$$\begin{array}{lcl}
\mathit{fixup}(e)(X_0, X_1) & = & \mathbf{do} \\
\quad \mathbf{let} \;\; \Gamma_{\mathit{trim}} & = & \mathrm{r}\boldsymbol{\Gamma}|_{\mathrm{fvs}(\langle \mathrm{r}\boldsymbol{H} \mid e \mid \mathrm{r}\boldsymbol{K} \rangle)} \\
\qquad\quad H_{\mathit{prop}_j} & = & H_j|_{X_j} \\
\qquad\quad \theta_{\mathit{prop}_j} & = & \mathrm{r}\boldsymbol{\theta_j}|_{\mathrm{fvs}(\langle \mathrm{r}\boldsymbol{H} \mid e \mid \mathrm{r}\boldsymbol{K} \rangle)} \\
\qquad\quad K_{\mathit{prop}_j} & = & \mathit{drop}(|\mathrm{r}\boldsymbol{K}|, K_j) \\
 & & \\
\qquad\quad Y_j & = & (\mathrm{fvs}\left(H_{\mathit{prop}_j}\right) \cup \mathit{rng}(\theta_{\mathit{prop}_j}) \cup \mathrm{fvs}\left(K_{\mathit{prop}_j}\right) \\
 & \cup & \{x_j \mid \exists y, z. y \neq z \wedge y \mapsto x_j \in \mathrm{r}\boldsymbol{\theta_j} \wedge z \mapsto x_j \in \mathrm{r}\boldsymbol{\theta_j}\}) \\
 & \smallsetminus & X_j \\
\multicolumn{3}{l}{\quad \text{for } x_j \in Y_j \ \mathbf{do} \ \mathit{fixup}_x(x_j, j)} \\
\multicolumn{3}{l}{\quad \mathbf{if} \ Y_0 \not\equiv \varnothing \vee Y_1 \not\equiv \varnothing} \\
\multicolumn{3}{l}{\qquad \mathbf{then} \ \mathit{fixup}(e)(X_0 \cup Y_0, X_1 \cup Y_1)} \\
\multicolumn{3}{l}{\qquad \mathbf{else} \ (\left\langle H_{\mathit{prop}_0} \middle| \theta_{\mathit{prop}_0} \middle| K_{\mathit{prop}_0} \right\rangle_{\Sigma_0|\Gamma_0}, \langle \mathrm{r}\boldsymbol{H} \mid e \mid \mathrm{r}\boldsymbol{K} \rangle_{\mathrm{r}\boldsymbol{\Sigma}|\Gamma_{\mathit{trim}}}, \left\langle H_{\mathit{prop}_1} \middle| \theta_{\mathit{prop}_1} \middle| K_{\mathit{prop}_1} \right\rangle_{\Sigma_1|\Gamma_1})}
\end{array}$$

$$\mathit{fixup}_x(x, j) = \begin{cases} \mathrm{r}\boldsymbol{H} \coloneqq \mathrm{r}\boldsymbol{H} \smallsetminus \{y \mid y \mapsto x \in \mathrm{r}\boldsymbol{\theta_j}\} & x : \tau \mapsto e \in H_j \wedge \neg \mathit{cheap}(e) \\ \mathrm{r}\boldsymbol{K} \coloneqq \mathit{take}(i, \mathrm{r}\boldsymbol{K}) & K_j[i] = \mathbf{update} \ x : \tau \\ () & \text{otherwise} \end{cases}$$

Figure 5.4: MSG *fixup* functions

Example 1: preventing work duplication by generalising the heap First, consider the following example of two inputs to MSG that differ in their work-sharing properties:

$$\mathcal{S}_0 = \langle a \mapsto f \ y \mid (a, a) \mid \epsilon \rangle \quad \mathcal{S}_1 = \langle b \mapsto f \ y, c \mapsto f \ y \mid (b, c) \mid \epsilon \rangle$$

After msg_e and msg_K complete, we have:

$$\begin{array}{rcl}
\mathrm{r}\boldsymbol{H} & = & d \mapsto f \ y, e \mapsto f \ y \\
e & = & (d, e) \\
\mathrm{r}\boldsymbol{K} & = & \epsilon \\
\mathrm{r}\boldsymbol{\theta_0} & = & \{f \mapsto f, y \mapsto y, d \mapsto a, e \mapsto a\} \\
\mathrm{r}\boldsymbol{\theta_1} & = & \{f \mapsto f, y \mapsto y, d \mapsto b, e \mapsto c\}
\end{array}$$

Because the variable a occurs twice in the range of $\mathrm{r}\boldsymbol{\theta_0}$, we know that work will have been duplicated if a refers to an expensive heap binding in $\mathcal{S}_0$, which in this example it does. The first iteration of *fixup* detects this work duplication and extends Y_0 accordingly:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
X_0 & = & \varnothing & X_1 & = & \varnothing \\
H_{\mathit{prop}_0} & = & \epsilon & H_{\mathit{prop}_1} & = & \epsilon \\
K_{\mathit{prop}_0} & = & \epsilon & K_{\mathit{prop}_1} & = & \epsilon \\
\theta_{\mathit{prop}_0} & = & \{f \mapsto f, y \mapsto y\} & \theta_{\mathit{prop}_1} & = & \{f \mapsto f, y \mapsto y\} \\
Y_0 & = & \{f, y, a\} & Y_1 & = & \{f, y\}
\end{array}$$

The resulting call $\mathit{fixup}_x(a, 0)$ rectifies the work duplication issue by removing the d and e bindings from the common heap, so on the next iteration we have:

$$\mathrm{r}\boldsymbol{H} = \epsilon$$

$$\begin{array}{rcl@{\qquad\qquad}rcl}
X_0 & = & \{f, y, a\} & X_1 & = & \{f, y\} \\
H_{\mathit{prop}_0} & = & a \mapsto f \ y & H_{\mathit{prop}_1} & = & \epsilon \\
\theta_{\mathit{prop}_0} & = & \{f \mapsto f, y \mapsto y, d \mapsto a, e \mapsto a\} & \theta_{\mathit{prop}_1} & = & \{f \mapsto f, y \mapsto y, d \mapsto b, e \mapsto c\} \\
Y_0 & = & \varnothing & Y_1 & = & \{b, c\}
\end{array}$$

On this iteration, we have detected that the removal of d and e that was triggered on the previous iteration means that we now have to ensure that b and c are bound on the right hand side. The calls $\mathit{fixup}_x(b,1)$ and $\mathit{fixup}_x(c,1)$ do nothing since we already trimmed the common heap, and so the next iteration of *fixup* is also the final one, terminating with:

$$X_0 = \{f,y,a\} \qquad X_1 = \{f,y,b,c\}$$

Example 2: work duplication discovered transitively Consider these two inputs to MSG:

$$\mathcal{S}_0 = \langle a \mapsto f\ b, b \mapsto f\ a \,|\, (a,b) \,|\, \epsilon\rangle \quad \mathcal{S}_1 = \langle c \mapsto f\ d \,|\, (c,e) \,|\, \epsilon\rangle$$

After msg_e and msg_K complete, we have:

$$\begin{aligned}
\mathrm{r}\boldsymbol{H} &= g \mapsto f\ i \\
e &= (g,h) \\
\mathrm{r}\boldsymbol{K} &= \epsilon \\
\mathrm{r}\boldsymbol{\theta_0} &= \{f \mapsto f, g \mapsto a, i \mapsto b, h \mapsto b\} \\
\mathrm{r}\boldsymbol{\theta_1} &= \{f \mapsto f, g \mapsto c, i \mapsto d, h \mapsto e\}
\end{aligned}$$

Unlike our previous example, there is no immediate work duplication problem. This does not mean that *fixup* has no work to do: its first iteration will still need to ensure that b is bound in the left-hand individual heap:

$$\begin{aligned}
X_0 &= \varnothing & X_1 &= \varnothing \\
H_{\mathit{prop}\,0} &= \epsilon & H_{\mathit{prop}\,1} &= \epsilon \\
K_{\mathit{prop}\,0} &= \epsilon & K_{\mathit{prop}\,1} &= \epsilon \\
\theta_{\mathit{prop}\,0} &= \{h \mapsto b\} & \theta_{\mathit{prop}\,1} &= \{h \mapsto e\} \\
Y_0 &= \{b\} & Y_1 &= \{e\}
\end{aligned}$$

Because the b binding transitively refers to a, the following *fixup* iteration will attempt to bind a in the left-hand individual heap:

$$\begin{aligned}
X_0 &= \{b\} & X_1 &= \{e\} \\
H_{\mathit{prop}\,0} &= b \mapsto f\ a & H_{\mathit{prop}\,1} &= \epsilon \\
\theta_{\mathit{prop}\,0} &= \{h \mapsto b\} & \theta_{\mathit{prop}\,1} &= \{h \mapsto e\} \\
Y_0 &= \{f,a\} & Y_1 &= \varnothing
\end{aligned}$$

However, since the a binding is expensive, we can't bind a in the left-hand individual heap without also preventing its use in the common heap, so the resulting $\mathit{fixup}_x(a,0)$ call removes g from the common heap to prevent work duplication. The next iteration of *fixup* therefore detects that the corresponding binding on the other side, c, must be put into the individual heap:

$$\mathrm{r}\boldsymbol{H} = \varnothing$$

$$\begin{aligned}
X_0 &= \{b,f,a\} & X_1 &= \{e\} \\
H_{\mathit{prop}\,0} &= a \mapsto f\ b, b \mapsto f\ a & H_{\mathit{prop}\,1} &= \epsilon \\
\theta_{\mathit{prop}\,0} &= \{g \mapsto a, h \mapsto b\} & \theta_{\mathit{prop}\,1} &= \{g \mapsto c, h \mapsto e\} \\
Y_0 &= \varnothing & Y_1 &= \{c\}
\end{aligned}$$

After one more iteration to ensure the free variables of the a binding are bound, the *fixup* iteration terminates:

$$X_0 = \{b,f,a\} \qquad X_1 = \{e,c,f,d\}$$

Example 3: generalising the stack due to update frames The examples above have dealt with heap bindings. In this final example, we see the analogous processes for the stack. Consider these two MSG inputs:

$$\mathcal{S}_0 = \langle a \mapsto b + 1 \,|\, g \,|\, \textbf{update}\ b, \bullet\ a \rangle \quad \mathcal{S}_1 = \langle \epsilon \,|\, g \,|\, \textbf{update}\ d, \bullet\ c \rangle$$

After msg_e and msg_K complete, we have:

$$\begin{array}{rcl} \mathrm{r}\boldsymbol{H} & = & \epsilon \\ e & = & g \\ \mathrm{r}\boldsymbol{K} & = & \textbf{update}\ e, \bullet\ f \\ \mathrm{r}\boldsymbol{\theta_0} & = & \{g \mapsto g, e \mapsto b, f \mapsto a\} \\ \mathrm{r}\boldsymbol{\theta_1} & = & \{g \mapsto g, e \mapsto d, f \mapsto c\} \end{array}$$

In this example, like the last, there is no immediate work duplication problem, but *fixup* will find that it needs to bind a in the left-hand individual heap:

$$\begin{array}{rclrcl} X_0 & = & \varnothing & X_1 & = & \varnothing \\ H_{prop\,0} & = & \epsilon & H_{prop\,1} & = & \epsilon \\ K_{prop\,0} & = & \epsilon & K_{prop\,1} & = & \epsilon \\ \theta_{prop\,0} & = & \{g \mapsto g, f \mapsto a\} & \theta_{prop\,1} & = & \{g \mapsto g, f \mapsto c\} \\ Y_0 & = & \{g, a\} & Y_1 & = & \{g, c\} \end{array}$$

The next iteration will attempt to find the variable b that is transitively used by the a heap binding:

$$\begin{array}{rclrcl} X_0 & = & \{g, a\} & X_1 & = & \{g, c\} \\ H_{prop\,0} & = & a \mapsto b + 1 & H_{prop\,1} & = & \epsilon \\ \theta_{prop\,0} & = & \{g \mapsto g, f \mapsto a\} & \theta_{prop\,1} & = & \{g \mapsto g, f \mapsto c\} \\ Y_0 & = & \{b\} & Y_1 & = & \varnothing \end{array}$$

Unlike in the last example, the transitively used variable b is bound by an update frame and so the call $\mathit{fixup}_x(b, 0)$ ends up causing a portion of the common stack to be dropped. The next *fixup* iteration detects that it is done and returns:

$$\begin{array}{rclrcl} & & & \mathrm{r}\boldsymbol{K} & = & \epsilon \\ X_0 & = & \{g, a, b\} & X_1 & = & \{g, c\} \\ K_{prop\,0} & = & \textbf{update}\ b, \bullet\ a & K_{prop\,1} & = & \textbf{update}\ d, \bullet\ c \\ \theta_{prop\,0} & = & \{g \mapsto g\} & \theta_{prop\,1} & = & \{g \mapsto g\} \end{array}$$

5.6 Is the MSG necessarily well-typed?

Before we move on to define the final part of the MSG algorithm (the function msg_τ implementing type MSG), we stop to consider how our choice of a typed language impacts the algorithm. Supercompilation work typically makes use of the MSG in an untyped setting, but when working with a typed language we need to deal with the additional constraint that the result of MSG must be well-typed. This introduces further complications, as we will show in the following examples.

Type variable reuse When we take the MSG of the two untyped terms $f\ \mathit{True}\ \mathit{True}$ and $f\ 1\ 1$ we would be justified in using a new variable to generalise each point of difference between the two inputs, so a sensible generalised common term would be $f\ x\ y$.

Now consider the two typed terms $\lambda(x :: \mathit{Int})\ (f :: \mathit{Int} \to \mathit{Char}).f\ x$ and $\lambda(x :: \mathit{Bool})\ (f :: \mathit{Bool} \to \mathit{Char}).f\ x$. A naive MSG of these two terms might take the same approach at the type level, and assign a new type variable to every point of difference between the two, like so:

$$\lambda(x :: \alpha)\ (f :: \beta \to \mathit{Char}).f\ x$$

Unfortunately, we can see immediately that this term does not type check in isolation. If we instead ensure we reuse the same type variable in all generalised positions if appropriate, the term will type check:

$$\lambda(x :: \alpha)\ (f :: \alpha \to \mathit{Char}).f\ x$$

Higher-rank types Consider also the MSG of the two typed terms $\lambda(f :: \forall a.a).f\ \mathit{Int}$ and $\lambda(f :: \forall a.\mathit{Int}).f\ \mathit{Bool}$. Due to variable capture, there is clearly no way for the MSGed λ binder to have a $\forall$type because the only two sensible such terms are:

$$\lambda(f :: \forall a.a).f\ \beta \quad \lambda(f :: \forall a.\mathit{Int}).f\ \beta$$

(Note that $\theta_0 = \{\beta \mapsto \mathit{Int}\}, \theta_1 = \{\beta \mapsto \mathit{Bool}\}$ in both cases.) Neither proposed term is a generalisation of *both* of the inputs to MSG. We might instead say that the MSG is:

$$\lambda(f :: \alpha).f\ \beta(\theta_0 = \{\alpha \mapsto \forall a.a, \beta \mapsto \mathit{Int}\}, \theta_1 = \{\alpha \mapsto \forall a.\mathit{Int}, \beta \mapsto \mathit{Bool}\})$$

Unfortunately, this generalised term is not well typed due to the type application to β, so we can see that there is no non-trivial generalisation of these two terms.

5.7 Conditions for MSG type correctness

The previous section demonstrates that a generalised term is not necessarily type correct, even if the terms it is a generalisation of are well-typed. In this section we will establish simple syntactic conditions on the result of term generalisation that are sufficient to establish that the generalised term is well typed.

For the purposes of proving correctness, we restrict ourselves to the setting of System Fω[3], and do not consider features of the full Core language such as **let**. Nonetheless, we conjecture that our our correctness proof could be extended to Core without exposing any problems or requiring the use of different proof techniques.

In order to state what we wish to prove, we need to define the concept of *non-coupling* (type) substitution pairs:

Definition 5.7.1 (Non-coupling substitutions)**.** The type substitutions θ_0 and θ_1 are non-coupling iff:

1. They have common domains: $\exists \Sigma.\mathrm{dom}\,(\theta_0) = \mathrm{dom}\,(\theta_1) = \Sigma$
2. All pairs of types are unique: $\neg\exists \alpha{:}\kappa \in \Sigma, \beta{:}\kappa \in \Sigma.\alpha \neq \beta \wedge \alpha\theta_0 = \beta\theta_0 \wedge \alpha\theta_1 = \beta\theta_1$
3. Each pair of types is non-coupling: $\forall \alpha{:}\kappa \in \Sigma.\neg \mathit{coupled}\,(\alpha\theta_0, \alpha\theta_1)$

The coupling relation picks out those pairs of terms that are "the same at the root" and is defined by the following axioms:

$$\frac{}{\mathit{coupled}\,(\tau_0\ \upsilon_0, \tau_1\ \upsilon_1)} \quad \frac{}{\mathit{coupled}\,((\to),(\to))} \quad \frac{}{\mathit{coupled}\,(\forall\alpha{:}\kappa.\,\tau_0, \forall\alpha{:}\kappa.\,\tau_1)}$$

[3] The syntax and static semantics of System Fω and its type substitutions are given in full in Appendix B, along with the detailed proof of the correctness of type generalisation.

Note that the conditions required by coupling reflect the lessons learnt from the examples in the previous section. (Because of our restriction to System Fω, the only coupled pair of type constructors is $(\to)$: in a language with more type constructors such as Core, all identical type constructors must also be considered coupled.)

With this setup, we can prove the following theorem, which we will use to justify our type generalisation algorithm in Section 5.8:

Theorem 5.7.1 (Correctness of type generalisation). *In System Fω, if we have Σ_0, Σ_1, Σ, Γ_0, Γ_1, Γ, e_0, e_1, e, τ_0, τ_1, θ_0, θ_1 such that:*

- *The ungeneralised terms are well typed: $\Sigma_0|\Gamma_0 \vdash e_0 \ : \ \tau_0$ and $\Sigma_1|\Gamma_1 \vdash e_1 \ : \ \tau_1$*
- *θ_0 and θ_1 are non-coupling with common domain Σ*
- *The term is a generalisation of the originals: $e\theta_0 = e_0$ and $e\theta_1 = e_1$*
- *The type environment is a generalisation of the originals: $\Gamma\theta_0 = \Gamma_0$ and $\Gamma\theta_1 = \Gamma_1$.*

Then there exists a (unique) type τ such that:

- *The generalised term is well typed: $\Sigma|\Gamma \vdash e \ : \ \tau$*
- *The type is a generalisation of the originals: $\tau\theta_0 = \tau_0$ and $\tau\theta_1 = \tau_1$*

Proof. See Appendix B □

5.8 Type MSG algorithm

We are now in a position to define the type MSG msg_τ that is used by the term MSG to generalise type arguments and the types of variable binders. Our type MSG (presented in Figure 5.5) makes use of mutable variables, just like the term and stack MSG. However, due to its simpler nature it only needs to inspect the accumulated substitutions $\mathrm{r}\boldsymbol{\theta_j}$.

The algorithm is defined bearing in mind the syntactic conditions on type-correctness of the MSGed term developed in the previous section, a fact which we prove with the following theorem:

Theorem 5.8.1. *If $\theta_j \leftarrow \mathrm{r}\boldsymbol{\theta_j}$ are substitutions that are non-capturing for A and non-coupling, then after executing $msg_\tau(\tau_0, \tau_1)(A)$, we have that:*

I The substitutions are extensions of the old ones: $\forall j.\theta_j \subseteq \mathrm{r}\boldsymbol{\theta_j}$.

II The substitutions $\mathrm{r}\boldsymbol{\theta_j}$ are still non-capturing for A and non-coupling.

III If the call succeeded with result τ then:

(a) The result is a generalisation: $\forall j.\tau_j = \tau\mathrm{r}\boldsymbol{\theta_j}$.

(b) This is the most specific generalisation satisfying the conditions I and II.

Proof. The proof of I is straightforward by observing that the sole line which modifies $\mathrm{r}\boldsymbol{\theta_j}$ simply extends it.

The non-capturing portion of II follows easily by observing the explicit check for A-capture that is made before calling gen_τ. To show non-coupling, observe that:

- All pairs of types are unique because previously-allocated type variables may be reused when msg_τ consults the $\mathrm{r}\boldsymbol{\theta_j}$ before allocating a fresh type variable.

$$\boxed{\mathit{msg}_\tau(\tau_0,\tau_1) = M(\tau)}$$

$$\begin{array}{rcl}
\mathit{msg}_\tau(\alpha,\alpha)(A \cup \{\alpha\}) & = & \alpha \\
\mathit{msg}_\tau(\mathbf{T},\mathbf{T})(A) & = & \mathbf{T} \\
\mathit{msg}_\tau(\tau_0\ \upsilon_0,\tau_1\ \upsilon_1)(A) & = & \mathit{msg}_\tau(\tau_0,\tau_1)(A)\ \mathit{msg}_\tau(\upsilon_0,\upsilon_1)(A) \\
\mathit{msg}_\tau(\forall\alpha{:}\kappa.\,\tau_0, \forall\alpha{:}\kappa.\,\tau_0)(A) & = & \forall\alpha{:}\kappa.\,\mathit{msg}_\tau(\tau_0,\tau_1)(A \cup \{\alpha\}) \\
\mathit{msg}_\tau(\tau_0,\tau_1)(A) & = & \begin{cases} \alpha & \alpha \mapsto \tau_0 \in \mathrm{r}\boldsymbol{\theta_0} \wedge \alpha \mapsto \tau_1 \in \mathrm{r}\boldsymbol{\theta_1} \\ \mathbf{fail} & A \cap (\mathrm{fvs}\,(\tau_0) \cup \mathrm{fvs}\,(\tau_1)) \neq \varnothing \\ \mathit{gen}_\tau(\tau_0,\tau_1) & \text{otherwise} \end{cases}
\end{array}$$

$$\boxed{\mathit{gen}_\tau(\tau_0,\tau_1) = M(\tau)}$$

$$\begin{array}{rcl}
\mathit{gen}_\tau(\tau_0,\tau_1) & = & \mathbf{do}\ \alpha \leftarrow \mathit{fresh} \\
 & & \quad \text{if } \exists\kappa.\mathit{kindof}(\tau_0) \equiv \mathit{kindof}(\tau_1) \equiv \kappa \\
 & & \quad\quad \text{then } \mathrm{r}\boldsymbol{\Sigma} := \mathrm{r}\boldsymbol{\Sigma}, \alpha{:}\kappa \\
 & & \quad\quad \text{else } \mathbf{fail} \\
 & & \quad \mathrm{r}\boldsymbol{\theta_j} := \mathrm{r}\boldsymbol{\theta_j} \cup \{\alpha \mapsto \tau_j\} \\
 & & \quad \alpha
\end{array}$$

$$\boxed{\mathit{kindof}(\tau_j) = \kappa_j}$$

$$\frac{\Sigma_j \vdash^\kappa \tau_j \ :\ \kappa_j}{\mathit{kindof}(\tau_j)\ =\ \kappa_j}$$

Figure 5.5: Type MSG

- All pairs of types that *are* added to $\mathrm{r}\boldsymbol{\theta_j}$ may not be *coupled* because coupled type pairs are caught by earlier pattern matches in the msg_τ function.

The proof of generalisation required for IIIa follows straightforwardly from the fact that once an entry is added to the substitution it may not be removed (I) and that such an entry is added whenever the MSGed type is in fact generalised.

Finally, to see that this is in fact a *most specific* generalisation, simply note that when we extend the substitutions by calling gen_τ we have already determined that there is no common outermost structure between the two types.

□

This allows us to prove our overall goal:

Theorem 5.8.2 (MSG well-typed)**.**

$$\begin{array}{l}
\mathit{msg}(\mathcal{S}_0,\mathcal{S}_1) = (\langle H_0{}' \,|\, \theta_0 \,|\, K_0{}' \rangle, \mathcal{S}, \langle H_1{}' \,|\, \theta_1 \,|\, K_1{}' \rangle) \wedge \vdash \mathcal{S}_0 \ :\ \tau_0 \wedge \vdash \mathcal{S}_1 \ :\ \tau_1 \\
\quad \Longrightarrow \quad \exists\tau.\tau\theta_0 = \tau_0 \ \wedge\ \tau\theta_1 = \tau_1 \wedge \vdash \mathcal{S} \ :\ \tau
\end{array}$$

Proof. Theorem 5.8.1 shows that msg_τ preserves the non-coupling and non-capturing nature of the accumulated type substitutions. Since the initial (empty) type substitutions that are set by *msg* are also non-coupling and non-capturing, the final type substitution must also be non-coupling. As a result, Theorem 5.7.1 may be used to prove that the common state returned by *msg* is well-typed. □

5.9 MSGing states rather than terms

Working with states rather than terms is useful for our MSG algorithm. There are two principal reasons for this:

- Identifying the stack explicitly is crucial in our argument that our MSG obeys cost equivalence. When given two states:

  $$\begin{array}{lcl} \mathcal{S}_0 & = & \langle f \mapsto \ldots \mid \bullet\ x, \mathbf{case}\bullet\mathbf{of}\ \mathit{True} \to g; \mathit{False} \to h \mid f \rangle \\ \mathcal{S}_1 & = & \left\langle f \mapsto \ldots \;\middle|\; \bullet\ x, \mathbf{case}\bullet\mathbf{of}\ \begin{array}{ll}\mathit{True} & \to g \\ \mathit{False} & \to h\end{array}, \bullet\ x, \mathbf{case}\bullet\mathbf{of}\ \begin{array}{ll}\mathit{True} & \to g \\ \mathit{False} & \to h\end{array} \;\middle|\; f \right\rangle \end{array}$$

  The algorithm determines the most specific generalisation of the two states to be a new state:

  $$\langle f \mapsto \ldots \mid \bullet\ x, \mathbf{case}\bullet\mathbf{of}\ \mathit{True} \to g; \mathit{False} \to h \mid f \rangle$$

  Which is cost-equivalent to the inputs in the sense that:

  $$\begin{array}{lcl} \mathcal{S}_0 & \overset{\bowtie}{\sim} & \langle\!\langle \epsilon \mid \langle\!\langle f \mapsto \ldots \mid \bullet\ x, \mathbf{case}\bullet\mathbf{of}\ \mathit{True} \to g; \mathit{False} \to h \mid f \rangle\!\rangle \mid \epsilon \rangle\!\rangle \\ \mathcal{S}_1 & \overset{\bowtie}{\sim} & \left\langle\!\!\left\langle \epsilon \;\middle|\; \left\langle\!\!\left\langle f \mapsto \ldots \;\middle|\; \bullet\ x, \mathbf{case}\bullet\mathbf{of}\ \begin{array}{ll}\mathit{True} & \to g \\ \mathit{False} & \to h\end{array} \;\middle|\; f \right\rangle\!\!\right\rangle \;\middle|\; \bullet\ x, \mathbf{case}\bullet\mathbf{of}\ \begin{array}{ll}\mathit{True} & \to g \\ \mathit{False} & \to h\end{array} \right\rangle\!\!\right\rangle \end{array}$$

  A more usual approach to obtaining this generalisation in a supercompiler which is based on terms instead of states, is the "bowtie" generalisation of Mitchell [2008], which takes the two corresponding input *terms*:

  $$\begin{array}{lcl} e_0 & = & \mathbf{let}\ f = \ldots\ \mathbf{in}\ \mathbf{case}\ f\ x\ \mathbf{of}\ \mathit{True} \to g; \mathit{False} \to h \\ e_1 & = & \mathbf{let}\ f = \ldots\ \mathbf{in}\ \mathbf{case}\ (\mathbf{case}\ f\ x\ \mathbf{of}\ \mathit{True} \to g; \mathit{False} \to h)\ x\ \mathbf{of}\ \mathit{True} \to g; \mathit{False} \to h \end{array}$$

  And generalises them to a new term:

  $$\mathbf{let}\ f = \ldots\ \mathbf{in}\ \mathbf{case}\ f\ x\ \mathbf{of}\ \mathit{True} \to g; \mathit{False} \to h$$

  Such that:

  $$\begin{array}{lcl} e_0 & = & \mathbf{let}\ \mathit{gen} = \mathbf{let}\ f = \ldots\ \mathbf{in}\ \mathbf{case}\ f\ x\ \mathbf{of}\ \mathit{True} \to g; \mathit{False} \to h\ \mathbf{in}\ \mathit{gen} \\ e_1 & = & \begin{array}{l}\mathbf{let}\ \mathit{gen} = \mathbf{let}\ f = \ldots\ \mathbf{in}\ \mathbf{case}\ f\ x\ \mathbf{of}\ \mathit{True} \to g; \mathit{False} \to h \\ \qquad\qquad \mathbf{in}\ \mathbf{case}\ \mathit{gen}\ x\ \mathbf{of}\ \mathit{True} \to g; \mathit{False} \to h\end{array} \end{array}$$

  But notice that:

  $$e_0 \not\overset{\bowtie}{\sim} \mathbf{let}\ \mathit{gen} = \mathbf{let}\ f = \ldots\ \mathbf{in}\ \mathbf{case}\ f\ x\ \mathbf{of}\ \mathit{True} \to g; \mathit{False} \to h\ \mathbf{in}\ \mathit{gen}$$

  (and similarly for e_1) since pulling out a term into a **let**-binding may introduce extra non-normalising reductions that could have been performed with a normalising reduction rule had the term been left in place. Working directly with states, and hence allowing instantiation with both a surrounding heap *and a stack* avoids introducing extra *gen*-like bindings and hence eases the proof of supercompiler correctness in Section 3.8.

- Explicitly identifying the stack makes it natural to match terms "inside out" (starting from the focus of evaluation) rather than "outside in" (starting from the final stack frame) as with most implementations. An example that demonstrates that this strategy can lead to better generalisation is the MSG of these two states:

  $$\begin{array}{lcl} \mathcal{S}_0 & = & \left\langle \begin{array}{l} \mathit{odd} \mapsto \lambda n.\, \mathbf{case}\ n\ \mathbf{of}\ Z \to \mathit{False} \\ \qquad\qquad\qquad\qquad S\ m \to \mathbf{case}\ (\mathit{odd}\ m)\ \mathbf{of}\ \mathit{True} \to \mathit{False} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{False} \to \mathit{True} \end{array} \;\middle|\; \mathit{odd} \;\middle|\; \bullet\ n \right\rangle \\ \mathcal{S}_1 & = & \langle \mathit{odd} \mapsto \ldots \mid \mathit{odd} \mid \bullet\ n, \mathbf{case}\bullet\mathbf{of}\ \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True} \rangle \end{array}$$

With corresponding terms:

$$\begin{array}{lcl} \mathit{rebuild}\ \mathcal{S}_0 & = & \textbf{let}\ \mathit{odd} = \ldots \textbf{in}\ \mathit{odd}\ n \\ \mathit{rebuild}\ \mathcal{S}_1 & = & \textbf{let}\ \mathit{odd} = \ldots \textbf{in}\ \textbf{case}\ \mathit{odd}\ n\ \textbf{of}\ \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True} \end{array}$$

A standard call-by-name MSG on terms (such as that of HOSC [Klyuchnikov, 2010a]: "bowtie" generalisation does not generally exhibit this problem) would claim that these two terms have the trivial MSG x (with appropriate instantiating variable-to-term substitutions) and thus a supercompiler using such a MSG to implement generalisation (Section 6.2) will be forced to *split* rather than generalise, producing suboptimal code that contains a residual β-reduction of *odd*.

In contrast, our *msg* matches the stack from the innermost frame outwards, and so it will return the state $\langle \mathit{odd} \mapsto \ldots \mid \mathit{odd} \mid \bullet\ n \rangle$ as the MSG of $\mathcal{S}_0$ and $\mathcal{S}_1$, giving the supercompiler the opportunity to avoid a residual β-reduction of *odd*.

Chapter 6

Improving the basic supercompiler

In this chapter, we describe three additions to the core supercompilation algorithm designed to improve the quality of the output code:

- **Rollback**: we show (Section 6.1) how to reduce supercompilation code duplication by letting the supercompiler *roll back* to a previous state at certain points.
- **Generalisation**: we show (Section 6.2) how we can achieve better specialisation by selectively throwing away information that is causing the termination test to fail. In particular:
  - We describe a novel approach to term generalisation in a supercompiler that uses tags, which we call *growing tag generalisation*.
  - We show how to implement growing tag generalisation with a minimum of code by reusing the logic of *split* (Section 6.2.2).
  - We compare growing tag generalisation with a generalisation technique based on our call-by-need MSG (Section 6.2.3).
  - We discuss type generalisation to combat the issue of type overspecialisation (Section 6.2.6).
- **Speculation**: the need for a call-by-need supercompiler to preserve sharing may lead to a loss of information and hence a reduction in the amount of optimisation the supercompiler may achieve. We describe how a "speculating" reduction semantics can be used to detect those heap bindings (such as partial applications), for which no work is lost when they are duplicated (Section 6.3).

Each technique is easy to implement by modifying either a single module of our modular algorithm, or by changing the way we use our existing modules in the top-level *sc* implementation. We evaluate each technique individually in Chapter 7.

6.1 Rollback

Suppose that we apply *reduce* to the following term:

$$\textbf{let}\, f\ x = \textbf{case}\, f\ x\ \textbf{of}\ \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True}\ \textbf{in}\, f\ \mathit{True}$$

Evaluation yields a sequence of terms of the form

$f\ \mathit{True}$
$\textbf{case}\, f\ \mathit{True}\ \textbf{of}\ \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True}$
$\textbf{case}\ (\textbf{case}\, f\ \mathit{True}\ \textbf{of}\ \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True})\ \textbf{of}\ \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True}$
$\ldots$

and so does not terminate. Therefore, the *terminate* call in *reduce* (as defined in Section 3.3) will eventually say *Stop*, and *reduce* will return whatever term it has reached, say **case** (**case** *f x* **of** *True* → *False*; *False* → *True*) **of** *True* → *False*; *False* → *True*. But since we have detected probable divergence it might reduce output code size to discard the fruitless work and *roll back* to an earlier term that has not grown so much. For this example, this would let us improve the output of the supercompiler from the term:

```
let h0 = let f x = h1 x
         in case f True of True → True; False → False
    h1 x = let f x = h1 x
           in case f x of True → False; False → True
in h0
```

To the smaller term:

```
let h0 = let f x = h1 x in f True
    h1 x = let f x = h1 x
           in case f x of True → False; False → True
in h0
```

Because the termination criteria is used in two ways in the supercompiler, there are two opportunities to introduce rollback—we not only add it to *reduce*, but to *sc* as well.

6.1.1 Rolling back reduction

The most straightforward change is to the *reduce* function (Section 3.3). It would be possible for *reduce* to revert all the way to its original input term in the event of divergence, but that risks discarding any *useful* computation performed, along with the bloat. For example, suppose the body of the **let** in the example above was *id* (*f* 2) where *id* is the identity function. Then it would be a pity to discard the work of reducing the call to *id*.

A more sophisticated approach is to modify the termination test API, so that when *terminate* reports *Stop*, it also yields information recorded along with the earlier *State* that is related to the new one by ⊴. A simple implementation of this could be as follows:[1]

```
type History a = [(State, a)]
emptyHistory :: History a

data TermRes a = Stop a | Continue (History a)

terminate :: History a → State → a → TermRes a
terminate hist here here_extra
  | prev_extra: _ ← [ prev_extra
                    | (prev, prev_extra) ← hist
                    , prev ⊴ here ]
  = Stop prev_extra
  | otherwise
  = Continue ((here, here_extra) : hist)
```

To allow *reduce* to rollback, we can now use the extra data field in the *History* to store the current *State* whenever the termination criteria is tested[2]. Should we be forced to

[1] Our implementation is actually based on a version of the termination combinators introduced in Appendix D, extended with the ability to return a value with a *Stop*.

[2] It may seem redundant to store the *State* twice in each *History* entry, but this design is chosen for uniformity with Section 6.1.2

Stop reduction, that stored *State* is returned instead of the latest (more-*reduce*d) version. A suitable implementation of *reduce* is as follows (the code may be compared with the original version in Section 3.3):

```
reduce :: State → State
reduce = go emptyHistory
  where
    go hist 𝒮 = case terminate hist 𝒮 𝒮 of
      Stop 𝒮_old       → 𝒮_old
      Continue hist′ → case step 𝒮 of
        Nothing → 𝒮
        Just u𝒮′ → go hist′ (normalise u𝒮′)
```

A simple example of where this makes a difference is using *reduce* on a term such as this:

```
let loop xs = loop (x : xs)
    id y = y
in id (loop [ ])
```

A supercompiler without *reduce*-rollback would produce a *State* such as this one:

```
let loop xs = loop (x : xs)
    xs0 = [ ]
    xs1 = x : xs0
    xs2 = x : xs1
    xs3 = x : xs2
in loop xs3
```

Our supercompiler instead rolls back to an earlier *State* where we duplicate less code:[3]

```
let loop xs = loop (x : xs)
    xs0 = [ ]
    xs1 = x : xs0
    xs2 = x : xs1
in loop xs2
```

The *reduce* function with rollback is, pleasingly, idempotent.

6.1.2 Rolling back driving

The other use of *terminate* occurs in the *sc* function itself, where it controls how deeply nested recursive invocations of *sc* can become.

In the introduction to Section 6.1.1 we already saw how rollback could improve *reduce*, but can the same idea really improve *sc* as well? We claim it can, in particular by reducing the extent to which loops are fruitlessly peeled and unrolled. Consider the state $\mathcal{S}_0$:

```
let doubleacc n v = case n of Z   → v
                              S m → let w = (S (S v))^t0
                                    in (doubleacc m w)^t1
```

[3] The supercompiler still unrolls the call to *loop* twice, whereas one unrolling might seem more reasonable. This happens because the call to *loop* in the body of the **let** is assigned a different tag to the occurrence of *loop* in the body of *loop* itself.

```
     v = (S Z)^{t_0}
in (doubleacc n v)^{t_1}
```

After reduction and splitting, we will drive the following subcomponent $\mathcal{S}_1$:

```
let doubleacc n v = ...
    v = (S Z)^{t_0}
    w0 = (S (S v))^{t_0}
in (doubleacc m0 w0)^{t_1}
```

Now, *tagBag* $\mathcal{S}_0 \trianglelefteq$ *tagBag* $\mathcal{S}_1$, so we will *split* without reducing, resulting in driving the following subcomponent state $\mathcal{S}_2$:

```
let doubleacc n v = ...
in case n of Z      → v
             S m0 → let w0 = (S (S v))^{t_0}
                    in (doubleacc m0 w0)^{t_1}
```

We then supercompile a subcomponent derived from the S branch, $\mathcal{S}_3$:

```
let doubleacc n v = ...
    w0 = (S (S v))^{t_0}
in (doubleacc m0 w0)^{t_1}
```

However, *tagBag* $\mathcal{S}_0 \trianglelefteq$ *tagBag* $\mathcal{S}_3$, so we *split* again, driving the body of *doubleacc* as a subcomponent. This results in tying back to $\mathcal{S}_2$, so the final supercompiled program is:

```
let h0 n = case n of Z      → S Z
                     S m0 → h1 m0
    h1 m0 = let doubleacc n v = h2 n v
                v = S Z
                w0 = S (S v)
            in doubleacc m0 w0
    h2 n v = case n of Z      → v
                       S m0 → h3 m0 v
    h3 m0 v = let doubleacc n v = h2 n v
                  w0 = S (S v)
              in doubleacc m0 w0
in h0 n
```

Although this supercompiled result is not bad, it has peeled *doubleacc* once for no gain (observe that the *h2* and *h3* functions form the main recursive loop, and *h0* and *h1* just perform the first iteration).

If we change the supercompiler so that it rolls back to the state into which it is embedded when the termination test fails, then we can avoid this peeling. What this would mean for our example is that when we detect the termination test failure caused by driving $\mathcal{S}_1$, we would roll back to $\mathcal{S}_0$ and act as if the termination test had failed at *that* point. If we do this, we will recursively supercompile a new subcomponent $\mathcal{S}'_1$:

```
let doubleacc n v = ...
in case n of Z      → v
             S m0 → let w0 = (S (S v))^{t_0}
                    in (doubleacc m0 w0)^{t_1}
```

This leads to supercompiling a new state $\mathcal{S}'_2$:

```
let doubleacc n v = ...
    w0 = (S (S v))^t0
in (doubleacc m0 w0)^t1
```

Note that *tagBag* $\mathcal{S}_0 \trianglelefteq$ *tagBag* $\mathcal{S}'_2$, but due to the rollback we are behaving as if $\mathcal{S}_0$ failed the termination test, so $\mathcal{S}_0$ will not be in the *History*. Therefore, after *reduce* and *split*, we drive $\mathcal{S}'_3$:

```
let doubleacc m w = ...
    w0 = (S (S v))^t0
    w1 = (S (S w0))^t0
in (doubleacc m1 w1)^t1
```

However, *tagBag* $\mathcal{S}'_2 \trianglelefteq$ *tagBag* $\mathcal{S}'_3$ and so we will roll back to $\mathcal{S}'_2$ and act as if the termination test had failed there, and the resulting *split* will drive a subcomponent (from the body of *doubleacc*) which just ties back to $\mathcal{S}'_1$, leading to the final program:

```
let h0 n = let doubleacc n v = h1 n v
               v = S Z
           in doubleacc n v
    h1 n v = case n of Z    → v
                       S m0 → h2 0 m v
    h2 m0 v = let doubleacc n v = h1 n v
                  w0 = S (S v)
              in doubleacc m0 w0
in h0 n
```

This result is much better than what we got without rollback, since we have eliminated the useless loop-peeling.

Implementing *sc*-rollback So we would also like to roll back in *sc*, but doing so is complicated by the *ScpM* monadic structure—we must somehow roll back the monad-carried information as well.

The easiest way to implement this is by making *ScpM* an exception monad, in which rollback is triggered by throwing an exception. However, rollback should not revert to the immediately-enclosing invocation of *sc* but rather to the invocation that processed the *State* that is $\trianglelefteq$ the current state. So we need to throw an exception that will only be caught by the "right" handler. An elegant way to express this idea is with a single new primitive in the *ScpM* monad:

```
type Throw c = ∀b.c → ScpM b
catchScpM :: (Throw c → ScpM a)   -- Action to try
          → (c → ScpM a)          -- Handler
          → ScpM a
```

The second argument is the easy one: it is the exception handler, invoked if the first argument throws an exception. The first argument is the action to run in the scope of the exception handler, but it needs to know how to throw an exception to this particular invocation of *catchScpM*. So *catchScpM* applies its first argument to a freshly-minted "how to throw" function. This latter function takes a value of the type expected by the handler, c in this signature, and throws the exception. This signature allows the code that raises the exception to pass a value of type c to the handler, to communicate some information about the failure.

It is straightforward to implement *catchScpM* by changing the internal implementation of the *ScpM* to use a continuation-passing monad [Wadler, 1992].

Given *catchScpM*, we can implement a version of *sc* with rollback. We make use of the same enhanced *terminate* function, but this time the “extra information” passed to *terminate* and returned by *Stop* is the “how to throw function”.

```
type ThrowTerm = () → ScpM Term
sc′ :: History ThrowTerm → State → ScpM Term
sc′ hist 𝒮
      =   check ‘catchScpM‘ (λ() → split (sc hist) 𝒮)
  where
    check throw = case terminate hist 𝒮 throw of
      Stop throw_old → throw_old ()
      Continue hist′ → split (sc hist′) (reduce 𝒮)
```

If we are forced to terminate (the *Stop* branch), then instead of continuing from the current *State* (which triggered the termination condition), we raise an exception using the exception raiser stored with the *State* which “blew the whistle”. When resuming execution from an exception handler, we know that supercompiling the state associated with that handler eventually caused the termination criteria to fire. Therefore, we act as if the state had failed the termination test and do not reduce it before splitting. It is remarkable how little the structure of the supercompiler is disturbed by this change.

For now, the “exception type” c in the type of *catchScpM* is (). However, we will instantiate it with a more interesting type when we consider generalisation in Section 6.2.

Note that the *History* at the point we raise an exception (with $\mathit{throw}_{\mathit{old}}$) may be longer than the *History* at the point we roll back to. In fact, it is not necessary to preserve this longer history when we rollback—we can make do with the shorter history at the point the exception handler was installed. Nonetheless, this does not affect termination of the supercompiler (Section 6.1.3).[4]

One interesting question is what should happen to the *ScpM*-carried information when we rollback. In particular, in between the time the exception handler was installed and when an exception is raised we may have completed supercompilation of some new h-functions—what happens to those when we roll back? One strategy would be to discard them, on the basis that since we have rolled back they will not necessarily be used. However, we found that in practice these specialisations were often useful even after rolling back—retaining generated h-functions caused the supercompilation time of the *digitsofe2* benchmark [Partain, 1993] to decrease by 85%.

For this reason, our supercompiler retains as many h-functions as possible when rolling back. One subtlety is that in between the point at which the exception handler is installed and the point at which the exception is raised we may have made some new promises that have not yet been fulfilled. If we are to roll back to the handler, the supercompilation of these promises is aborted and these promises can never be fulfilled—so we *do* have to discard any h-functions (and their dependents) that have tied back on the basis of those promises. All of this is neatly hidden away inside the implementation of *catchScpM*.

[4] Using the longer history can actually be *dangerous* for termination, since invoking the exception handlers embedded in the history may allow us to “roll back” to an invocation of *sc* which has already been rolled back past by the use of another rollback. This can easily lead to non-termination.

6.1.3 Termination and correctness of rollback

Rollback does not require any major changes to the supercompiler correctness argument. When we roll back, the state we roll back to is just *split* instead of being *reduce*d and then *split*, which is clearly fine from a correctness perspective.

However, it is less obvious that a supercompiler with rollback still terminates. The way to see that it does is to realise that each promise on a chain of recursive invocations of sc' can only be rolled back to at most once. The reason for this is that when a later invocation of sc' rolls back to an earlier invocation $sc'\ opt\ \mathcal{S}$, the supercompiler continues by just *split*ting the $\mathcal{S}$ state without recording it in the termination history.

This explanation may seem counter-intuitive, as our supercompiler is only guaranteed to termination because we *do* extend the termination history. However, in this case the opposite is true: we are only guaranteed to terminate because we *do not* extend it.

To see why this is so, consider what would happen if $\mathcal{S}$ were recorded in the history given to *split* after a rollback. Executing $sc\ hist\ \mathcal{S}$ (which generates corresponding exception-throwing function *throw*), we might easily get into a situation where:

1. $terminate\ hist\ \mathcal{S}\ throw\ =\ Continue\ hist'$

2. $sc\ hist\ \mathcal{S}$ invokes $sc\ hist'\ \mathcal{S}'$

3. $terminate\ hist'\ \mathcal{S}'\ throw'\ =\ Stop\ throw$, so the recursive invocation calls *throw*

4. The code we roll back to invokes $split\ (sc\ hist')\ \mathcal{S}$ (note that using $hist'$ instead of *hist* here is how this example differs from what our sc' function actually does)

5. But by coincidence $split\ (sc\ hist')\ \mathcal{S}$ splits to the same state we would recursively supercompile normally, so we recursively invoke $sc\ hist'\ \mathcal{S}'$

We are now in a loop which continually recurses between $sc\ hist'\ \mathcal{S}'$ and $split\ (sc\ hist')\ \mathcal{S}$.

By not extending the history upon rollback we can avoid this problem. Furthermore, we do not risk non-termination by not extending the history since, as we proved in Section 3.7.3, it is impossible to have an infinite chain of recursive invocations which consist only of *split* steps.

6.2 Generalisation

A shortcoming of the supercompiler as described so far is that it does not make use of the standard technique of *generalisation* [Turchin, 1988; Burstall and Darlington, 1977] to guess good induction hypotheses. Generalisation is an important strategy for optimising programs whose supercompilation drives sequences of *State*s that do not tie back—such as those programs that make use of accumulating parameters.

Note that there is a terminology clash between the supercompilation technique of generalisation (which we cover in this section) and the notion of a most-specific-generalisation of two states or terms (covered in Chapter 5). The relationship between the two concepts is that an algorithm for finding the most-specific-generalisation may be useful in the implementation of a generalisation algorithm for the supercompiler (Section 6.2.3), but not all generalisation algorithms need make use of the MSG.

6.2.1 The problem

An example of a function for which generalisation helps the supercompiler achieve optimisation is *foldl*:

$$\begin{array}{lll}
\mathit{foldl}\ c\ n\ \mathit{xs} = (\mathbf{case}\ \mathit{xs}^{t_1}\ \mathbf{of} & [\,] & \to n \\
 & (y : \mathit{ys}) & \to \mathbf{let}\ m = (c\ n\ y)^{t_4} \\
 & & \quad\ \ \mathbf{in}\ (((\mathit{foldl}^{t_2}\ c)^{t_3}\ m)^{t_5}\ \mathit{ys})^{t_6})^{t_7}
\end{array}$$

Now, suppose we supercompile the invocation $((\mathit{foldl}^{t_a}\ (\wedge)^{t_b}\ n)^{t_c}\ \mathit{xs})^{t_d}$ which might be the implementation of Haskell's standard *and* function. We will call the corresponding state $\mathcal{S}_0$. After reduction and splitting, we will drive the subcomponent state $\mathcal{S}_1$:

$$\begin{array}{l}
\mathbf{let}\ \mathit{foldl}\ c\ n\ \mathit{xs} = \ldots \\
\quad (\wedge) = \ldots \\
\quad \mathit{m0} = ((\wedge)\ n\ \mathit{y0})^{t_4} \\
\mathbf{in}\ (((\mathit{foldl}^{t_2}\ (\wedge))^{t_3}\ \mathit{m0})^{t_5}\ \mathit{ys0})^{t_6})\ t_7
\end{array}$$

The tag-bag for $\mathcal{S}_0$ is $\{\!\{t_b, t_c, t_d, t_8, t_8, t_9\}\!\}$ and for $\mathcal{S}_1$ is $\{\!\{t_3, t_4, t_5, t_6, t_8, t_8, t_9\}\!\}$, so the termination criterion allows us to continue. After reducing and splitting, we drive $\mathcal{S}_2$:

$$\begin{array}{l}
\mathbf{let}\ \mathit{foldl}\ c\ n\ \mathit{xs} = \ldots \\
\quad (\wedge) = \ldots \\
\quad \mathit{m0} = ((\wedge)\ n\ \mathit{y0})^{t_4} \\
\quad \mathit{m1} = ((\wedge)\ \mathit{m0}\ \mathit{y1})^{t_4} \\
\mathbf{in}\ (((\mathit{foldl}^{t_2}\ (\wedge))^{t_3}\ \mathit{m1})^{t_5}\ \mathit{ys1})^{t_6})\ t_7
\end{array}$$

The tag-bag for $\mathcal{S}_2$ is $\{\!\{t_3, t_4, t_4, t_5, t_6, t_8, t_8, t_9\}\!\}$. Considered as a set, this has the same tags as the tag-bag for $\mathcal{S}_1$, but a greater multiplicity for t_4—the "growing tag"—so the termination test tells us to stop. The resulting call to *split* without an intervening *reduce* causes us to supercompile the following subcomponent, which just represents a totally unspecialised version of *foldl*:

$$\begin{array}{lll}
\mathbf{let}\ \mathit{foldl}\ c\ n\ \mathit{xs} = \ldots & & \\
\mathbf{in}\ (\mathbf{case}\ \mathit{xs}^{t_1}\ \mathbf{of} & [\,] & \to n \\
 & (y : \mathit{ys}) & \to \mathbf{let}\ m = (c\ n\ y)^{t_4} \\
 & & \quad\ \ \mathbf{in}\ (((\mathit{foldl}^{t_2}\ c)^{t_3}\ m)^{t_5}\ \mathit{ys})^{t_6})^{t_7}
\end{array}$$

Ideally, we would hope that supercompiling the input program *foldl* $(\wedge)$ *n xs* would have created an output program containing a copy of *foldl* that has been specialised for its function argument. This example has shown that in fact supercompilation only succeeds in specialising the first two iterations of *foldl* on the function argument: all later iterations are entirely unspecialised.

Completing the example, we find that our final output program will contain a peeled, unrolled[5], unspecialised copy of *foldl*:

$$\begin{array}{llll}
\mathbf{let}\ \mathit{h0}\ n\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of}\ [\,] \to n & & & \\
\qquad (\mathit{y0} : \mathit{ys0}) & \to \mathit{h1}\ n\ \mathit{y0}\ \mathit{ys0} & & \\
\quad \mathit{h1}\ n\ \mathit{y0}\ \mathit{ys0} = \mathbf{case}\ \mathit{ys0}\ \mathbf{of} & & [\,] & \to (\wedge)\ n\ \mathit{y0} \\
 & & (\mathit{y1} : \mathit{ys1}) & \to \mathit{h2}\ n\ \mathit{y0}\ \mathit{y1}\ \mathit{ys1} \\
\quad \mathit{h2}\ n\ \mathit{y0}\ \mathit{y1}\ \mathit{ys1} = \mathbf{let}\ \mathit{foldl}\ c\ n\ \mathit{xs} = \mathit{h3}\ c\ n\ \mathit{xs} & & &
\end{array}$$

[5] The techniques discussed in Section 6.1.2 will prevent this peeling and unrolling

```
                            m0 = (∧) n y0
                            m1 = (∧) m0 y1
                         in foldl (∧) m1 ys1
   h3 c n xs = case xs of []          → n
                          (y0 : ys0) → h4 c n y0 ys0
   h4 c n y0 ys0 = case ys0 of []         → c (c n y0) y1
                               (y1 : ys1) → h5 c n y0 y1 ys1
   h5 c n y0 y1 ys1 = let foldl c n xs = h3 c n xs
                          m0 = c n y0
                          m1 = c m0 y1
                      in foldl c m1 ys1
in h0 n xs
```

The problem is that the only thing our standard algorithm can do when the termination test fires is *split*, which forces us to supercompile a copy of *foldl* which is not specialised on *any* of its arguments. What we want is to *specialise* with respect to the *one* argument that is not changing, namely (∧), but *parameterise* over the argument that *is* changing, the accumulator n.

The problem of how to continue when the supercompiler termination criteria fails is well known and is solved by the choice of some *generalisation* method [Turchin, 1988; Sørensen and Glück, 1995]. The goal of generalisation is to use the specialisations generated thus far to infer a "more general" specialisation that subsumes all of them.

In this section, we will discuss two methods of generalisation:

- Firstly, we will discuss a generalisation method suitable for supercompilers that make use of tag-bags for their termination tests, based upon what we call *growing-tags*. The advantage of this generalisation method is that it is very easy to implement on top of the machinery we have already defined for *split*, and gives good results in many common cases: Section 6.2.2.

- Secondly, we discuss a more standard generalisation method using an adaptation of the most-specific generalisation (MSG): Section 6.2.3. We generally find that the MSG gives better results than growing-tags generalisation (Section 7.5). Nonetheless, growing-tags generalisation may still be useful in the implementation of a supercompiler that only incorporates a splitter and term matcher, and does not have a full MSG implementation.

6.2.2 Growing-tag generalisation

The idea behind growing-tag generalisation is that when the termination test fails there is usually at least one tag that can be "blamed" for this, in the sense that the number of occurrences of that tag has grown since we supercompiled the prior state, causing the size of the tag-bag to rise and hence the termination test to fire.

In our *foldl* example, there is indeed such a "growing" tag—t_4. We can use the set of growing tags to generalise the *State* being supercompiled by somehow *removing from the state all syntax tagged with the growing tag(s)* and then recursively supercompiling the thus-eviscerated state. In our example from page 108, when the termination test fails on $\mathcal{S}_2$, we would residualise the heap bindings for *m0* and *m1* because they are tagged (at the root) with the growing tag t_4, and recursively supercompile a subcomponent which is really just a version of $\mathcal{S}_2$ which lacks those bindings:

```
let foldl c n xs = ...
      (∧) = ...
in ((foldl^t2 (∧))^t3 m1)^t5 ys1) t6
```

Now, since the accumulator *m2* has been residualised, this recursively supercompiled state can immediately tie back to the h function for $\mathcal{S}_0$, and we get a loop that implements *foldl*:

```
let h0 n xs = case xs of [ ]          → n
                         (y0 : ys0) → h1 n y0 ys0
    h1 n y0 ys0 = case ys0 of [ ]          → (∧) n y0
                              (y1 : ys1) → h2 n y0 y1 ys1
    h2 n y0 y1 ys1 = let m0 = (∧) n y0
                         m1 = (∧) m0 y1
                     in h0 m1 ys1
in h0 n xs
```

The resulting loop has still been unrolled (because it took two *foldl* inlinings before we could spot the pattern of growth), but unlike the output program produced by the supercompiler without generalisation, the loop *is* specialised on the (∧) argument.

This technique combines very nicely with rollback—with rollback enabled, we can simply force residualisation of any syntax in the *older State* that is tagged with a growing tag. So for our *foldl* example, failure of the termination test would cause us to roll-back to $\mathcal{S}_1$, at which point we would notice that the (∧) *n y0* thunk is tagged with the growing tag t_4 and thus residualise it. The resulting program will be an optimal specialised loop that has not been unrolled.

Given two states, we can compute the set of growing tags between them as follows:

```
bagMinus :: Ord a ⇒ Bag a → Bag a → Bag a
bagToSet :: Ord a ⇒ Bag a → Set a   -- Discards duplicates
type Growing = Set Tag
findGrowing :: State → State → Growing
findGrowing 𝒮1 𝒮2 = bagToSet $ tagBag 𝒮2 `bagMinus` tagBag 𝒮1
```

In general, it might be the case that no tag is growing with respect to the previous tag bag, which happens if and only if the latest tag-bag is exactly equal to the older one. In this case we have to fall back to an alternative generalisation method, or else (if all other generalisation methods also fail) *split* in the same way as a supercompiler without generalisation would.

The plumbing needed to implement the growing-tag generalisation method can be added to the basic supercompiler in a very straightforward manner. We show how it can be be added to the supercompiler with *sc*-rollback that we defined in Section 6.1.2. All that needs to be done is to replace the definition of *sc′* with the following:

```
type ThrowTerm = State → ScpM Term
sc′ :: History ThrowTerm → State → ScpM Term
sc′ hist 𝒮 = check `catchScpM` (λ𝒮′. tryTags (sc hist) 𝒮 𝒮′ `orElse` split (sc hist) 𝒮)
  where
    check throw = case terminate hist 𝒮 throw of
      Stop throw_old    → throw_old 𝒮
      Continue hist′ → split (sc hist′) (reduce 𝒮)
```

$$
\begin{array}{l}
\mathit{orElse} :: \mathit{Maybe}\ a \to a \to a \\
\mathit{orElse}\ (\mathit{Just}\ x)\ \_ = x \\
\mathit{orElse}\ \mathit{Nothing}\ x = x
\end{array}
$$

Note that we instantiated the "exception type" parameter c of *catchScpM* with *State* (cf. Section 6.1.2).

To make use of the growing-tags information, we introduce *tryTags*, a function similar to *split*, but which only residualises those parts of the input *State* that are marked with a growing tag (and anything that those residualised portions transitively depend on). Because it is possible that no tag may be growing, *tryTags* may fail, and in those cases we fall back to using *split*. In our implementation, *tryTags* shares almost all of its code with *split*, and can be defined using the $\widehat{\mathit{push}}$ and $\mathit{traverse}\mathring{\mathit{State}}$ functions we introduced in Chapter 4 as follows:

$$
\begin{array}{l}
\textbf{type}\ \mathit{Generaliser} = (\mathit{State} \to \mathit{ScpM\ \ Term}) \\
\qquad\qquad\qquad\qquad \to \mathit{State} \to \mathit{State} \\
\qquad\qquad\qquad\qquad \to \mathit{Maybe}\ (\mathit{ScpM\ \ Term}) \\
\mathit{tryTags} :: \mathit{Generaliser} \\
\mathit{tryTags}\ \mathit{opt}\ \mathcal{S}\ \mathcal{S}' \ = \ \mathit{fmap}\ (\lambda f. f\ \mathit{opt})\ \ (\mathit{tryTags}'\ \mathit{growing}\ \mathcal{S}) \\
\quad \textbf{where}\ \ \mathit{growing} \ = \ \mathit{findGrowing}\ \mathcal{S}\ \mathcal{S}'
\end{array}
$$

$$
\begin{array}{l}
\mathit{tryTags}' :: \mathit{Growing} \to \mathit{State} \\
\qquad\qquad \to \mathit{Maybe}\ ((\mathit{State} \to \mathit{ScpM\ \ Term}) \to \mathit{ScpM\ \ Term}) \\
\mathit{tryTags}'\ \mathit{growing}\ \ \langle H \,|\, e \,|\, K \rangle \\
\quad = \begin{cases} \mathit{Just}\ \ (\lambda \mathit{opt}.\, \mathit{traverse}\mathring{\mathit{State}}\ \mathit{opt}\ \mathring{\mathcal{S}}) & N \neq \varnothing \\ \mathit{Nothing} & \text{otherwise} \end{cases} \\
\quad \textbf{where}\ \ \mathring{\mathcal{S}} \ = \ \widehat{\mathit{push}}(N)\,\langle H \,|\, e \,|\, K \rangle \\
\qquad\qquad\quad N \ = \ \quad \{x \mid x{:}\tau \mapsto e^t \in H,\ t \in \mathit{growing}\} \\
\qquad\qquad\qquad\quad \cup \ \ \{i \mid K[i] = \kappa^t,\ t \in \mathit{growing}\}
\end{array}
$$

The presentation of *tryTags* as a call to an auxiliary function *tryTags′* is to set us for a slight refinement of rollback in Section 6.2.4. The check for a non-empty generalisation set N is necessary for correctness (Section 6.2.8).

Note that *tryTags′* calls $\widehat{\mathit{push}}$ with a generalisation set which *never* contains •. This reflects the fact that when generalising we never want to residualise the focus of the state, only some fraction of the state's heap or stack. This is in stark contrast to invocations of $\widehat{\mathit{push}}$ by *split*, which always use the generalisation set $\{\bullet\}$ (Section 4.3.3).

In practice, there will sometimes be several growing tags, in which case supercompilation can continue if we residualise syntax tagged with *any* non-empty subset of the growing tags. Presently, our implementation does not take advantage of this freedom: any syntax tagged with a growing tag is residualised by *tryTags*.

To exemplify *tryTags*, recall our previous example's $\mathcal{S}_2$, which we wanted to generalise:

$$
\begin{array}{l}
\textbf{let}\ \mathit{foldl}\ c\ n\ \mathit{xs} = \ldots \\
\qquad (\wedge) = \ldots \\
\qquad \mathit{m0} = ((\wedge)\ n\ \mathit{y0})^{t_4} \\
\qquad \mathit{m1} = ((\wedge)\ \mathit{m0}\ \mathit{y1})^{t_4} \\
\textbf{in}\ (((\mathit{foldl}^{t_2}\ (\wedge))^{t_3}\ \mathit{m1})^{t_5}\ \mathit{ys1})^{t_6}
\end{array}
$$

Generalisation of this state with the growing tag set $\{t_4\}$ would call

$$
\widehat{\mathit{push}}(\{\mathit{m0}, \mathit{m1}\}) \left\langle \begin{array}{l} \mathit{foldl} \mapsto \ldots, (\wedge) \mapsto \ldots, \\ \mathit{m0} \mapsto ((\wedge)\ n\ \mathit{y0})^{t_4}, \mathit{m1} \mapsto ((\wedge)\ \mathit{m0}\ \mathit{y1})^{t_4} \end{array} \middle|\, \mathit{foldl}^{t_2} \,\middle|\, \bullet\ (\wedge)^{t_3}, \bullet\ \mathit{m1}^{t_5}, \bullet\ \mathit{ys1}^{t_6} \right\rangle
$$

which gives the pushed state:

$$\left\langle \begin{array}{l} \mathit{m0} \mapsto \langle (\wedge) \mapsto \ldots \,|\, ((\wedge)\ n\ \mathit{y0})^{t_4} \,|\, \epsilon\rangle, \\ \mathit{m1} \mapsto \langle (\wedge) \mapsto \ldots \,|\, ((\wedge)\ \mathit{m0}\ \mathit{y1})^{t_4} \,|\, \epsilon\rangle \end{array} \middle| \left\langle \mathit{foldl} \mapsto \ldots, (\wedge) \mapsto \ldots \middle| \mathit{foldl}^{t_2} \middle| \bullet\ (\wedge)^{t_3}, \bullet\ \mathit{m1}^{t_5}, \bullet\ \mathit{ys1}^{t_6} \right\rangle \middle| \epsilon \right\rangle$$

Observe that recursive supercompilation of the subcomponent state

$$\left\langle \mathit{foldl} \mapsto \ldots, (\wedge) \mapsto \ldots \middle| \mathit{foldl}^{t_2} \middle| \bullet\ (\wedge)^{t_3}, \bullet\ \mathit{m1}^{t_5}, \bullet\ \mathit{ys1}^{t_6} \right\rangle$$

will tie back to $\mathcal{S}_0$, just as we claimed.

6.2.3 MSG-based generalisation

Growing-tag generalisation is simple to implement by reusing code from the implementation of *split*, and we find it to be fairly effective in practice. However, it suffers from the problem that if a binding is generalised by growing-tag generalisation, that binding is generalised away from all use sites. In contrast, generalisation based on the MSG algorithm of Chapter 5 is capable of making a generalisation decision on a per-use-site basis. To illustrate the difference, consider the term **let** $x = e$ **in** (x, x). Growing-tag generalisation is able to generalise x away from both use sites, yielding (x, x), but it is incapable of generalising x away from a subset of use sites to yield a term such as **let** $x = e$ **in** (x, y).

This problem is made particularly acute by the combination of data constructor wrappers (necessary to handle the partially-applied data constructors: Section E.3.6) and speculation (Section 6.3). With this combination of features, for any given data constructor, all saturated occurrences of that data constructor will be assigned the same tag.

This in turn means that if we supercompile a function with an accumulating list argument (such as the standard efficient implementation of *reverse*) the tag associated with the (:) data constructor will be implicated as a growing tag, leading to *every* cons-cell heap-bound in the current state being generalised away, regardless of whether those cells are used in the accumulator or not.

A small demonstration of the problem that makes use of a data constructor wrapper but does not rely on speculation is the following state $\mathcal{S}_0$:

$$\begin{array}{l}
\mathbf{let}\ (:)_{\mathrm{wrap}}\ x\ \mathit{xs} = ((:)\ x\ \mathit{xs})^{t_0} \\
\qquad \mathit{replicacc}\ n\ \mathit{xs} = \mathbf{case}\ n\ \mathbf{of} \\
\qquad\quad Z \quad\ \to \mathit{xs} \\
\qquad\quad S\ m \to \mathbf{let}\ \mathit{as} = ((:)_{\mathrm{wrap}}\ x\ \mathit{xs})^{t_1} \\
\qquad\qquad\qquad \mathbf{in\ case}\ \mathit{as}\ \mathbf{of}\ \_ : \_ \to \mathit{replicacc}\ m\ \mathit{as} \\
\qquad \mathit{bs} = ((:)_{\mathrm{wrap}}\ c\ \mathit{cs})^{t_2} \\
\mathbf{in\ case}\ \mathit{bs}\ \mathbf{of}\ \_ : \_ \to \mathbf{case}\ \mathit{replicacc}\ n\ \mathit{xs}\ \mathbf{of} \\
\qquad [\,] \quad \to \mathbf{case}\ \mathit{bs}\ \mathbf{of}\ \_ : \_ \to \mathit{True} \\
\qquad \_ : \_ \to \mathbf{case}\ \mathit{bs}\ \mathbf{of}\ \_ : \_ \to \mathit{True}
\end{array}$$

After reducing and splitting $\mathcal{S}_0$, we will supercompile $\mathcal{S}_1$:

$$\begin{array}{l}
\mathbf{let}\ (:)_{\mathrm{wrap}}\ x\ \mathit{xs} = \ldots \\
\qquad \mathit{replicacc}\ n\ \mathit{xs} = \ldots \\
\qquad \mathit{bs} = ((:)\ c\ \mathit{cs})^{t_0} \\
\qquad \mathit{as0} = ((:)\ x\ \mathit{xs})^{t_0} \\
\mathbf{in\ case}\ \mathit{replicacc}\ \mathit{m0}\ \mathit{as0}\ \mathbf{of} \\
\qquad [\,] \quad \to \mathbf{case}\ \mathit{bs}\ \mathbf{of}\ \_ : \_ \to \mathit{True} \\
\qquad \_ : \_ \to \mathbf{case}\ \mathit{bs}\ \mathbf{of}\ \_ : \_ \to \mathit{True}
\end{array}$$

After one more round of reducing and splitting, we get $\mathcal{S}_2$:

```
let $(:)_{\text{wrap}}$ x xs = ...
    replicacc n xs = ...
    bs = $((:)\ c\ cs)^{t_0}$
    as0 = $((:)\ x\ xs)^{t_0}$
    as1 = $((:)\ x\ as0)^{t_0}$
in case replicacc m1 as1 of
    [ ]   → case bs of _: _ → True
    _: _  → case bs of _: _ → True
```

We have that *tagBag* $\mathcal{S}_1 \trianglelefteq$ *tagBag* $\mathcal{S}_2$, so the termination test fails, and growing-tags generalisation will attempt to supercompile the following generalised state $\mathcal{S}_3$:

```
let $(:)_{\text{wrap}}$ x xs = ...
    replicacc n xs = ...
in case replicacc m1 as1 of
    [ ]   → case bs of _: _ → True
    _: _  → case bs of _: _ → True
```

Notice that we have generalised away the heap binding for *bs*, even though it was not involved in the accumulator of *replicacc*. This prevents us from simplifying away the expression **case** *bs* **of** _: _ → *True* when the supercompiler eventually ends up driving it.

In contrast, the MSG would have supercompiled the generalised state $\mathcal{S}_3'$:

```
let $(:)_{\text{wrap}}$ x xs = ...
    replicacc n xs = ...
    bs = $((:)\ c\ cs)^{t_0}$
in case replicacc m1 as1 of
    [ ]   → case bs of _: _ → True
    _: _  → case bs of _: _ → True
```

This is much better because we lose less information.

As we have already defined a *msg* algorithm, implementing MSG-based generalisation in our framework is almost as straightforward as implementing growing-tag generalisation. To do so, we replace the call to *tryTags* we had in the modified *sc*′ function of Section 6.2.2 with a call to *tryMSG*, defined as follows:

```
tryMSG :: Generaliser
tryMSG opt $\mathcal{S}$ $\mathcal{S}'$ = case msg $\mathcal{S}'$ $\mathcal{S}$ of
  Just (_, $\mathcal{S}_{common}$, inst) → fmap (λf → f opt) (tryMSG′ $\mathcal{S}_{common}$ inst)
  Nothing                      → Nothing

tryMSG′ :: State → (Heap, Subst, Stack)
             → Maybe ((State → ScpM Term) → Scp Term)
tryMSG′ $\mathcal{S}_{common}$ ⟨H | θ | K⟩
  | $H \not\equiv \epsilon \vee K \not\equiv \epsilon$
  = Just $ λopt → do e ← opt $\mathcal{S}_{common}$
                           split opt (H, substTerm θ e, K)
  | otherwise
  = Nothing
```

As in the case of the growing-tag generaliser *tryTags*, we have defined *tryMSG* using an auxiliary function *tryMSG*′ to prepare us for Section 6.2.4. The check for a non-empty heap or stack in *tryMSG*′ is necessary for correctness (Section 6.2.8).

Of course, the MSG-based and tag-based generalisation methods can be combined, using a combinator such as the following:

```
plusGeneraliser :: Generaliser → Generaliser → Generaliser
plusGeneraliser gen1 gen2 opt 𝒮 𝒮′ = gen1 opt 𝒮 𝒮′ ‘mplus‘ gen2 opt 𝒮 𝒮′

mplus :: Maybe a → Maybe a → Maybe a
mplus (Just x) _       = Just x
mplus Nothing mb_x = mb_x
```

In our implementation, we attempt MSG-based generalisation first, and if that fails (which can happen if *msg* fails because e.g. the focus of one input state is x and the other focus is a λ, or because the guard in *tryMSG′* fails) we attempt tag-based generalisation. If both generalisation methods fail we fall back on *split*. In practice, we find that for our benchmark suite (Chapter 7) none of the benchmarks experience a situation in which MSG fails to generalise but where growing-tags generalisation is able to find a generalisation.

The generaliser can be thought of as another independent module in our modular presentation of the supercompilation algorithm (Chapter 3).

6.2.4 Rolling back to generalise

In the implementation of *sc′* with generalisation that we proposed in Section 6.2.2, if the termination test failed we *always* rolled back and then tried to generalise the earlier, shallower state that we rolled back to. Although this is not incorrect, it does not necessarily produce the best code.

One common example where this strategy fails is if we are using MSG-based generalisation and the new state is an instance of the old one. This is such a common phenomenon that it occurs in the example we used to introduce generalisation in Section 6.2, the supercompilation of the term $((\mathit{foldl}^{t_a}\ (\wedge)^{t_b}\ n)^{t_c}\ \mathit{xs})^{t_d}$. If you recall, in that example we first recursively drove $\mathcal{S}_1$:

$$\begin{array}{l}
\mathbf{let}\ \mathit{foldl}\ c\ n\ \mathit{xs} = \ldots \\
\quad (\wedge) = \ldots \\
\quad \mathit{m0} = ((\wedge)\ n\ \mathit{y0})^{t_4} \\
\mathbf{in}\ (((\mathit{foldl}^{t_2}\ (\wedge))^{t_3}\ \mathit{m0})^{t_5}\ \mathit{ys0})^{t_6})\ t_7
\end{array}$$

And then $\mathcal{S}_2$:

$$\begin{array}{l}
\mathbf{let}\ \mathit{foldl}\ c\ n\ \mathit{xs} = \ldots \\
\quad (\wedge) = \ldots \\
\quad \mathit{m0} = ((\wedge)\ n\ \mathit{y0})^{t_4} \\
\quad \mathit{m1} = ((\wedge)\ \mathit{m0}\ \mathit{y1})^{t_4} \\
\mathbf{in}\ (((\mathit{foldl}^{t_2}\ (\wedge))^{t_3}\ \mathit{m1})^{t_5}\ \mathit{ys1})^{t_6})\ t_7
\end{array}$$

The tag-bags of these two states are embedded, and so the termination test fails. With growing-tags generalisation, we would roll back to $\mathcal{S}_1$ and then residualise the binding *m0* which is tagged with the growing tag t_4. However, $\mathcal{S}_2$ is an instance of $\mathcal{S}_1$ and so with MSG-based generalisation we are unable to generalise $\mathcal{S}_1$ (the check in *tryMSG′* for a non-empty instantiating heap or stack will fail), and consequently with the version of *sc′* presented in Section 6.2.2 we will end up *split*ting $\mathcal{S}_1$ to produce the final code:

```
let h0 n xs = case xs of
        []          → n
        (y0 : ys0) → h1 n y0 ys0
    h1 n y0 ys0 = let foldl c n xs = h2 c n xs
                      m0 = (∧) n y0
                  in foldl (∧) m0 ys0
    h2 c n xs = case xs of [ ]        → n
                          (y0 : ys0) → let foldl c n xs = h2 c n xs
                                           m0 = (∧) n y0
                                       in foldl (∧) m0 ys0
in h0 n xs
```

Note that the inner loop *h2* is not specialised on the functional argument *c*, which we can see statically will always be instantiated with (∧). If we had *not* rolled back and instead generalised $\mathcal{S}_2$, we would have got the much better output code:

```
let h0 n xs = case xs of [ ]        → n
                         (y0 : ys0) → h1 n y0 ys0
    h1 n y0 ys0 = case ys0 of [ ]        → (∧) n y0
                              (y1 : ys1) → h2 n y0 y1 ys1
    h2 n y0 y1 ys1 = let m0 = (∧) n y0
                     in h1 m0 y1 ys1
in h0 n xs
```

The fundamental problem is that *split*ting can be seen as a very aggressive form of generalisation (in the sense that they both throw away information which we have available at compile time with the goal of avoiding non-termination), and is consequently generally much worse for the quality of the output code than the controlled information loss we get from generalisation. Therefore, if we have to make a choice between:

- *Either* not rolling back to the earlier embedded state (i.e. not reducing code bloat), but being able to successfully generalise the later state (improving optimisation)
- *Or* rolling back to the earlier embedded state (i.e. reducing code bloat) but being forced to *split* rather than generalise (impeding optimisation)

If given this choice, we should prefer not rolling back, so that we can avoid having to *split*.

It is worth noting that not being able to generalise *after* rolling back even though we can generalise *before* the roll back usually only affects MSG-based generalisation. This problem very rarely occurs when using the growing-tag generalisation, because it tends to be the case that if the new state is generalisable then the old one will be as well.

The reason for this is that the new state will be generalisable if there is at least one growing tag, and there is at least one heap binding or stack frame in the new state tagged with the growing tag. However, since the tag-bags for the old and the new state are embedded, it must be the case that at least one bit of syntax (heap binding, stack binding or focus) in the old state was also tagged with that growing tag. As long it is not the old state's focus which is so tagged, it will therefore be the case that at least one heap binding or stack frame in the *old* state tagged with the growing tag, which means that growing-tag generalisation of the old state will succeed.

It is straightforward to change the implementation of *sc′* so that it does not roll back if doing so would lose a generalisation opportunity:

```
sc′ hist S = check ‘catchScpM‘
               (λmb_gen_old. case mb_gen_old of Just gen_old → gen_old (sc hist)
                                                   Nothing      → split (sc hist) S)
    where
      check throw = case terminate hist S (S, throw) of
        Stop (S_old, throw_old) → case tryMSG S_old S of
          (Nothing,     Nothing)      → throw_old Nothing
          (Just gen_old, _)           → throw_old (Just gen_old)
          (Nothing,     Just gen_new) → gen_new (sc hist)
        Continue hist'          → split (sc hist') (reduce S)
```

Where we make use of a significantly more complicated type for generalisers:

```
type Generaliser = State → State
                      → (Maybe ((State → ScpM Term) → ScpM Term),
                         Maybe ((State → ScpM Term) → ScpM Term))
tryTags, tryMSG :: Generaliser
```

The changes from the initial *Generaliser* type are as follows:

- We simultaneously compute a generalisation for both the "old" (shallower) and "new" states. This is because we do not know which one we will end up generalising.
- Because we don't have access to the *History* that the shallower state was supercompiled in at the time we generalise, we delay the point at which we have to supply it by pushing the demand for the *opt*::*State* → *ScpM Term* function used to recursively supercompile inside the *Maybe* returned by the generaliser.

Although the sc' we defined above uses *tryMSG* as a generaliser, we can adapt both of our previous generalisation methods to this framework straightforwardly, as we conveniently defined them in terms of suitable auxilliary functions $tryTags'$ and $tryMSG'$:

```
tryTags :: Generaliser
tryTags S S'  =  (tryTags' growing S, tryTags' growing S')
  where growing  =  findGrowing S S'

tryMSG :: Generaliser
tryMSG S S' = case msg S S' of
  Just (inst_l, S_common, inst_r) → (tryMSG' S_common inst_l, tryMSG' S_common inst_r)
  Nothing                         → (Nothing, Nothing)
```

6.2.5 MSG-based generalisation and variable specialisation

When supercompiling, it can sometimes happen that we drive a state $f\ x\ x$ and later drive a similar state $f\ y\ z$ which differs from the earlier one only in the sense that it is less specialised on variables. Because these two states will commonly have identical tag bags they will be embedded in the tag-bag termination test, and the supercompiler as described so far will thus *split* the later state $f\ y\ z$ without reducing it (in particular, MSG-based generalisation will not occur because of the triviality checks in the $tryMSG'$ function of Section 6.2.3).

This causes problems in practice, such as when supercompiling the Haskell term $foldr\ (+)\ 0\ [x + y \mid x \leftarrow xs, y \leftarrow xs]$, which can be translated to a state (using the "TQ translation" for list comprehensions [Peyton Jones and Wadler, 1987]) as follows:

```
foldr (+) 0 (go1 xs)
  where
```

```
go1 [ ]      = [ ]
go1 (x : xs) = go2 ys
  where
    go2 [ ]      = go1 xs
    go2 (y : ys) = (x + y) : go2 ys
```

Supercompiling, we find that this program cannot be deforested because an earlier state:

```
let ys = y0 : ys_0
    go1 = ...
    go2_0 [ ]      = go1 xs_0
    go2_0 (y : ys) = (x0 + y) : go2 ys
in foldr (+) n_0 (go2 ys_0)
```

Is embedded by the tag-bag test with the later state:

```
let ys = y0 : ys_0
    go1 = ...
    go2_1 [ ]      = go1 xs_1
    go2_1 (y : ys) = (x1 + y) : go2 ys
in foldr (+) n_1 (go2 ys_1)
```

Note that we can't tie back the later state to the earlier one because there is no way to rename the earlier state to the later one, since the earlier state mentions the free variable *ys_0* twice, whereas the later state uses two distinct free variables in those positions. As a result, the supercompiler is forced to *split* without reducing, which impedes deforestation.

It seems unfortunate that supercompiling an earlier state *f x x* and then a later state *f x y* forces us to *split*. If we had instead supercompiled an earlier state *f y z* and later came to supercompile *f x x* we would simply tie back to the earlier promised *h*-function and hence avoid splitting altogether (Section 5.2.1), so whether or not we can avoid splitting depends on the order in which we see the two states. This is particularly annoying as in practice the vastly more common case is that we supercompile a more-specialised state like *f x x* first, followed by a less-specialised state like *f y z*, so we are forced to *split*.

Happily, there is an improvement we can make to our MSG generalisation which avoids having to make this *split*. The idea is to make the triviality check more discerning, so that if *both* instantiations reported by *msg* are trivial (i.e. have empty instantiating heaps and stacks), we allow generalisation via rollback. This can be expressed in code with this new *tryMSG* function:

```
tryMSG :: Generaliser
tryMSG S S' = case msg S S' of
  Nothing → (Nothing, Nothing)
  Just (inst_l, S_common, inst_r)
    | let allow (H, θ, K) opt = Just $ do e ← opt S_common
                                            split opt (H, substTerm θ e, K)
    → case (nonTrivial inst_l, nonTrivial inst_r) of
        (True, True)   → (allow inst_l, allow inst_r)
        (True, False)  → (allow inst_l, Nothing)
        (False, True)  → (Nothing,      allow inst_r)
        (False, False) → (allow inst_l, Nothing)
```

```
nonTrivial :: (Heap, Subst, Stack) → Bool
nonTrivial (H, _, K) = H ≢ ϵ ∨ K ≢ ϵ
```

The behaviour of this *tryMSG* differs from that of Section 6.2.4 only because of the (*False*, *False*) case, which returns (*allow* $inst_l$, *Nothing*). If this case instead returned (*Nothing*, *Nothing*) then this new *tryMSG* would behave identically to the old one.

However, this one small difference causes a big change: now if we are attempting generalisation between an earlier state $f\ x\ x$ and a new state $f\ y\ z$, MSG-based generalisation will be allowed to generalise via rollback. As a result, the stack of *sc* invocations will be unwound to the point at which $f\ x\ x$ was supercompiled and the variable-generalised state $f\ y\ z$ will be supercompiled in its place.

It is important that we only allow generalisation via *rollback* in the (*False*, *False*) case. If we allowed generalisation without rollback then there is a danger we could end up recursively supercompiling $f\ y\ z$ as a child of a state $f\ y\ z$, which would cause the supercompiler to immediately tie back, forming a black hole **let** $h = h$ **in** h. When we generalise via rollback we do not risk this.

Furthermore, whenever we roll back to generalise we know that the term we will roll back to drive is guaranteed to be *strictly* less variable-specialised than what it replaces. The reason for this is that if we are rolling back from $\mathcal{S}'$ to an earlier state $\mathcal{S}$, then we must already have attempted to tie back $\mathcal{S}'$ to the existing promise for $\mathcal{S}$. If there was any way to instantiate $\mathcal{S}$ to obtain $\mathcal{S}'$ then we would have memoised rather than generalised, so it must be the case that the generalisation of $\mathcal{S}$ and $\mathcal{S}'$ will be strictly less variable-specialised than $\mathcal{S}$. For example, if $\mathcal{S} = f\ a\ b\ b$ and $\mathcal{S}' = f\ c\ c\ d$ then the generalisation could be $f\ a\ b\ c$, which is strictly more general than either side. If $\mathcal{S} = f\ a\ b$ and $\mathcal{S}' = f\ c\ c$ then we would have memoised rather than generalised: if we had (erroneously) failed to memoise and instead rolled back to $\mathcal{S}$ in order to drive the generalisation $f\ a\ b$ as a child, the supercompiler would have returned a black hole **let** $h = h$ **in** h.

6.2.6 Type generalisation

Most of the time, a supercompiler will use generalisation only as a last resort: we only generalise if the sc' termination test fails. The reason for this is that generalisation throws away information, and throwing information away can reduce the amount of optimisation that supercompilation can achieve.

However, while throwing away information is usually a bad idea, throwing away *type* information not only does not impede optimisation (because type information is computationally irrelevant), but also may increase supercompiler speed because promises will be specialised on less type information and so memoisation may tie back more often.

As an example, consider supercompiling the following program:

```
let length = Λα. λxs. case xs of [ ]      → Z
                                 (y : ys) → S (length α ys)
    f xs = length Bool   xs
    g xs = length Int    xs
    h xs = length Double xs
in (f, g, h)
```

With our standard supercompilation algorithm, we get the following output code:

```
let h0 (xs :: [Bool]) = case xs of [ ]      → Z
                                   (y : ys) → S (h0 ys)
```

```
    h1 (xs :: [Int]) = case xs of []       → Z
                                  (y : ys) → S (h1 ys)
    h2 (xs :: [Double]) = case xs of []       → Z
                                     (y : ys) → S (h2 ys)
    f xs = h0 xs
    g xs = h1 xs
    h xs = h2 xs
in (f, g, h)
```

The functions *h0*, *h1* and *h2* are versions of *length* specialised on the *type* arguments *Bool*, *Int* and *Double* respectively. A simple change to the supercompiler to eagerly throw away type information will allow us to common these definitions up into a loop of type $\forall\alpha.[\,a\,] \to Nat$, reducing both output code size and supercompilation time.

Of course, we cannot simply throw away all type information, because Core is a typed language. We may only throw away type information if we are certain that the resulting term will still be well typed. Luckily, we can reuse the versatile *msg* function to do this, as we demonstrated in Section 5.7. To exploit type generalisation using *msg*, we replace our memoiser with one which checks whether the new state being supercompiled has an MSG against any previous promise which *generalises only type information.* For example, if we already had a promise for $(\lambda(xs :: [\,Bool\,]).\,length\ Bool\ xs)$ and *memo* is asked to supercompile $(\lambda(ys :: [\,Int\,]).\,length\ Int\ ys)$, by taking the *msg* of these two states we can deduce that the common term $(\lambda(zs :: [\,\alpha\,]).\,length\ \alpha\ ys)$ is well-typed. Furthermore, this common term has only generalised away type information (i.e. the instantation of α).

The *memo opt* call can take advantage of this discovery by invoking *opt* on the common state $(\lambda(zs :: [\,\alpha\,]).\,length\ \alpha\ ys)$, rather than on the original, ungeneralised state $(\lambda(ys :: [\,Int\,]).\,length\ Int\ ys)$. The advantage of this is that by creating a promise for the type-generalised state, we open the door to extra future tieback opportunities.

Concretely, this can be implemented as follows:

```
memo :: (State → ScpM Term)
     → State → ScpM Term
memo opt 𝒮 = do
  ps ← promises
  let tiebacks = [ return (var (name p) ‘tyApps‘ map (substTyVar θ) (ftvs p)
                                         ‘apps‘ (unit : map (substVar θ) (fvs p)))
                 | p ← ps
                 , Just θ ← [match (meaning p) 𝒮]
                 ]
      tygens =   [ do e ← memo opt 𝒮_common
                      return (substTerm θ_1 e)
                 | p ← ps
                 , Just (θ_0, 𝒮_common, θ_1) ← [typeGeneralise (meaning p) 𝒮]
                 , substNonTrivial θ_1
                 ]
  case tiebacks ++ tygens of
    res : _ → res
    []      → do
      hn ← freshVar
      let (tvs, vs) = freeVars 𝒮
      promise P {name = hn,
                 ftvs = map fst tvs, fvs = map fst vs,
                    meaning = 𝒮}
   e ← opt 𝒮
   bind hn (forAllTys tvs (funTys (unitTy : map snd vs) (stateType 𝒮)))
           (tyLambdas tvs (lambdas ((unit, unitTy) : vs) e))
   return (var hn `tyApps` map fst tvs `apps` map fst (unit : vs))
```

Where we have the following auxilliary functions:

$$\begin{array}{l}
\mathit{substNonTrivial} :: \mathit{Subst} \to \mathit{Bool} \\
\mathit{substNonTrivial}(\theta) \;=\; \exists \alpha, \tau. \alpha \mapsto \tau \in \theta \wedge \nexists \beta. \tau \equiv \beta
\end{array}$$

$$\begin{array}{l}
\mathit{typeGeneralise} :: \mathit{State} \to \mathit{State} \to \mathit{Maybe}\ (\mathit{Subst}, \mathit{State}, \mathit{Subst}) \\
\mathit{typeGeneralise}(\mathcal{S}_0, \mathcal{S}_1) \;=\; (\theta_0, \mathcal{S}, \theta_1) \\
\quad \textbf{where} \;\; (\langle \epsilon \,|\, \theta_0 \,|\, \epsilon \rangle, \mathcal{S}, \langle \epsilon \,|\, \theta_1 \,|\, \epsilon \rangle) \;=\; \mathit{msg}(\mathcal{S}_0, \mathcal{S}_1)
\end{array}$$

The function *typeGeneralise* implements a restricted *msg* which only generalises type information, so that:

$$\begin{array}{l}
\mathit{typeGeneralise}\ \mathcal{S}_0\ \mathcal{S}_1 \;=\; \mathit{Just}\ (\theta_0, \mathcal{S}, \theta_1) \\
\quad \Longrightarrow \;\; \forall.(\mathit{rebuild}\ \mathcal{S})\theta_j \underset{\sim}{\lhd\!\rhd} \mathcal{S}_j
\end{array}$$

Note that just like *match*, *typeGeneralise* will sometimes throw away potential positive information by returning substitutions that are non-injective on term variables. For example, *typeGeneralise* will succeed given the two states $f\ \mathit{Bool}\ c\ c$ and $f\ \alpha\ d\ e$. For more discussion on this point, see Section 5.2.1.

It might appear that our new *memo* does twice as much work as the old one, since it has to test each previous promise not only using *match* but also *typeGeneralise*. However, both *match* and *typeGeneralise* are implemented in terms of *msg*, which does the heavy lifting. It is a straightforward matter to share the work of the *msg* between the two calls, so the additional cost of type generalisation checks should be low.

Example We can demonstrate how this alternative *memo* function works by using the *length* example above. Supercompiling the initial term, *split* will eventually drive the following three subcomponents:

$$\begin{array}{lcl}
\mathcal{S}_0 & = & \langle \mathit{length} \mapsto \ldots \,|\, \mathit{length}\ \mathit{Bool}\ \mathit{xs} \,|\, \epsilon \rangle \\
\mathcal{S}_1 & = & \langle \mathit{length} \mapsto \ldots \,|\, \mathit{length}\ \mathit{Int}\ \mathit{xs} \,|\, \epsilon \rangle \\
\mathcal{S}_2 & = & \langle \mathit{length} \mapsto \ldots \,|\, \mathit{length}\ \mathit{Double}\ \mathit{xs} \,|\, \epsilon \rangle
\end{array}$$

Supercompilation of $\mathcal{S}_0$ proceeds normally, generating a copy of *length* specialised on the *Bool* type argument. However, when the supercompiler comes to supercompile $\mathcal{S}_1$, *typeGeneralise* detects that $\mathcal{S}_0$ and $\mathcal{S}_1$ gives rise to a generalised state:

$$\mathcal{S}_4 \;=\; \langle \mathit{length} \mapsto \ldots \,|\, \mathit{length}\ \alpha\ \mathit{xs} \,|\, \epsilon \rangle$$

where $\theta_0 = \{\alpha \mapsto \mathit{Int}\}$. As a result, *memo* does not create a promise for $\mathcal{S}_1$ and instead recursively supercompiles $\mathcal{S}_4$. Supercompilation of $\mathcal{S}_4$ proceeds normally, generating a copy of *length* unspecialised on its type argument. Finally, the supercompiler comes to $\mathcal{S}_2$. Because $\mathcal{S}_2$ is a simple type instance of $\mathcal{S}_4$, *match* detects that a tie back is possible and supercompilation terminates, producing the final program:

```
let h0 (xs :: [Bool]) = case xs of [ ]      → Z
                                   (y : ys) → S (h0 ys)
```

```
    h1 (α :: ∗) (xs :: [α]) = case xs of [ ]      → Z
                                         (y : ys) → S (h1 α ys)
    f xs = h0 xs
    g xs = h1 Int xs
    h xs = h1 Double xs
in (f, g, h)
```

Compared to the version of the output generated without type generalisation, there is one fewer specialisation of *length*.

Type-generalising the older state In the above example, it might seem odd that the copy of *length* specialised for *Bool* (*h0*) is still present in the output even though we have implemented type generalisation. To fix this issue, it is possible to extend our new *memo* so that when we use the old promise for a *fulfilled* *h*-function to detect that we can generalise the current state, we overwrite the fulfilment for that old *h*-function with some code that just makes an appropriately instantiated call to the type-generalised *h*-function we are about to create. So in the example above, at the time we detect the type-generalisation opportunity (when driving *length Int*), we would overwrite the fulfilment for *h0* with a call to *h1 Bool*.

With this change our *length* example would produce the following supercompiled code:

```
let h0 (xs :: [Bool]) = h1 Bool xs
    h1 (α :: ∗) (xs :: [α]) = case xs of [ ]      → Z
                                         (y : ys) → S (h1 α ys)
    f xs = h0 xs
    g xs = h1 Int xs
    h xs = h1 Double xs
in (f, g, h)
```

It is possible to apply this same idea to the case where the old promise is *unfulfilled*, but in this case we may end up *bind*ing the same *h*-function more than once, and so we have to modify *bind* so that the first call "wins".

Both of these changes will reduce output code size (after dropping dead *h*-functions), but they will not reduce supercompiler runtime.

A problem with these two changes arises if the supercompiler uses *sc*-rollback. The reason is that the techniques above might create a fulfilment which references a promise (for the type-generalised state) which is later rolled back, leaving the corresponding *h*-function unbound in the output program. For this reason, our implementation does not overwrite old fulfilments when *sc*-rollback is turned on.

Can there be more than one "best" type generalisation? One possible concern about the implementation of *memo* above is that the algorithm always chooses the first type generalisation that it discovers. It is not immediately clear that this always leads to us generalising away as much type information as possible.

One potentially worrying scenario is that we could have promises for the states $\mathcal{S}_0 = f\ Int\ Char$ and $\mathcal{S}_1 = f\ Char\ Bool$, and then come to drive $\mathcal{S}_2 = f\ Int\ Bool$. If we type-generalise $\mathcal{S}_2$ against $\mathcal{S}_0$ and $\mathcal{S}_1$ we find the generalised states $f\ Int\ \alpha$ and $f\ \alpha\ Bool$ respectively, but it is clear that if $\mathcal{S}_0$ and $\mathcal{S}_1$ are type correct, then the more-general state $f\ \alpha\ \beta$ will also be type correct, and we should prefer driving that to either of the two instantiated versions.

In fact, this $\mathcal{S}_0/\mathcal{S}_1$ scenario cannot occur because *memo does not record a promise when it detects a type generalisation*, so that environment of promises could not exist. Instead, when *memo* is asked to drive $\mathcal{S}_1$ it would have detected a type generalisation using the previous state $\mathcal{S}_0$, and therefore only recorded a promise for the generalised state $f\ \alpha\ \beta$.

The following theorem about type generalisation can be used to show this fact:

Theorem 6.2.1. *If there exist:*

- *State $\mathcal{S}_{ca}$, and non-coupling type substitutions θ_{ca0} and θ_{ca1} such that $\mathcal{S}_{ca}\theta_{ca0} = \mathcal{S}_c$ and $\mathcal{S}_{ca}\theta_{ca1} = \mathcal{S}_a$*
- *State $\mathcal{S}_{cb}$, and non-coupling type substitutions θ_{cb0} and θ_{cb1} such that $\mathcal{S}_{cb}\theta_{cb0} = \mathcal{S}_c$ and $\mathcal{S}_{cb}\theta_{cb1} = \mathcal{S}_b$*

Then there exists $\mathcal{S}_{ab}$ and non-coupling type substitutions θ_{ab0} and θ_{ab1} such that $\mathcal{S}_{ab}\theta_{ab0} = \mathcal{S}_a$ and $\mathcal{S}_{ab}\theta_{ab1} = \mathcal{S}_b$

Proof. We are restricting attention to type substitutions, so the non-type structure of the states must match exactly and we can concentrate on corresponding pairs of type τ_{ca} and τ_{cb} in the respective states. The proof follows inductively on the structure of τ_{ca} and τ_{cb} in a similar fashion to Lemma B.0.1. □

From this, we can see that if there are two earlier promises $\mathcal{S}_a$ and $\mathcal{S}_b$ against which we can type-generalise the current state $\mathcal{S}_c$, it must be the case that a type-generalisation existed between $\mathcal{S}_a$ and $\mathcal{S}_b$ themselves. If such a type-generalisation exists then the right-hand substitution must either be trivial or non-trivial. Consider each case separately:

- If the right-hand substitution was non-trivial (as it was in the earlier example, with $\mathcal{S}_a = \mathcal{S}_0$, $\mathcal{S}_b = \mathcal{S}_1$ and $\mathcal{S}_c = \mathcal{S}_2$) then we would never have recorded a promise for the later of the two states $\mathcal{S}_a$ and $\mathcal{S}_b$, contradicting the premise that we had earlier promises for both of them.
- If the right-hand substitution was trivial, then promises for both $\mathcal{S}_a$ and $\mathcal{S}_b$ *would* have been recorded by *memo*.

  An example of how this case can occur is if we have earlier promises for $\mathcal{S}_3 = f\ Int\ Char$ and $\mathcal{S}_4 = f\ Int\ \alpha$. If we were to type-generalise a later state $\mathcal{S}_5 = f\ Bool\ Char$ against $\mathcal{S}_3$ we would obtain the generalised state $f\ \alpha\ Char$, but if generalising $\mathcal{S}_5$ against $\mathcal{S}_4$ we would derive the generalised state $f\ \alpha\ \beta$. Ideally, we would like that the more-general $f\ \alpha\ \beta$ state is the one that is chosen as the type-generalised one by *memo*, rather than the other option $f\ \alpha\ Char$ (of course, if we *do* choose $f\ \alpha\ Char$ it will be type-generalised against $\mathcal{S}_3$ in the recursive call of *memo opt*, but it is more efficient to make the more-general choice up front).

  If the right-hand substition was trivial, then the newer promise $\mathcal{S}_b$ ($\mathcal{S}_4$ in our example) will always be at least as general as the older promise $\mathcal{S}_a$ ($\mathcal{S}_3$ in our example), so as long as *memo* always prefers to type-generalise against *more recent* promises then it will always get the maximum possible amount of type generalisation. This can be achieved by arranging that *promises* returns a list of promises with the most recent promises *earlier in the list.*

With this constraint on *promises*, from these two cases we can see that *memo* always performs the maximum amount of generalisation that is justified by the set of promises at the time *memo* decides to type-generalise.

Instance matching on types Above, we defined a new *memo* function that attempts to eagerly throw away type information. However, notice that the code for detecting tieback opportunities closely follows that in Section 3.4, with no special provisions for detecting type instance matches. This reflects the fact that the version of the supercompiler we defined in Chapter 3 was already implicitly performing type instance matches if they were possible. Of course, with the modifications made to *memo* in this section, opportunities for type instance matches will tend to arise much more often.

6.2.7 Termination of generalisation

We cannot take the termination argument we made in Section 3.7 and apply it unmodified to a supercompiler that uses the generalisation methods described above. The principal problem is our previous argument assumed that the supercompilation function $sc = memo\ sc'$ would always recurse via a call to *split* if the termination test failed, but a supercompiler which does generalisation will not in general do so: sometimes it will recurse via *split*, but other times it may directly invoke *sc* on a new, generalised state.

However, with a slight modification to the argument it is possible to show that the supercompiler still terminates by exploiting the fact that any recursive calls that could not have originated from *split* will always recursively supercompile a state which is a generalisation of the current one, in the sense that it will be equal to the current state in some context and renaming.

For clarity, we ignore rollback in this proof and focus on the issues raised by generalisation, though we believe the result extends to the supercompiler with rollback.

Theorem 6.2.2 (Generalisation well-foundedness)**.** *sc recurses a finite number of times.*

Proof. Proceed by contradiction. If *sc* recursed an infinite number of times, then by definition the call stack would contain infinitely many activations of $sc\ hist\ \mathcal{S}$ for (possibly repeating) sequences of *hist* and $\mathcal{S}$ values. Denote the infinite chains formed by those values as $\langle hist_0, hist_1, \ldots\rangle$ and $\langle \mathcal{S}_0, \mathcal{S}_1, \ldots\rangle$ respectively.

Now, observe that it must be the case that there are infinitely many i for which the predicate $isContinue\ (terminate\ hist_i\ \mathcal{S}_i)$ holds. This follows because the only other possibility is that there must exist some j such that $\forall l. l \geq j \implies isStop\ (terminate\ hist_l\ \mathcal{S}_l)$. On such an *isStop* suffix with $\mathcal{S}_l \equiv \langle H_l \,|\, e_l \,|\, K_l\rangle$, then for all $l \geq j$:

- Either this *sc* activation is recursing through a standard call to *split*, or through a recursive call by *tryTags* or *tryMSG* which is recursively supercompiling one of the states residualised *around* the main generalised state in the focus. In this case, $\mathcal{S}_{l+1}$ will be equal to one of the states that would be recursively supplied to *opt* by a call $split\ opt\ \langle H'_l \,|\, e'_l \,|\, K'_l\rangle$ where:

  - $H'_l \subseteq H_l$
  - K'_l `isInfixOf` K_l
  - $e'_l \in subterms(\mathcal{S}_l)$

  We call such a recursive call a *split*-recursion.

- Or else we are recursing via the type generalisation in *memo* or as the term in the focus that is recursively driven by *tryTags*/*tryMSG*. In this case, $\mathcal{S}_l$ is a *instance of* the $\mathcal{S}_{l+1}$ i.e. $\exists H, \theta, K. \langle H, H_{l+1}\theta \,|\, e_{l+1}\theta \,|\, K_{l+1}\theta, K\rangle \equiv \mathcal{S}_l$.

  We call such a recursive call a generalisation-recursion.

By the modified property of *split* (defined in Section 3.7.3) and the properties of *alt-heap* and *subterms* we also have that

$$\begin{aligned}
&\forall l. l \geq j \implies \exists \theta. \\
&\quad H_l \theta \subseteq H_j \cup \mathit{alt\text{-}heap}(e_j, K_j) \\
&\wedge \quad K_l \theta \text{ `}\mathit{isInfixOf}\text{` } K_j \\
&\wedge \quad e_l \theta \in \mathit{subterms}(\mathcal{S}_j)
\end{aligned}$$

Taking these two facts together, we can therefore conclude that the infinite suffix must repeat itself at some point (up to renaming of the free variables): $\exists l_0, l_1, \theta. l_0 \geq j \wedge l_1 > l_0 \wedge \mathcal{S}_{l_0}\theta = \mathcal{S}_{l_1}$. However, we required that *match* always succeeds when matching two terms equivalent up to renaming, which means that $\mathit{sc}\ \mathit{hist}_{l_1}\ \mathcal{S}_{l_1}$ would have been tied back by *memo* rather than recursing. This contradicts our assumption that this suffix of *sc* calls is infinite, so it must be the case that there are infinitely many i such that $\mathit{isContinue}\ (\mathit{terminate}\ \mathit{hist}_i\ \mathcal{S}_i)$.

Now, form the infinite chain $\langle \mathcal{S}'_1, \mathcal{S}'_2, \ldots \rangle$ consisting of a restriction of $\mathcal{S}_i$ of to those elements for which $\mathit{isContinue}\ (\mathit{terminate}\ \mathit{hist}_i\ \mathcal{S}_i)$ holds. As in Section 3.7, this contradicts the fact that $\trianglelefteq$ is a well-quasi-order. □

6.2.8 Correctness of generalisation

Once again, we find that the correctness argument of Section 3.8 is insufficient to explain why the supercompiler with generalisation is correct. In this section we argue why the supercompiler is still correct in the presence of generalisation.

The origin of the problem is that the correctness argument relies on the extra ticks added by the *delay* function to "pay for" the calls to the h-functions that are created by *memo*. However, *delay* only adds a single tick of delay to each subterm, and generalisation may cause a chain of several h-functions to be created with the same subterm in the focus, and that single tick will only be enough to pay for the very first h-function. An example would be if the state $\mathcal{S} = \langle x \mapsto \mathit{True} \,|\, \bullet\ x \,|\, f \rangle$ is generalised by MSG to $\mathcal{S}' = \langle \epsilon \,|\, \bullet\ x \,|\, f \rangle$. For this to be justified by the old correctness argument we would need the two states to be related by same correctness property as we assumed for the splitter (Definition 3.8.2), i.e. we would need that:

$$(\mathit{delay}\,\langle x \mapsto \mathit{True} \,|\, \bullet\ x \,|\, f \rangle = \textbf{let}\ x = \mathit{True}\ \textbf{in}\ f\ x) \gtrsim \textbf{let}\ x = \mathit{True}; h\ x = f\ x\ \textbf{in}\ h\ x$$

However, this clearly does not hold since the supercompiled term will always require an extra non-normalising β-reduction to reach a value than the delayed input does.

One way to repair the proof is to observe that if the supercompiler hypothetically did not create promises (and hence h-functions) from calls originating from generalisation-recursion (in the sense of Theorem 6.2.2) then the output would be correct by the same argument as in Section 3.8, since we would only create a h-function when doing *split*-recursion and hence the single tick introduced by *delay* would be sufficient to make the correctness proof go through.

The only danger with such a modified supercompilation algorithm would be that memoising less might make the supercompiler fail to terminate. However, we can see that this will not happen because whenever we recurse via a generalisation-recursion, the old state is not only an instance of the new state, but a *strict* instance of it: i.e. there is some information in the old state that is not in the new state, be that some type information, a stack frame, or a (non-dead) heap binding. This property is ensured by:

- The explicit checks in *tryMSG′* and *tryTags′* for non-trivial generalisations[6].
- The check in *memo* that the substitution returned by *typeGeneralise* is non-trivial according to *substNonTrivial*.

As a result of this strict-instance property, all potential infinite chains of recursion in *sc* must be interspersed by infinitely many *split*-recursions, which are sufficient to show that *sc* eventually ties back.

This shows that a hypothetical supercompiler which did not create promises upon generalisation-recursion is both terminating and correct. To see that the supercompilation algorithm as we have actually presented it is correct, observe that the process of inlining all calls to *h*-functions originating from a generalisation-recursion must terminate, since all recursive loops involving those *h*-functions will eventually be broken by a *h*-function originating from a *split*-recursion. Therefore, inlining all calls to *h*-functions originating from a generalisation-recursion in the output of our supercompilation algorithm terminates, and yields a term equivalent to that returned by the hypothetical supercompiler. This shows that if the hypothetical algorithm is correct (which it is, as we argued above) then our presented algorithm must be.

6.3 Speculative evaluation

A call-by-need supercompiler will unavoidably sometimes find itself discarding information in *split* due to work duplication concerns. An example of this is a program such as:

```
let odd n = case n of Z   → False
                      S m → even m
    even n = case n of Z   → True
                       S m → odd m
    b = odd unk
in (if b then x else y, if b then y else x)
```

If we were free to push the *b* binding into each component of the pair, we could fuse the consumption of the *Bool* result by the **if** expressions into *odd*, hence deforesting the intermediate Boolean values:

```
let h0 n a b = case n of Z   → b
                         S m → h1 m a b
    h1 n a b = case n of Z   → a
                         S m → h0 m a b
in (h0 unk x y, h0 unk y x)
```

However, if both components of the pair are forced then the entire structure of *unk* will be deconstructed by repeated **case** scrutinisation twice, as opposed to once in the original term. Work is duplicated by pushing the *b* binding into each component of the pair. To prevent such problems, we saw in Chapter 4 how *split* forces residualisation of any heap bindings that are not syntactic values, if pushing them down into recursively supercompiled states would potentially duplicate work.

[6] For these checks to be sufficient we need that the states we attempt to generalise don't contain dead heap bindings, a property which is easy to establish in the implementation by inserting calls to a function *gc* :: *State* → *State* that discards such bindings.

However, the check for something being a syntactic value is too strict—many things are in fact safe to duplicate even though they are not manifestly values. A simple example is a partial application:

```
let (∧) = λx y. case y of True → x; False → False
    g = (∧) True
in (g False, g True)
```

As described, our supercompiler would produce the following output for the program:

```
let g = λy. case y of True → x; False → False
in (g False, g True)
```

The (∧) call in the *g* binding has been inlined, but the two calls to *g* have not been. This is because (∧) *True* is an application, not a value, so the *g* binding appears to be expensive and hence is not pushed down into the components of the pair by *split*. This prevents the supercompiler from eliminating the **case**.

Of course, after supercompiling *g* we discover that it actually evaluates to a value—but we discover that fact too late to make use of it. Another manifestation of this problem is:

```
let isJust    mb = case mb of Nothing → False; Just _ → True
    isNothing mb = case mb of Nothing → True;  Just _ → False
    x = let y = True
        in Just y
in (isJust x, isNothing x)
```

The *x* binding is not a manifest value since in actually consists of a **let**-binding wrapped around a value. As a result, it will not pushed down by *split*, the supercompiler will fail to eliminate the **case**s, and we will get this output:

```
let h0 mb = case mb of Nothing → False; Just _ → True
    h1 mb = case mb of Nothing → True;  Just _ → False
    x = let y = True in Just y
in (h0 x, h1 x)
```

As a final example of how work duplication checks can prevent optimisation, consider the supercompilation of *iterate not True*, which reduces to the following state:

```
let not = ...
    iterate f x = let y = f x
                      ys = iterate f y
                  in y : ys
    x = True
    y0 = not x
    ys0 = iterate not y0
in x : ys0
```

This state will *split* and *reduce* to:

```
let not = ...
    iterate f x = ...
    x = True
    y0 = not x
```

```
      y1 = not y0
      ys1 = iterate not y1
  in y0 : ys1
```

If we were to *split* once again *y0* would be residualised because it is used by both *not y0* and in the actual cons-cell in the focus of the state. This impedes optimisation and generates a residual program like:

```
let h0 = let x = True in x : h1
    h1 = let y0 = False in y0 : h2 y0
    h2 y0 = let y1 = not y0 in y1 : h2 y1
in h0
```

This program is suboptimal because it contains residual case-scrutinisations (in *not*) which we could have eliminated statically. Ideally, our initial *split* and *reduce* would have reduced the *y0* heap binding as well, to produce:

```
let not = ...
    iterate f x = let y = f x
                      ys = iterate f y
                  in y : ys
    x = True
    y0 = False
    ys0 = iterate not y0
in x : ys0
```

With *y0* a manifest value, there are no work duplication problems and we can supercompile the input into an optimal loop:

```
let h0 = let x = True in x : h1
    h1 = let y0 = False in y0 : h0
in h0
```

Our solution for these problems is to use *let-speculation* to discover those heap bindings that are “morally” values. To implement this, all calls of *reduce* within the main *sc′* function are replaced with a call *speculate.reduce*, where *speculate* reduces any non-values in the resulting *Heap* to values by invoking *reduce* on them. One way to implement a suitable *speculate* would be:

```
speculate :: State → State
speculate ⟨H | e | K⟩ = ⟨speculateHeap H | e | K⟩

speculateHeap :: Heap → Heap
speculateHeap H = speculateMany (bvsHeap H) H
  where
    speculateMany xs H = foldl speculateOne H xs

    speculateOne :: Heap → Var → Heap
    speculateOne H x
      | H ≡ H′, x:τ ↦ e
      , ⟨H″ | e′ | ϵ⟩ ← reduce (normalise ⟨H′ | e | ϵ⟩)
      = H″, x:τ ↦ e′
      | otherwise
      = H
```

Where *bvsHeap* :: *Heap* → [*Var*] is defined by $bvsHeap(H) = \{x \mid x : \tau \mapsto e \in H\}$. Remember that because we work with normalised states, if *reduce* returns a state with an empty stack then the focus must either be a value or a variable (which will either unbound in the corresponding heap or be bound to a value in that same heap). Both variables and values are *cheap* and hence will be propagated without fear of work duplication by algorithms like *split* and *msg*, and so in these cases we use the reduced version as the new definition for the heap binding being speculated.

Cheap expressions that are "close to" values (like $(\wedge)$ *True* or **let** y = *True* **in** *Just* y in our example) will be replaced with syntactic values by this speculator, which allows *split* to residualise less and propagate more information downwards. The use of this technique means that unlike Mitchell [2010], we do not require a separate arity analysis stage to supercompile partial applications well. Furthermore, this technique automatically achieves useful **let**-floating, and also allows fully-applied function calls and primop applications to be duplicated if those calls quickly reach values[7].

6.3.1 Recursive speculation

Using *reduce* on some heap binding may give rise to further heap bindings requiring speculation. For example, if we speculate the heap binding $y \mapsto$ **let** x = *not True* **in** *Just* x, after reduction the outgoing heap will contain a binding for the term *not True*.

Naturally, we wish to speculate such bindings recursively, but implemented naively this creates a new source of non-termination—for example, consider speculating a heap binding for *enumFrom Z*, denoting an infinite list of Peano numbers. After a call to *reduce*, we will have the state $\langle y0 \mapsto S\ Z, ys0 \mapsto enumFrom\ y0 \mid Z : ys0 \mid \epsilon \rangle$. If we recursively speculate the *ys0* binding in turn we generate the new bindings $y1 \mapsto S\ y0$ and $ys1 \mapsto enumFrom\ y1$, and if we recursively speculate *ys1* it is clear that we will be in a loop.

Furthermore, recursive speculation can be another source of exponential explosion in supercompilation. Consider this program:

```
let f1 x = f2 y ++ f2 (y + 1)
      where y = (x + 1) * 2
    f2 x = f3 y ++ f3 (y + 1)
      where y = (x + 1) * 2
    f3 x = f4 y ++ f4 (y + 1)
      where y = (x + 1) * 2
    f4 x = [x + 1]
    shared = f1 0
in (sum shared, length shared)
```

Reduction of this term only serves to place the let-bound terms *shared* and *f1* to *f4* in the heap. Once there, they will be speculated. The fn functions are already values, so this has no effect. However, speculation of *shared* gives rise to three new heap bindings:

$$y_0 \mapsto (0+1) * 2 \qquad f2_0 \mapsto f2\ y_0 \qquad f2_1 \mapsto f2\ (y+1)$$

Speculating *either* the $f2_0$ or $f2_1$ bindings will give rise to three new bindings:

$$y_1 \mapsto (y_0+1) * 2 \qquad f3_0 \mapsto f3\ y_1 \qquad f3_1 \mapsto f3\ (y_1+1)$$

[7] Speculation of saturated applications is a particularly important feature if (like CHSC) your supercompiler introduces data constructor wrappers to deal with partial application of constructors in the input language.

Overall we end up speculating 1 application of *f1*, 2 applications of *f2*, 4 applications of *f3*, and 2^{n-1} applications of fn.

We solve the problem of recursive speculation non-termination in the same way as we solve the problem of supercompiler termination: by using a well-quasi-order based termination test. However, unlike our use of a termination test in the standard supercompiler, we will *thread* the *History* through our speculator (like state) rather than passing it strictly down into recursive calls (like an environment). The reason for this is that this will help to prevent exponential explosion in the speculator without losing most "obvious" speculation improvements such as reduction of partial applications and **let**-floating.

Additionally, we will make use of a speculator which is *idempotent*. The reason that we desire this property is that we will *speculate* the entire available heap upon every non-*Stop*ping invocation of sc'. Furthermore, the heap passed to a recursive invocation of sc' is usually substantially similar to the heap of the parent call. Therefore, in a nest of recursive sc' invocations we will end up driving substantially similar heaps multiple times, and it would be strange if *speculate* made progress on evaluating the heap bindings with every successive invocation. An example of the behaviour we would like to avoid is that given the input state:

$$\mathcal{S}_0 = \left\langle \mathit{repeat} \mapsto \ldots, x \mapsto \mathit{repeat}\ y \middle| a \middle| \begin{array}{l}\textbf{case} \bullet \textbf{of } \mathit{True} \to b \\ \qquad\qquad \mathit{False} \to c\end{array}, \begin{array}{l}\textbf{case} \bullet \textbf{of } \mathit{True} \to f\ x \\ \qquad\qquad \mathit{False} \to g\ x\end{array} \right\rangle$$

We might *speculate* it to:

$$\mathcal{S}_0' = \left\langle \mathit{repeat} \mapsto \ldots, x \mapsto y : x_0, x_0 \mapsto \mathit{repeat}\ y \middle| a \middle| \begin{array}{l}\textbf{case} \bullet \textbf{of } \mathit{True} \to b \\ \qquad\qquad \mathit{False} \to c\end{array}, \begin{array}{l}\textbf{case} \bullet \textbf{of } \mathit{True} \to f\ x \\ \qquad\qquad \mathit{False} \to g\ x\end{array} \right\rangle$$

And then when we *split* to drive the subcomponent state $\mathcal{S}_1$:

$$\mathcal{S}_1 = \left\langle \mathit{repeat} \mapsto \ldots, x \mapsto y : x_0, x_0 \mapsto \mathit{repeat}\ y \middle| b \middle| \begin{array}{l}\textbf{case} \bullet \textbf{of } \mathit{True} \to f\ x \\ \qquad\qquad \mathit{False} \to g\ x\end{array} \right\rangle$$

We could *speculate* in a similar manner to obtain:

$$\mathcal{S}_1' = \left\langle \mathit{repeat} \mapsto \ldots, x \mapsto y : x_0, x_0 \mapsto y : x_1, x_1 \mapsto \mathit{repeat}\ y \middle| b \middle| \begin{array}{l}\textbf{case} \bullet \textbf{of } \mathit{True} \to f\ x \\ \qquad\qquad \mathit{False} \to g\ x\end{array} \right\rangle$$

Notice that the x list is becoming longer with every invocation of sc'. If we are careful to make our speculator idempotent, we will tend to avoid this problem.

In order to achieve idempotence, it is convenient to make use of rollback in a similar way to Section 6.1. For this purpose we make use of a simple continuation-passing monad *SpecM* supporting the following operations:

$$\begin{array}{l}
\mathit{runSpecM} :: \mathit{SpecM}\ \mathit{Heap} \to \mathit{Heap} \\
\mathit{catchSpecM} :: ((\forall b. c \to \mathit{SpecM}\ b) \to \mathit{SpecM}\ a) \\
\qquad\qquad\quad \to (c \to \mathit{SpecM}\ a) \to \mathit{SpecM}\ a \\
\mathit{callCC} :: ((\forall b. a \to \mathit{SpecM}\ b) \to \mathit{SpecM}\ a) \to \mathit{SpecM}\ a \\
\mathit{callCC}\ f = \mathit{catchSpecM}\ f\ \mathit{return}
\end{array}$$

We also need some straightforward operations on *Heap*s:

$$\begin{array}{ll}
 & \mathit{insertHeap} :: \mathit{Var} \to \mathit{Term} \to \mathit{Heap} \to \mathit{Heap} \\
(\cup_H) :: \mathit{Heap} \to \mathit{Heap} \to \mathit{Heap} & \mathit{heapFromList} :: [(\mathit{Var}, \mathit{Term})] \to \mathit{Heap} \\
(\setminus_H) :: \mathit{Heap} \to \mathit{Heap} \to \mathit{Heap} & \mathit{singletonHeap} :: \mathit{Var} \to \mathit{Term} \to \mathit{Heap}
\end{array}$$

We will also use a *topologicalSort* function to speculate the heap in dependency order, as far as that is possible. The list returned by *topologicalSort* is ordered so that strongly connected components (SCCs) earlier in the list do not make reference to any heap-bound variable bound by a later SCC.

```
topologicalSort :: Heap → [HeapSCC]
data HeapSCC = AcyclicSCC (Var, Term) | CyclicSCC [(Var, Term)]
```

The last piece we need is a function similar to the standard termination test *terminate*. We do not want to make use of the standard tag-bag termination test if we are threading the history rather than passing it down as an environment, because the fact that $\mathcal{S}_1 \trianglelefteq \mathcal{S}_2 \not\Longrightarrow \mathcal{S}_2 \trianglelefteq \mathcal{S}_1$ means that our speculator could be sensitive to the order in which unrelated recursive speculations are made. We do not want it to be the case that if we have the state $\langle x \mapsto e_0, y \mapsto e_1 \,|\, (x, y) \,|\, \epsilon \rangle$ we might succeed in reducing both e_0 and e_1 to a value if we speculate x and then y but not if we were to speculate y and then x.

The *terminateSet* function therefore defines a termination test like *terminate*, but the test is implemented by the symmetric WQO $\mathcal{S}_0 \trianglelefteq_{set} \mathcal{S}_1 \iff dom(tagBag(\mathcal{S}_0)) = dom(tagBag(\mathcal{S}_1))$ rather than the usual tag-bag WQO. Like in Section 6.1, we allow *History* to contain some "extra information".

```
terminateSet :: History a → State → a → TermRes a
```

Now we have all the pieces in place, we can define a suitable idempotent heap speculator (Figure 6.1).

The overall plan of *speculateHeap* is to run through the input heap bindings in topological order, and have *speculateHB* reduce each binding in an accumulating heap of previously-speculated bindings. Any new heap bindings arising from reducing a given heap binding are recursively speculated by *speculateSCC*, giving rise to a logical tree of calls to *speculateHB*. The path from the root of the tree to the current leaf is recorded in the *Path* that is passed throughout the speculator.

Generally, the heap of previously-speculated bindings H_{env} passed to *speculateHB* will contain all the information we have about the free variables of the heap binding to be speculated. However, the situation is complicated somewhat by cyclic SCCs where we cannot make such a guarantee without implementing a fixed point, and where we do not want the amount of reduction the speculator can achieve to depend on the arbitrary order in which the bindings occur in the *CyclicSCC* list. To avoid these problems, we reduce each element of a cycle in an environmental heap which does not contain bindings for any of the other bindings in the cycle.

We rollback (via *catchSpecM*) to the heap binding which is the common ancestor (in the *speculateHB* tree) of the current heap binding and the one which triggered the termination test. This is necessary for idempotence, so that if we speculate this heap:

```
iterate f x = ...
not x = ...
y = True
ys = iterate not y
```

We detect the non-termination that arises from speculating a call to *iterate* (which produces an infinite list, and so cannot be fully speculated) and roll back to restore the original binding *ys* = *iterate not y* instead of producing a speculated heap such as:

```
iterate f x = ...
not x = ...
```

```
y = True
ys = y : ys0
y0 = False
```

```
type Path = [(Var, SpecHistory → SpecM (SpecHistory, Heap))]
speculateHeap :: Heap → Heap
speculateHeap H_top = foldl′ speculateTopSCC ϵ (topologicalSort H_top)
  where
    speculateTopSCC :: Heap → HeapSCC → Heap
    speculateTopSCC H′ scc
      = runSpecM (fmap snd (speculateSCC (return ()) [ ] ϵ (emptyHistory, H′) scc))
    speculateSCC :: SpecM () → Path
                 → Heap → (History [Var], Heap) → HeapSCC
                 → SpecM (History [Var], Heap)
    speculateSCC fail_rb path H_env (hist, H′) scc = case scc of
      AcyclicSCC xe → fmap (λ(hist, H′_extra). (hist, H′_extra ∪_H H′))
                           (speculateHB fail_rb path hist (H_env ∪_H H′) xe)
      CyclicSCC xes
        → callCC (λk → let fail′_rb = fail_rb ≫ k (hist, H′ ∪_H heapFromList xes)
                           go (hist, H″) = fmap (λ(hist, H″_extra). (hist, H″_extra ∪_H H″))
                                                .speculateHB fail′_rb path hist (H_env ∪_H H′)
                       in foldM go (hist, H′) xes)
    speculateHB :: SpecM () → Path
                → History [Var] → Heap → (Var, Term)
                → SpecM (History [Var], Heap)
    speculateHB fail_rb path hist H_env (x, e)
      | let S@(_, _, K_norm) = normalise (H_env, e, ϵ)
            (H′, e′, K′) = reduce S
            H_difference = H′ ∖_H H_env
      , K′ ≡ ϵ
      , let recurse path′ hist′ = fmap (λ(hist, h′). (hist, insertHeap x′ e′ h′))
                                       (foldM (speculateSCC fail_rb path′ H_env) (hist′, ϵ)
                                              (topologicalSort H_difference))
      → if K_norm ≡ ϵ
         then recurse path hist
         else catchSpecM (λrb. let path′ = path ++ [(x, rb)]
                               in case terminateSet hist (gc S) (map fst path′) of
                                    Stop path_old → commonAncestorRB path_old path hist
                                    Continue hist′ → recurse path′ hist′)
                         (λhist. fail_rb ≫ return (hist, singletonHeap x e))
      | otherwise
      → fail_rb ≫ return (hist, singletonHeap x e)
    commonAncestorRB :: [Var] → Path → History [Var]
                     → SpecM (History [Var], Heap)
    commonAncestorRB (_ : x_old : path_old) (_ : (x, rb) : path)
      | x_old ≡ x = commonAncestorRB (x_old : path_old) ((x, rb) : path)
    commonAncestorRB _                      ((_, rb): _)
      = rb
```

Figure 6.1: Idempotent heap speculator

```
ys0 = y0 : ys1
ys1 = iterate not y0
```

The history (of type *History* [*Var*]) used by the termination test is threaded throughout the speculator, with the notable feature that top-level heap bindings (i.e. those already present in the output of the *reduce* performed by sc') do *not* thread the history. The reason for this is that two top-level heap bindings do not have a common ancestor in the *speculateHB* tree, and so we cannot sensibly implement rollback for that situation.

There is another form of rollback in the speculator: we may roll back via a use of fail_{rb} if any heap binding which is a child of a *CyclicSCC* fails to reduce to a cheap term. The reason for doing this is that we do not want to reduce just *some* members of the *CyclicSCC*, since that might break the cyclic loop and hence cause the *topologicalSort* of the output heap to be less constrained in a subsequent invocation of *speculateHeap*, which could in turn lead to the H_{env} for a call to *speculateHB* potentially containing more information than it did last time.

A concrete example of what might go wrong is that we could have the cyclic SCC $y = \mathit{Just}\ (\mathit{not}\ x); x = \mathit{const}\ \mathit{True}\ y$ which we partially-speculated to $y = \mathit{Just}\ (\mathit{not}\ x); x = \mathit{True}$. A subsequent *speculateHeap* would identify x and y as non-cyclic and so we would be able to speculate the y binding to *Just False*, which would violate idempotency.

The function $\mathit{gc} :: \mathit{State} \to \mathit{State}$ is a "garbage collector" for states that drops any heap bindings that are syntactically dead. This is used because we anticipate that any individual heap binding we speculate will most likely only use a fraction of the rest of the heap, and by reducing the size of the heap in a *State* we reduce the size of the corresponding tag-bag and hence also reduce the likelihood that the termination test will fail. Furthermore, we do not test the termination test if the heap binding being speculated was cheap *before* reduction, which prevents spurious rollback if we happen to e.g. speculate two distinct bindings $x = \mathit{True}^{t_0}$ and $y = \mathit{True}^{t_0}$ which are tagged the same. This does not compromise termination of the speculator.

Theorem 6.3.1. *The speculateHeap function is idempotent.*

Proof. Each heap binding in the output of *speculateHeap* obeys at least one of:

1. The heap binding binds a cheap term.

2. The heap binding could not be reduced to a cheap term by *reduce* (either because there wasn't enough information available to do so, or because the termination test in *reduce* failed before this could happen).

3. The heap binding was part of a *CyclicSCC* that was rolled back to via fail_{rb}.

4. The heap binding was successfully reduced to a cheap term by *reduce*, but had its speculation rolled back to by *commonAncestorRB*.

We consider what a subsequent call to *speculateHeap* will do to each kind of binding one at a time.

Case 1 If the speculated heap binding is cheap (i.e. a variable or value), a subsequent *speculateHeap* will clearly leave it is a cheap term, so idempotence is straightforward.

Case 2 If *reduce* failed to reduce the heap binding to a cheap term in a previous *speculateHeap* call, it will still fail to do so in a later call, and so again idempotence follows. The complication with this case comes with the fact that the *reduce* is done in the environment of previously-speculated heap bindings H_{env}. In order for our argument to hold, we need that the H_{env} that reaches the binding in the subsequent *speculateHeap* is the same as the H_{env} in the earlier call (at least, it must be the same for those bound variables reachable from the binding itself—we don't mind if it contains a different set of unreachable bindings).

The reason that this is true is that we can assume that all heap bindings that were in the original H_{env} will be treated idempotently by a subsequent *speculateHeap* call, so the actual right-hand-side of all bindings will be unchanged. Furthermore, the actual reachable variables bound in the H_{env} will be unchanged from last time thanks to the fact that our topological-order traversal means that we add exactly those bindings which are reachable to the H_{env} before we speculate a binding (except for *CyclicSCC*s, which won't be modified by speculation except they are driven completely to cheap terms, in which situation this case wouldn't apply).

Case 3 Because we can assume that the environment of heap bindings that reaches the *speculateSCC* of a *CyclicSCC* will be unchanged between two consecutive *speculateHeap* calls (by the same argument as the previous case), the reduction of the *CyclicSCC* to a value must fail in exactly the same way in both calls, rolling back to place the *CyclicSCC* nearest to the root of the *speculateHB* tree unmodified in the output heap.

Case 4 In this case, because the heap binding is present in the output of *speculateSCC* it must mean that two of its children heap bindings in the *speculateHB* tree were embedded into each other, and this heap binding was the common ancestor. Therefore, because the (reachable) H_{env} is exactly the same as last time (by the same argument as in the last two cases), when we speculate this heap binding we must end up speculating exactly the same child bindings and hence causing exactly the same rollback.

□

Performance impact of speculation As described above, we perform a full run of speculation using *speculateHeap* upon every invocation of *sc*. In practice, this would be rather expensive. To avoid paying the full cost of speculation, our implementation records which heap bindings have already been speculated, so that *speculateHeap* can skip speculating any binding which has already been thus marked. This is sufficient to reduce the performance cost of using speculation to only approximately 4% of supercompiler runtime (Section 7.7).

Chapter 7

Experiments

In this chapter we evaluate our supercompiler implementation using a number of benchmarks. The benchmarks are drawn from several sources:

1. Firstly, we use all of the benchmarks from the "imaginary" section of the Nofib benchmark suite [Partain, 1993]. These small benchmarks were written without supercompilation or deforestation in mind, so they allow us to test whether supercompilation will have a positive effect on programs that do not have obvious deforestation opportunities.

   We do not attempt to supercompile the full Nofib suite because the other Nofib benchmarks are considerably more complicated and generally suffer from extremely long supercompilation times.

2. Secondly, we use all the additional (non-Nofib) benchmarks described in Jonsson [2011]. These benchmarks all have obvious deforestation opportunities, so we use them to check whether our supercompiler is able to exploit such opportunities.

3. Lastly, we use a number of original microbenchmarks designed to test various deforestation and specialisation opportunities:

   - *Accumulator* tests deforestation of the composition $foldl\ c\ n\ (enumFromTo\ a\ b)$, which requires the supercompiler to generalise an accumulating parameter.
   - *Ackermann* tests specialising an Ackermann function with first argument 2.
   - $AckermannPeano-1$ tests specialising an Ackermann function for Peano numbers with first argument 1.
   - $AckermannPeano-2$ tests specialising an Ackermann function for Peano numbers with first argument 2.
   - *EvenDouble* tests deforestation of the composition $even\ (double\ x)$ for Peano number x, where *double* is implemented with an accumulating argument.
   - *EvenDoubleGenerator* tests deforestation of the composition $even\ (double\ x)$ for Peano number x, where *double* is implemented without an accumulator.
   - *KMP* tests specialisation of a string matcher over the alphabet $\{A, B\}$ looking for the fixed pattern AAB.
   - *LetRec* demonstrates deforestation in programs using recursive-**let** by calculating **let** $ones = 1{:}ones$ **in** $map\ (\lambda x \to x{+}1)\ ones$ and $map\ (\lambda x \to x{+}1)\ (repeat\ 1)$.
   - *MapMapFusion* test deforestation of a $length.map\ f.map\ g$ composition.
   - *ReverseReverse* tests supercompilation on the composition $reverse.reverse$.

- *SumSquare* tests deforestation of $foldl' \; (+) \; 0 \; [k*m \mid k \leftarrow enumFromTo \; 1 \; n, m \leftarrow enumFromTo \; 1 \; k]$ (this benchmark originated from the testsuite of Mitchell's supercompiler [Mitchell, 2010]).
- *ZipMaps* demonstrates deforestation of *zip*-like functions by supercompiling $length \,\$\, zip \; (map \; Left \; xs) \; (map \; Right \; xs)$.
- *ZipTreeMaps* demonstrates deforestation of *zip*-like functions on binary trees by supercompiling

  $$sizeT \; (zipT \; (mapT \; Left \; (buildTree \; n \; Empty)) \; (mapT \; Right \; (buildTree \; n \; Empty)))$$

  where *buildTree n* returns a complete binary tree of depth n and *sizeT* returns the number of nodes in the tree.

One of the main unsolved challenges of supercompilation is how to be sufficiently aggressive to garner its benefits, without also risking massive code bloat without any accompanying performance gain (Section 9.3). This challenge is not a theoretical one. In our experiments, supercompilation of some benchmarks (including some very small ones) failed to terminate after several hours, and we resorted to hand-tuning supercompiler flags to incorporate their results. Precisely:

- Supercompilation of the *tak* benchmark did not terminate when positive information propagation was enabled. Therefore, all *tak* benchmark numbers are reported with positive information propagation turned off.
- Supercompilation of *wheel−sieve* benchmarks did not terminate regardless of whether positive information propagation was enabled. These benchmarks are all run using our work-bounding mechanism (Section E.4.2), which limits the total number of β-reductions performed by the supercompiler to at most 10 times the size (in number of AST nodes) of the input program.

We believe that this non-termination arises not because an error in our implementation or proofs means that the supercompiler is diverging, but rather because these programs suffer from the code explosion problems inherent to supercompilation. The evidence we have for this is that the tree of calls to *sc* for these programs is characterised by having very large numbers of states which are superficially similar but on closer inspection differ slightly in binding structure or the precise degree of compile-time information which is available. This pattern is characteristic of programs undergoing code explosion.

The supercompiler implementation incorporates the work of Appendix E, in particular the inlining control mechanisms described in Section E.4. All benchmarks used the $-O2$ flag of GHC to enable all of GHC's optimising transformations, including strictness analysis and GHC's own function specialisation mechanisms such as constructor specialisation [Peyton Jones, 2007]. All benchmarks were carried out on an unloaded machine equipped with two 2.8 GHz Intel Xeon quad cores and 8 GB of RAM.

7.1 Overall results

Headline results for the supercompiler are presented in Figure 7.1. Overall the performance results are good: allocations fell by an average of 34% while runtime fell by 42% on average. Unsurprisingly, the effect of supercompilation tends to be much more dramatically positive for those benchmarks which are specifically designed to have exploitable fusion opportunities, but some of the Nofib benchmarks do achieve good speedups, such

Test	**Cmp.**[a]	**Run**[b]	**Mem.**[c]	**Size**[d]
Accumulator	+51.4%	+34.8%	-23.1%	+22.1%
Ackermann	+316.2%	-26.7%	-52.1%	+36.2%
AckermannPeano-1	+72.5%	-50.0%	-9.1%	+27.9%
AckermannPeano-2	+25525.0%	-24.8%	-20.9%	+180.7%
Append	+10.3%	+6.8%	+2.9%	+12.5%
EvenDouble	+16.2%	+0.6%	-0.0%	+16.2%
EvenDoubleGenerator	+10.8%	-52.9%	-58.3%	+8.3%
Factorial	+61.1%	+0.0%	+0.0%	+16.8%
KMP	+56.8%	-34.0%	+0.0%	+77.5%
LetRec	+125.0%	-95.1%	-100.0%	-46.5%
MapMapFusion	+13.5%	-54.5%	-55.0%	+18.9%
Raytracer	+60.0%	-39.4%	-51.7%	+4.2%
ReverseReverse	+15.0%	+0.0%	-0.0%	+20.2%
SumSquare	+296.8%	-11.7%	+109.0%	+32.3%
SumTree	+1055.0%	-89.4%	-100.0%	+28.1%
TreeFlip	+707.5%	-91.2%	-100.0%	+33.4%
ZipMaps	+44.7%	-69.3%	-71.9%	+3.6%
ZipTreeMaps	+8808.5%	-73.2%	-79.8%	+57.9%
bernouilli	+49961.7%	-3.6%	-8.6%	+167.9%
exp3_8	+26802.3%	-1.7%	-0.0%	+188.9%
gen_regexps	+5969.6%	+0.0%	+0.3%	+4.5%
integrate	+48022.9%	-62.3%	-61.4%	+98.3%
paraffins	+42320.0%	+7.7%	+0.2%	+0.5%
primes	+21556.4%	-14.3%	+15.5%	+100.9%
queens	+35627.5%	+0.0%	+40.9%	+89.0%
rfib	+20245.9%	+0.0%	-0.1%	+13.7%
tak	+29135.1%	+0.0%	+8226.9%	+47.6%
wheel-sieve1	+18395.9%	+0.0%	-0.0%	+35.1%
wheel-sieve2	+15056.6%	-1.9%	-0.8%	+93.0%
x2n1	+132002.0%	+0.0%	-75.1%	-7.3%
Min	+10.3%	-95.1%	-100.0%	-46.5%
Max	+132002.0%	+34.8%	+8226.9%	+188.9%
Geometric Mean	+1865.5%	-42.2%	-33.7%	+37.1%

[a] Compile time change when supercompilation enabled

[b] Program runtime change when supercompilation enabled

[c] Runtime allocation change when supercompilation enabled

[d] Object file size change when supercompilation enabled

Figure 7.1: Comparison of non-supercompiled and supercompiled benchmarks

as *integrate* and *x2n1*, which allocate 61% and 75% less after supercompilation. Code size increases are moderate in most cases, but the compilation time penalty is severe: enabling supercompilation increases compilation time by an average factor of 20 times, with the penalty being much more pronounced on the Nofib benchmarks.

It is important to note that the baseline for these benchmark numbers already includes the effects of GHC's shortcut fusion [Gill et al., 1993] mechanism, which deforests many of the list-producing computations in these benchmarks. If we disable shortcut fusion, we can see exactly how much of an improvement supercompilation is over a compiler that does not implement any deforestation at all: these results are presented in Figure 7.2. The improvement here is much more obvious, with almost every single program allocating considerably less as a result of supercompilation.

Returning to the results of Figure 7.1 which compare the supercompiler with the version of GHC incorporating shortcut fusion, we notice that the *SumSquare*, *primes*, *queens* and *tak* benchmarks all have the amount which they allocate severely negatively impacted by supercompilation. We consider each of these cases and explain the cause.

7.1.1 Allocation increase due to strictness analysis

The supercompiled version of *tak* allocates over 80 times more than the non-supercompiled version. The reason for this is a bad interaction with GHC's strictness analysis optimisation.

The *tak* benchmark essentially consists of the following loop:

$$tak :: Int \rightarrow Int \rightarrow Int \rightarrow Int$$

Test	**Cmp.**[a]	**Run**[b]	**Mem.**[c]	**Size**[d]
Accumulator	+43.2%	+34.0%	-23.1%	-27.6%
Ackermann	+316.2%	-94.5%	-94.8%	-26.7%
AckermannPeano-1	+72.5%	-75.0%	-41.2%	+4.3%
AckermannPeano-2	+24900.0%	-31.7%	-28.5%	+130.7%
Append	+10.3%	-48.1%	-33.3%	-18.5%
EvenDouble	+16.2%	+0.0%	-30.8%	-18.8%
EvenDoubleGenerator	+10.8%	-70.4%	-68.7%	-23.9%
Factorial	+61.1%	-100.0%	-99.9%	-31.9%
KMP	+56.8%	-47.6%	-57.1%	+40.6%
LetRec	+125.0%	-98.0%	-100.0%	-65.4%
MapMapFusion	+13.5%	-82.7%	-73.5%	-21.1%
Raytracer	+60.0%	-71.4%	-64.1%	-44.6%
ReverseReverse	+17.9%	+0.0%	-0.0%	-18.6%
SumSquare	+324.1%	-72.8%	-8.0%	+39.7%
SumTree	+1000.0%	-91.8%	-100.0%	-19.2%
TreeFlip	+707.5%	-92.9%	-100.0%	-18.8%
ZipMaps	+48.6%	-84.5%	-79.5%	-17.5%
ZipTreeMaps	+9002.2%	-78.0%	-84.2%	+14.0%
bernouilli	+51050.0%	-18.5%	-15.6%	+45.3%
exp3_8	+26204.4%	-1.7%	-6.3%	+103.7%
gen_regexps	+6547.6%	+0.0%	-14.5%	-8.9%
integrate	+46098.0%	-90.8%	-84.2%	-15.2%
paraffins	+41796.3%	+6.1%	-1.0%	-34.6%
primes	+22126.3%	-41.2%	-23.5%	+58.3%
queens	+35627.5%	-88.9%	-94.3%	+32.9%
rfib	+20811.1%	-86.7%	-99.9%	-35.4%
tak	+29135.1%	-90.0%	-93.3%	-17.4%
wheel-sieve1	+19602.2%	-62.7%	-97.1%	-9.7%
wheel-sieve2	+16293.9%	-20.8%	-45.7%	+89.9%
x2n1	+134754.2%	-100.0%	-99.4%	-57.5%
Min	+10.3%	-100.0%	-100.0%	-65.4%
Max	+134754.2%	+34.0%	-0.0%	+130.7%
Geometric Mean	+1882.8%	-74.4%	-75.7%	-8.1%

[a] Compile time change when supercompilation enabled
[b] Program runtime change when supercompilation enabled
[c] Runtime allocation change when supercompilation enabled
[d] Object file size change when supercompilation enabled

Figure 7.2: Comparison of non-shortcut-fused and supercompiled benchmarks

```
tak x y z = if not (y < x) then z
                           else tak (tak (x − 1) y z)
                                    (tak (y − 1) z x)
                                    (tak (z − 1) x y)
```

Before supercompilation, it is manifestly obvious to GHC's strictness analyser that this loop is strict in all of its arguments. After supercompilation, the *tak* loop is split into several mutually recursive *h*-functions, each of which are too large for GHC's inlining heuristics to consolidate into a single function. With the core loop broken across several functions like this, GHC's strictness analyser produces worse results. The problem is exemplified by the following code:

```
  -# NOINLINE f #-
f :: (Int → Int → Bool) → Int → Int → Int
f p x y = if p x y then x + 1
                   else x ‘seq‘ y ‘seq‘ g p (x + 1) (y + 1)
  -# NOINLINE g #-
g :: (Int → Int → Bool) → Int → Int → Int
g _ x 0 = x + 1
g p x y = f p x y
```

(The *NOINLINE* annotations on these mutually-recursive functions prevent GHC from collapsing them into a single looping function, in just the same way that GHC's inlining heuristics prevent the real *h*-functions produced from *tak* from being collapsed together because they are too large.)

With this example, GHC's strictness analyser correctly deduces that *g* is strict in both *Int* arguments, but it can only deduce that *f* is strict in its first *Int* argument, *x*. This prevents GHC from unboxing the *y* argument to *f*, and so the final loop still allocates a *Int* box on every iteration. Notice that if the two functions had been presented as a single loop:

```
g′ :: (Int → Int → Bool) → Int → Int → Int
g′ _ x 0 = x + 1
g′ p x y = if p x y then x + 1
                    else x ‘seq‘ y ‘seq‘ g′ p (x + 1) (y + 1)
```

Then it would be manifestly obvious that g' is strict in both x and y and hence all the *Int* arguments could have been unboxed, leaving an optimal non-allocating loop.

Although this interaction with GHC's strictness analyser is unfortunate, it seems to argue more for improvements to GHC's current strictness analysis mechanism than for a change to our supercompilation approach. We came across several problems with the fragility of GHC's strictness analysis during our work on supercompilation: for example, an earlier version of the supercompiler transformed this same *tak* loop to:

```
tak′ :: Int → Int → Int → Bool → Int
tak′ x y z b = if b then z
                    else tak (let x′ = x − 1 in tak′ x′ y z (not (y < x′)))
                             (let y′ = y − 1 in tak′ y′ z x (not (z < y′)))
                             (let z′ = z − 1 in tak′ z′ x y (not (x < z′)))
tak :: Int → Int → Int → Int
tak x y z = tak′ x y z (not (y < x))
```

With the *tak* loop "rotated" in this way, GHC is only able to deduce that *tak* is strict in the *Int* argument *z*: it cannot detect that the **if**-expression scrutinising the *b* argument will always unleash a demand on the *x* and *y* arguments. As a result, only the *z* argument to *tak′* is unboxed and so the inner loop must perform a lot of allocation.

Test	**Cmp.**[a]	**Run**[b]	**Mem.**[c]	**Size**[d]
Ackermann	+42.2%	+4.7%	+0.0%	-11.0%
AckermannPeano-1	-23.2%	+0.0%	+0.0%	-5.9%
AckermannPeano-2	-4.3%	-0.3%	+0.0%	-5.8%
KMP	-8.6%	-6.1%	+0.0%	-21.7%
SumTree	-35.9%	+0.0%	+0.0%	-4.4%
TreeFlip	-33.4%	+0.0%	+0.0%	+1.5%
ZipMaps	-12.7%	+23.9%	+0.0%	-7.7%
ZipTreeMaps	-71.2%	-81.9%	-100.0%	-12.5%
bernouilli	+8.8%	-0.9%	+0.0%	+1.7%
exp3_8	-11.3%	+1.7%	-0.0%	-11.6%
integrate	+0.0%	+0.0%	+0.4%	+1.3%
primes	-5.2%	+16.7%	+0.1%	+3.0%
queens	-0.7%	+0.0%	-9.5%	+52.2%
rfib	-1.1%	+0.0%	+0.0%	+2.6%
… Unchanged benchmarks elided …				
Min	-71.2%	-81.9%	-100.0%	-21.7%
Max	+42.2%	+23.9%	+0.4%	+52.2%
Geometric Mean	-7.8%	-4.7%	-9.2%	-1.1%

[a] Compile time change when positive information disabled
[b] Program runtime change when positive information disabled
[c] Runtime allocation change when positive information disabled [d] Object file size change when positive information disabled

Figure 7.3: Comparison of supercompilation with and without positive information

7.1.2 Allocation increase due to positive information

The increased allocations in the *primes* and *queens* benchmarks are caused by positive information propagation. To see how positive information propagation can cause our programs to allocate more, consider the following example:

```
f _ [] = []
f p (_ : xs) = case p of (a, b) → p : f p xs
```

The supercompiler will transform this to:

```
f = h1
h1 _ [] = []
h1 p (_ : xs) = h2 p xs
h2 p xs = case p of (a, b) → h3 a b xs
h3 a b xs = let p = (a, b) in p : h1 p xs
```

Notice that the supercompiled code allocates one pair (a, b) for every iteration, whereas the original code only needs to allocate a cons-cell upon each iteration. Sometimes, GHC's later optimisation passes can prevent this allocation. For our example above, *h2* and *h3* will be inlined by GHC into their only use sites to give:

```
f = h1
h1 _ [ ] = [ ]
h1 p (_ : xs) = case p of (a, b) → let p′ = (a, b) in p′ : h1 p′ xs
```

At this point GHC's simplification optimisation pass will detect that p' has the same value as the syntactically enclosing p binding and hence eliminate the allocation of p'. However, this optimisation depends delicately on GHC being able to inline the allocation within a corresponding **case**-scrutinisation, which fails to happen in the *primes* benchmark because the equivalent of *h3* is both shared between several use sites and considered too large to duplicate by GHC's inlining heuristics.

Interestingly, even if positive information propagation is disabled the supercompiled version of *primes* and *queens* will still allocate more than the normally-optimised version (allocation falls by almost 10% in *queens*, but even after this fall it still allocates more than it does without supercompilation). The reason for this is that GHC's own optimisation passes can often cause "reboxing" (i.e. duplication of value allocations) in their own right. In the case of *primes*, the strictness optimisations transform a program similar to:

```
h1 x y ys = case y of Just y′ → case x of Just x′ → case x′ ‘mod‘ y′ ≡ 0 of
  True → case ys of [ ] → [ ]; (y : ys) → h1 x y ys
  False → y : h2 x ys
h2 x ys = case ys of [ ] → [ ]; (y : ys) → h1 x y ys
```

To the more-allocating program:

```
h1 x′ y ys = case y of Just y′ → case x′ ‘mod‘ y′ ≡ 0 of
  True → case ys of [ ] → [ ]; (y : ys) → h1 x′ y ys
  False → y : h2 (Just x′) ys
h2 x ys = case ys of [ ] → [ ]; (y : ys) → case x of Just x′ → h1 x′ y ys
```

In the case of *queens*, the constructor specialisation pass transforms a program similar to:

```
h1 xs = case xs of (y : ys) → h2 xs
h3 xs = case xs of (y : ys) → h1 xs
```

To the more-allocating program:

```
h1 y ys = h2 (y : ys)
h3 xs = case xs of (y : ys) → h1 y ys
```

The general case It is interesting to investigate the effects of disabling positive information propagation in general. The effect of this change is summarised in Figure 7.3. As you might expect, the results are mixed, with some programs suffering because the lack of positive information prevents some allocated constructors from being evaluated away at compile time, whereas in other cases the use of positive information causes the shared allocation of a single constructor to be duplicated into all use sites of that constructor. In the case of *ZipTreeMaps* the lack of positive information causes a better generalisation to be chosen, allowing deforestation of the *buildTree* calls in addition to the *sizeT* (*zipT* (*mapT* ...) (*mapT* ...)) composition.

7.1.3 Allocation increase caused by *reduce* stopping early

The *SumSquare* benchmark suffers because the tag-bag termination test is too restrictive: by causing *reduce* to terminate too early, the supercompiler misses critical deforestation opportunities. The problem is most easily illustrated by the *SumSquare* benchmark, which essentially consists of this Haskell expression:

$$\mathit{foldl'}\ (+)\ 0\ [\,x * y \mid x \leftarrow [\,1\,..\,n\,], y \leftarrow [\,1\,..\,x\,]\,]$$

After reduction and splitting on the term, we are essentially supercompiling the term e_0:

$$\mathit{foldl'}\ (+)\ 0\ [\,x * y \mid x \leftarrow 1 : [\,2\,..\,n\,], y \leftarrow [\,1\,..\,x\,]\,]$$

The first x is known and hence the full structure of the list $[\,1\,..\,x\,]$ can be deduced at compile time. As a consequence, the supercompiler's *reduce* function is able to make considerable progress, statically evaluating the sum as far as the next x:

$$\mathit{foldl'}\ (+)\ 1\ [\,x * y \mid x \leftarrow [\,2\,..\,n\,], y \leftarrow [\,1\,..\,x\,]\,]$$

After another round of reduction and splitting, we supercompile the term e_1:

$$\mathit{foldl'}\ (+)\ 1\ [\,x * y \mid x \leftarrow 2 : [\,3\,..\,n\,], y \leftarrow [\,1\,..\,x\,]\,]$$

For reasons related to our handling of primops this term is not embedded into e_0 (Section E.2), and so we attempt to *reduce*. A perfect *reduce* function would be able to once again evaluate the sum over $y \leftarrow [\,1\,..\,2\,]$ here. Unfortunately, doing so requires two iterations of the *foldl′* loop: one for $y = 1$ and one for $y = 2$. The tag-bag termination test that we use in *reduce* is incapable of distinguishing between these two iterations of the loop, and so it blows the whistle on the $y = 2$ iteration, and *reduce* returns a term with some potential β-reductions. The subsequent *split* then recursively supercompiles the list-*generating* comprehension separately from the list-*consuming* call to *foldl′*, and we lose all hope for deforestation.

The general case It is interesting to consider to what extent a lack of power in our *reduce* termination test is causing the output of supercompilation to be suboptimal. To investigate this, we changed our *reduce* loop so that it would tolerate 100 failures of the termination test before giving up, rather than the single failure that would normally prompt *reduce* to stop. With this modified *reduce* we obtained the supercompilation results of Figure 7.4. The results show that false non-termination is a serious problem: many benchmarks were improved by using a *reduce* function which is less likely to report false non-termination. Nonetheless, several benchmarks had at least one instance where *reduce* terminated too early but were not affected by this change, namely *Accumulator*, *ZipTreeMaps*, *gen_regexps*, *paraffins*, *primes*, $\mathit{wheel} - \mathit{sieve1}$ and $\mathit{wheel} - \mathit{sieve2}$.

7.2 Effect of disabling *sc*-rollback

In Figure 7.5 we show the effects of disabling the *sc*-rollback mechanism described in Section 6.1.2. Unlike *reduce*-rollback, *sc*-rollback affects almost all of the benchmarks because almost every benchmark will fail a *sc* termination test at some point, whereas failing a *reduce* termination test is comparatively rare. The effect of disabling *sc*-rollback can be dramatic: with it disabled, the size of the *ZipTreeMaps* binary grew by almost 1000 percent. Furthermore, with these flags the *queens* benchmark was miscompiled by our

Test	**Cmp.**[a]	**Run**[b]	**Mem.**[c]	**Size**[d]
Ackermann	-11.0%	+0.0%	+0.0%	-1.8%
AckermannPeano-1	+65.2%	+0.0%	+0.0%	-11.1%
AckermannPeano-2	+11.5%	+0.1%	+0.0%	-4.5%
SumSquare	+8.9%	-94.5%	-100.0%	-28.0%
bernouilli	+0.5%	-0.9%	-0.0%	-5.3%
integrate	+13.1%	-75.0%	-85.2%	-30.5%
… Unchanged benchmarks elided …				
Min	-11.0%	-94.5%	-100.0%	-30.5%
Max	+65.2%	+4.2%	+0.0%	+0.0%
Geometric Mean	+1.9%	-13.8%	-14.2%	-2.9%

[a] Compile time change when term. test made laxer
[b] Program runtime change when term. test made laxer
[c] Runtime allocation change when term. test made laxer
[d] Object file size change when term. test made laxer

Figure 7.4: Comparison of supercompilation without and with lenient *reduce* termination

Test	**Cmp.**[a]	**Run**[b]	**Mem.**[c]	**Size**[d]
Ackermann	+261.7%	-3.5%	+0.0%	+68.7%
AckermannPeano-1	-8.7%	+100.0%	+0.0%	+13.0%
AckermannPeano-2	+0.1%	-0.3%	-0.0%	+24.2%
EvenDouble	+0.0%	-0.6%	-0.0%	+5.5%
KMP	+20.7%	+57.6%	+0.0%	+12.2%
MapMapFusion	+4.8%	+10.0%	+0.0%	+11.3%
ReverseReverse	+0.0%	+0.7%	+0.0%	+14.4%
SumSquare	+0.0%	-2.9%	-0.0%	+24.6%
SumTree	+76.4%	-20.0%	+38761.6%	+124.6%
TreeFlip	+99.7%	-9.1%	+2.3%	+131.0%
ZipMaps	-7.3%	-6.3%	+0.0%	+5.9%
ZipTreeMaps	+1176.3%	-42.8%	-47.4%	+959.0%
bernouilli	+53.1%	+0.0%	+0.0%	+65.3%
digits-of-e1	+0.5%	-2.5%	-0.0%	+262.6%
exp3_8	+19.7%	+0.0%	-0.1%	+28.9%
integrate	+1.0%	-2.5%	+0.0%	+3.6%
primes	+1.2%	+0.0%	-0.0%	+0.6%
queens	+493.1%	N/A	N/A	+761.2%
tak	+10.9%	+0.0%	-1.9%	+46.4%
x2n1	+16.9%	+0.0%	+0.0%	+22.3%
… Unchanged benchmarks elided …				
Min	-8.7%	-42.8%	-47.4%	+0.0%
Max	+1176.3%	+100.0%	+38761.6%	+959.0%
Geometric Mean	+26.8%	+0.6%	+18.7%	+38.3%

[a] Compile time change when *sc* rb. disabled
[b] Program runtime change when *sc* rb. disabled
[c] Runtime allocation change when *sc* rb. disabled
[d] Object file size change when *sc* rb. disabled

Figure 7.5: Comparison of supercompilation with and without *sc*-rollback

supercompiler implementation due to an issue with name handling: supercompilation of *queens* terminated, but incorrect results are generated at runtime by the output program.

As expected, enabling *sc*-rollback tends to decrease the size of the generated code: in fact, none of the benchmarks had their supercompiled code shrink when disabling

Test	Cmp.[a]	Run[b]	Mem.[c]	Size[d]
Ackermann	-9.1%	+0.0%	+0.0%	-0.4%
AckermannPeano-1	+7.2%	+20.0%	+0.0%	-7.2%
AckermannPeano-2	+1.9%	+0.3%	+0.0%	+0.4%
SumSquare	+22.0%	+4.6%	+0.0%	+4.2%
ZipTreeMaps	+9.6%	-84.3%	-100.0%	-5.3%
bernouilli	-4.0%	-0.9%	-0.0%	-5.7%
integrate	+3.8%	-25.0%	-24.2%	-8.9%
primes	-0.4%	+0.0%	-0.0%	+0.6%
... Unchanged benchmarks elided ...				
Min	-9.1%	-84.3%	-100.0%	-8.9%
Max	+22.0%	+20.0%	+0.0%	+4.2%
Geometric Mean	-0.6%	-6.8%	-9.7%	-0.7%

[a] Compile time change when *reduce* rb. disabled

[b] Program runtime change when *reduce* rb. disabled

[c] Runtime allocation change when *reduce* rb. disabled

[d] Object file size change when *reduce* rb. disabled

Figure 7.6: Comparison of supercompilation with and without *reduce*-rollback

sc-rollback. In most cases, this code size decrease was accompanied by a decrease in supercompilation time, though this was by no means universal. The effect on program run time and allocation was more varied: most programs were unaffected by disabling *sc*-rollback, but a few benchmarks exhibited greatly increased (e.g. *SumTree*) or decreased (*ZipTreeMaps*) allocations. Generally the results indicate that optimisation opportunities are not being sacrificed by use of *sc*-rollback, and that *sc*-rollback is an extremely effective technique for reducing supercompilation time and output code size.

7.3 Effect of disabling *reduce*-rollback

In Section 6.1.1 we described a mechanism whereby we would roll back the reduction performed by *reduce* should the embedded termination check ever fail. Figure 7.6 summarises the effect of disabling this feature in the supercompiler. The results are mixed: *SumSquare* becomes substantially larger and slower with this change, but *integrate* and *ZipTreeMaps* become both smaller and faster.

It is unclear why disabling *reduce*-rollback improves the *integrate* and *ZipTreeMaps* benchmark. The most that can be said is that disabling the feature causes both benchmarks to generalise away less information during supercompilation invocations subsequent to the failing *reduce*. These two benchmarks seem to be very sensitive to the exact supercompilation algorithm, with almost any change causing generalisation to discover a core loop which is very efficient and compact in comparison with the non-deforested loop discovered by our standard algorithm. For example, we will see in Section 7.7 that disabling speculation in the supercompiler also greatly improves these two benchmarks: this result is particularly surprising when you consider that *reduce*-rollback has the effect of *reducing* the amount of reduction that the supercompiler does, but speculation has the effect of *increasing* the amount of reduction performed, and yet either change will improve the generalisation of these two benchmarks.

The *gen_regexps*, *paraffins*, $wheel - sieve1$ and $wheel - sieve2$ benchmarks all experienced at least one failure of the *reduce* termination test, but their overall results were unaffected by disabling *reduce*-rollback.

7.4 Effect of disabling generalisation

Test	**Cmp.**[a]	**Run**[b]	**Mem.**[c]	**Size**[d]
Accumulator	+18.9%	+33.7%	+70.0%	+11.9%
Ackermann	+223.4%	-44.7%	-50.8%	-9.6%
AckermannPeano-1	-15.9%	+0.0%	+0.0%	-20.2%
AckermannPeano-2	+1.2%	+17.7%	+13.2%	+28.5%
EvenDouble	-7.0%	+1.7%	+0.0%	-1.4%
Factorial	-6.9%	+0.0%	+0.0%	+2.4%
KMP	-8.6%	+98.8%	+0.0%	-17.9%
MapMapFusion	+7.1%	+126.0%	+111.1%	+16.8%
ReverseReverse	-4.3%	+0.0%	+0.0%	-4.1%
SumSquare	+61.0%	+5.5%	-0.0%	+34.7%
SumTree	-58.2%	+664.0%	+708255.3%	+10.9%
TreeFlip	-51.7%	+422.7%	+398308.2%	+10.9%
ZipMaps	-1.8%	+220.5%	+255.5%	-1.4%
ZipTreeMaps	-70.3%	+113.9%	+171.6%	-7.6%
bernouilli	+10.8%	-0.9%	+4.1%	+9.6%
digits-of-e1	-11.5%	-2.5%	-0.1%	-25.5%
exp3_8	+0.7%	-2.5%	+0.5%	+14.9%
integrate	-6.9%	+0.0%	+13.6%	+3.0%
primes	-14.0%	+0.0%	-13.8%	+21.7%
queens	+1930.9%	+100.0%	+493.8%	+627.5%
tak	N/A	N/A	N/A	N/A
x2n1	+25.5%	+0.0%	+301.2%	+29.2%
. . . Unchanged benchmarks elided . . .				
Min	-70.3%	-44.7%	-50.8%	-25.5%
Max	+1930.9%	+664.0%	+708255.3%	+627.5%
Geometric Mean	+4.8%	+31.3%	+112.1%	+9.4%

[a] Compile time change when generalisation disabled

[b] Program runtime change when generalisation disabled

[c] Runtime allocation change when generalisation disabled

[d] Object file size change when generalisation disabled

Figure 7.7: Comparison of supercompilation with and without generalisation

It is well known [Sørensen and Glück, 1995] that careful choice of generalisation heuristic (Section 6.2) is crucial to achieving good performance from a supercompiler. It should therefore come as no surprise that disabling generalisation altogether (and therefore always continuing to supercompile using *split* when the termination test in *sc* fails) is severely detrimental to performance. The results of disabling generalisation in our supercompiler are presented in Figure 7.7. Overall, the results show that on average generalisation greatly reduces not only the runtime and allocations of the generated code, but also tends to reduce output code size. One benchmark, *tak*, was not observed to terminate with generalisation turned off.

Perhaps the most surprising feature of these results is that the *Ackermann* and *primes* benchmarks appear to actually be improved by using *split* rather than a generalisation heuristic. The improvement in allocation for *Ackermann* is more inconsequential than the −50.8% change would imply, as the inner loop of the benchmark does not allocate at all, regardless of whether generalisation is turned on or off. The runtime reduction is significant, however, and appears to occur because the (purely numerical) inner loop generated with generalisation turned off happens to be a little more unrolled than the version generated by generalisation.

The *primes* program allocates less because the version without generalisation ends up residualising a completely inlined call to the *iterate* library function which is subsequently subject to shortcut fusion. On the other hand, the version of the program compiled with supercompiler generalisation is able to inline the *iterate* call but the supercompiler is not then able to follow through and achieve the same deforestation gains that shortcut fusion spots. As the resulting inlined copy of *iterate* is not subject to shortcut fusion, the net effect is that the program produced with generalisation allocates more.

Test	**Cmp.**[a]	**Run**[b]	**Mem.**[c]	**Size**[d]
Accumulator	+17.0%	+36.2%	+70.0%	+11.9%
Ackermann	+225.3%	-44.7%	-50.8%	-9.6%
AckermannPeano-1	-15.9%	+0.0%	+0.0%	-20.2%
AckermannPeano-2	+3.5%	+17.9%	+13.2%	+29.0%
EvenDouble	-7.0%	+0.0%	+0.0%	-1.4%
Factorial	-5.2%	+0.0%	+0.0%	+2.4%
ReverseReverse	-4.3%	+0.0%	+0.0%	-4.1%
SumSquare	+56.1%	+208.8%	+0.0%	+13.3%
SumTree	-58.7%	+668.0%	+708255.3%	+10.9%
TreeFlip	-51.7%	+422.7%	+398308.2%	+10.9%
ZipMaps	-9.1%	-0.6%	+0.0%	+0.2%
ZipTreeMaps	-70.1%	+114.5%	+171.6%	-7.6%
bernouilli	+10.8%	-0.9%	+4.1%	+9.6%
digits-of-e1	-12.0%	-3.7%	-0.1%	-25.5%
exp3_8	+1.6%	-4.2%	+0.5%	+14.9%
integrate	-6.6%	+2.5%	+13.6%	+3.0%
primes	-4.9%	+0.0%	-14.5%	+20.5%
queens	+1356.4%	N/A	N/A	+350.3%
tak	+12954.5%	+0.0%	-31.6%	+31.6%
x2n1	+24.4%	+0.0%	+301.2%	+29.2%
... Unchanged benchmarks elided ...				
Min	-70.1%	-44.7%	-50.8%	-25.5%
Max	+12954.5%	+668.0%	+708255.3%	+350.3%
Geometric Mean	+20.4%	+24.2%	+85.3%	+7.3%

[a] Compile time change when growing tags used
[b] Program runtime change when growing tags used
[c] Runtime allocation change when growing tags used
[d] Object file size change when growing tags used

Figure 7.8: Comparison of supercompilation with MSG and growing-tags generalisation

7.5 Effect of generalising with growing-tags rather than MSG

It is instructive to compare the effectiveness of using growing-tag generalisation (Section 6.2.2) instead of MSG-based generalisation (Section 6.2.3). The growing-tag generalisation is a rough-and-ready generalisation which is easier to implement than a MSG-based generalisation, but which we expect to perform worse in practice. Indeed, this is what we find in the results of Figure 7.8[1]: growing-tag generalisation is somewhat better than

[1] Just as in Section 7.2 we find that *queens* is miscompiled

having no generalisation at all, but not as good as the full MSG-based generalisation.

Test	**Cmp.**[a]	**Run**[b]	**Mem.**[c]	**Size**[d]
ZipTreeMaps	+35.8%	+0.6%	-0.0%	+5.7%
bernouilli	+0.4%	-0.9%	+0.0%	+1.4%
digits-of-e1	+0.1%	-3.7%	+0.0%	+2.6%
digits-of-e2	+1.4%	+0.7%	+0.0%	+2.1%
. . . Unchanged benchmarks elided . . .				
Min	-7.3%	-4.5%	-0.0%	+0.0%
Max	+35.8%	+2.0%	+0.0%	+5.7%
Geometric Mean	-0.2%	-0.4%	-0.0%	+0.4%

[a] Compile time change when type gen. disabled

[b] Program runtime change when type gen. disabled

[c] Runtime allocation change when type gen. disabled

[d] Object file size change when type gen. disabled

Figure 7.9: Comparison of supercompilation with and without type generalisation

7.6 Effect of disabling type generalisation

In Section 6.2.6 we introduced type generalisation: a technique for eagerly generalising away type information in order to ameliorate the problem of type overspecialisation. Because our benchmarks are small, they generally only instantiate the polymorphic functions that they use at one or two types, and so there are few opportunities for type generalisation to make a difference. The benchmark data for those benchmarks which *are* affected by disabling type generalisation are presented in Figure 7.9. Generally we see that disabling the feature has a modest worsening effect on code size and little-to-no effect on the performance of the supercompiled program, as we would expect.

The type generalisation opportunities in these programs are usually of a modest nature. Across all benchmarks, we often see that pairs of states such as $x :: \mathit{Int}$ and $x :: \mathit{Bool}$ will be generalised to $x :: \alpha$. Another common piece of code which is often subject to type generalisation is the function composition $f\ (g\ x)$.

7.7 Effect of disabling speculation

The effects of disabling the **let**-speculation mechanism of Section 6.3 are summarised in Figure 7.10. The results are a mixed bag: some benchmarks (such as *ZipTreeMaps*) are greatly improved by disabling speculation, while others (such as *x2n1*) are worsened. On average, the results indicate that disabling speculation actually improves code size, runtime and allocations. This unimpressive showing for speculation is not entirely unexpected, given that the benchmarks in our test suite do not make substantial use of idioms (such as partial applications) which speculation is particularly targeted towards optimising.

There are three principal benchmarks which benefit greatly from disabling speculation: *SumSquare*, *ZipTreeMaps* and *integrate*. In the case of *SumSquare*, this occurs because the main loop uses an auxilliary function *strictSum* defined by $\mathit{strictSum} = \mathit{foldl'}\ (+)\ 0$. Without speculation, the supercompiler cannot detect that this is in fact a cheap partial application, and as a result the output program is mostly λ-abstracted over *strictSum*. Although this destroys any opportunity for deforesting the list consumed by *strictSum*, this loss of information does have the benefit of causing the termination test in *reduce* to

Test	Cmp.[a]	Run[b]	Mem.[c]	Size[d]
Accumulator	+1.9%	+3.3%	+0.0%	+0.0%
Ackermann	+57.8%	+0.0%	+0.0%	-3.3%
AckermannPeano-1	-5.8%	+20.0%	+0.0%	+19.4%
AckermannPeano-2	-23.3%	-0.6%	+0.0%	-20.6%
Append	-4.7%	-2.5%	+0.0%	+0.0%
EvenDouble	-7.0%	+0.0%	+0.0%	+2.0%
EvenDoubleGenerator	-4.9%	+125.0%	+140.0%	+6.4%
Factorial	-3.4%	+0.0%	+0.0%	+0.0%
KMP	-25.9%	+61.2%	+0.0%	-39.8%
LetRec	+8.9%	+0.0%	+0.0%	+0.5%
MapMapFusion	+2.4%	+0.0%	+0.0%	+0.0%
Raytracer	+9.4%	-2.7%	+0.0%	+28.9%
ReverseReverse	+0.0%	+0.0%	+0.0%	+0.0%
SumSquare	+42.3%	-55.1%	-56.5%	-14.5%
SumTree	-5.2%	+0.0%	+0.0%	+0.0%
TreeFlip	-3.4%	-9.1%	+0.0%	+0.0%
ZipMaps	-7.3%	-0.6%	+0.0%	+0.2%
ZipTreeMaps	-54.7%	-84.9%	-100.0%	-7.3%
bernouilli	-3.3%	+3.8%	+2.2%	-15.2%
digits-of-e1	-0.4%	+4.5%	+0.0%	+3.4%
digits-of-e2	-10.8%	+103.7%	+94.5%	-13.7%
exp3_8	-16.6%	+0.0%	+0.0%	-27.5%
gen_regexps	-5.1%	+0.0%	-0.3%	-5.2%
integrate	-1.8%	-75.0%	-81.5%	-28.3%
paraffins	-1.9%	+0.0%	+0.0%	+0.0%
primes	-22.4%	+0.0%	+0.9%	-6.2%
queens	+15.1%	+0.0%	+8.2%	-12.5%
rfib	+0.9%	+0.0%	+0.0%	+0.0%
tak	-1.4%	+0.0%	+0.0%	+0.0%
wheel-sieve1	-2.3%	+0.0%	+0.0%	+0.0%
wheel-sieve2	-4.7%	+0.0%	-0.0%	-7.5%
x2n1	-18.3%	+0.0%	+337.2%	+42.6%
Min	-54.7%	-84.9%	-100.0%	-39.8%
Max	+57.8%	+125.0%	+337.2%	+42.6%
Geometric Mean	-4.8%	-6.5%	-7.2%	-4.3%

[a] Compile time change when speculation disabled
[b] Program runtime change when speculation disabled
[c] Runtime allocation change when speculation disabled
[d] Object file size change when speculation disabled

Figure 7.10: Comparison of supercompilation with and without speculation

not fail (cf. Section 7.1.3), which leads to a better generalisation being made in the same manner as we described in Section 7.1.3.

For *ZipTreeMaps* and *integrate*, the lack of speculation causes the supercompiler to residualise various shared lists to prevent work duplication. These residualisation decisions happen to be better generalisations for these two benchmarks than the generalisations that the supercompiler would otherwise choose in the presence of full information (we already noted in Section 7.3 that *ZipTreeMaps* and *SumSquare* are very sensitive to choice of generalisation).

7.8 Performance of the supercompiler

The supercompiler is extremely slow compared to the existing optimisation techniques implemented by GHC. We have benchmarked our supercompiler using the standard profiling tools for GHC-generated code, and found that the time spent by the supercompiler is roughly divided up as follows:

- 75% of supercompilation time is spent matching and MSGing terms (since matching is implemented in terms of MSG, these timings are conflated).
- 4% of time is spent computing *split*.
- 3% of time is spent on heap speculation.
- 18% of time is spent on name management such as collecting sets of free variables and looking up binders.

These results suggest that future work should pay more attention to an efficient implementation of state matching. Two ideas that seem particularly promisising are as follows:

1. λ-lifting essentially assigns every piece of code in the program a unique name. If we were to λ-lift all functions and case branches then we would be able to quickly compare two functions just by comparing these unique names, and we would never have to match under binders. This could considerably speed up pairwise matching.
2. At the moment memoisation is done by trying to match the state of interest against every previous state, one-by-one. Another approach would be to build a generalised trie [Hinze et al., 2000] keyed off of the state we wish to match. Careful thought would have to be given as to how this should work, given the work-sharing constraints of matching described in Chapter 5.

Chapter 8

Related work

In this chapter we survey the related literature. We begin with a very brief tour of the supercompilation literature Section 8.2, and then move on to discuss the closely related areas of deforestation (Section 8.3), shortcut fusion (Section 8.4), partial evaluation (Section 8.5) and online termination testing (Section 8.6), describing how each is related to our own work. Finally, in Section 8.7 we discuss those areas of related work which fail to fall under one of these broad headings.

8.1 Program derivation systems

Early work on program derivation focused on defining systems which could be used to find correct equivalences between programs, with little focus on how a program could automatically use such a system to find interesting equivalences. The fold-unfold framework [Burstall and Darlington, 1977] is the most prominent example of this era: this system uses the rules of definition-introduction, instantiation, unfolding, folding and abstraction to derive new programs. This simple and expressive approach can be seen as forming the basis of all of the program transformation techniques described in this chapter, including supercompilation itself.

Another prominent approach is the expression procedures of Scherlis [1981], which although less expressive than fold-unfold, can still in practice achieve most transformations of interest, and unlike the fold-unfold rules are guaranteed to preserve total correctness. Recent work [Tullsen et al., 1999] has shown there to be a close connection between expression procedures and fix-point fusion [Meijer et al., 1991].

8.2 Supercompilation

Supercompilation was introduced by Turchin [1986] as a program transformation technique for a functional programming language known as Refal. The fundamental concept is to transform an initial term into a potentially-infinite tree of “configurations” by *driving*: i.e. the unfolding of function definitions at their use sites. This possibly-infinite *process tree* is then converted to a certainly-finite graph by means of tying back configurations to previous ones where possible. If tying back is insufficient to produce a finite graph, configurations would be *generalised*, throwing away information and then re-driving the resulting configuration in the hope of eventually achieving tieback by choosing appropriate generalisations. Later work by Turchin would go into much more detail about the critical generalisation stage [Turchin, 1988].

The positive supercompiler of Sørensen et al. [1993] was a landmark in that it was the first presentation of the ideas of supercompilation in a traditional formalism for a standard first-order functional language. Unlike Turchin's original supercompiler, only positive information (that which can be represented by syntactic substitution) was propagated: *negative* information about which values a variable *could not* have was not propagated [Secher and Sørensen, 2000]. Later work described generalisation for the positive supercompiler [Sørensen and Glück, 1995].

Later supercompilers extended positive supercompilation to higher order languages with call-by-value [Jonsson and Nordlander, 2009], call-by-need [Mitchell, 2008, 2010] and call-by-name [Klyuchnikov, 2009] evaluation strategies. Other strands of research look at making supercompilation perform more powerful program transformations. Both distillation [Hamilton, 2007] and higher-level supercompilation [Klyuchnikov, 2010c] are capable of achieving superlinear speedups such as transforming a naive quadratic list reverse function into an efficient linear one. These results are achieved by using more powerful methods for deciding term equivalence than the syntactic equivalence a standard supercompiler will rely on.

8.3 Deforestation

Deforestation [Wadler, 1988] is an algorithm explicitly designed for removing intermediate data structures from programs. Deforestation as originally presented was only suitable for optimising first-order programs in so-called "treeless" form (identified via a syntactic test).

The treeless form is rather restricted: in particular, there is no **let**, arguments to function calls may only be variables, and function arguments have to be linear. Few real programs meet these requirements, so the the original paper proposed two extensions to handle more programs. Firstly, the algorithm was extended to handle those higher order functions expressible as non-recursive "higher order macros": with the addition of a **where** facility for local function definition, this allowed certain higher-order functions to be handled, including *map*. Secondly, non-treeless code was allowed to occur at certain positions in the input of deforestation by means of using type information to mark those terms which were not to be deforested—a technique known as "blazing".

Later work improved the applicability of deforestation by allowing blazing more often, and relaxing the linearity restriction—typically by using sharing analysis to discover more linear terms [Chin, 1990; Hamilton, 1993]. Deforestation was also extended to gracefully handle higher-order functions [Marlow and Wadler, 1992; Hamilton, 2005; Marlow, 1996].

Although the deforestation algorithm can be seen as a form of partial evaluator designed to statically evaluate **case** statements that scrutinise known constructors, deforestation is typically not as powerful as either supercompilation or partial evaluators due to the fact that deforestation propagates less information [Sørensen et al., 1994].

Relation to our work The algorithm outlined in this thesis can be described as a supercompiler in the formalisation of Sørensen et al. [1994], as we are usually careful to propagate positive information. However, the stated implementation of the matcher can sometimes discard positive information (Section 5.2.1). This would be straightforward to change, but we often find that positive information is not important for optimisation purposes (Section 7.1.2).

8.4 Short-cut deforestation

The "short-cut" family of deforestation techniques [Gill et al., 1993; Svenningsson, 2002; Coutts et al., 2007] are useful practical techniques for eliminating intermediate data structures from programs. They rely on rewriting library function such as *map*, *filter* and *foldr* in a stylised way, such that producer/consumer relationships between them can, after the compiler has performed some standard inlining steps, be spotted by a simple compiler rewrite rule and thus optimised.

Short-cut fusion techniques are much less powerful than supercompilation: the input program has be manually rewritten for these techniques to have any effect at all, they are typically aimed only at deforesting list data structures, and there is no way for them to achieve effects such as function specialisation. However, their simplicity and predictability has made them popular: in particular, shortcut fusion is the deforestation method implemented by GHC.

Relation to our work Shortcut deforestation techniques have the considerable benefit that they are usually more predictable than supercompilation. For example, Coutts [2010] proves that under some reasonable assumptions about the optimising power of the underlying compiler, stream fusion is guaranteed to remove intermediate lists. This predictable optimising power and relatively low time complexity make them more suitable for use as a compiler optimisation pass than supercompilation, deficits which our work has not addressed.

8.5 Partial evaluation

A considerable literature exists on the process of evaluating programs symbolically at compile time in order to achieve optimisation, under the general heading of *partial evaluation*: a good (if dated) survey of the literature is Consel and Danvy [1993]. Classic work on partial evaluation [Jones et al., 1985; Sestoft, 1986; Jones et al., 1993] operated under considerable restrictions: for example, only arguments whose *whole* structure was known at compile time could be specialised on. It was impossible to specialise a function on e.g. a particular shape of input list but leave it generalised over what the elements of that list actually are. Later work lifted these restrictions [Consel and Danvy, 1991; Bondorf, 1992] and even allowed functions to be specialised on abstract *properties* of their arguments [Consel and Khoo, 1993].

Later discoveries [Glück and Jørgensen, 1994; Nielsen and Sørensen, 1995] allowed partial evaluators to match and exceed the power of the deforestation algorithm.

Supercompilers, being symbolic evaluators, clearly have a considerable amount of overlap with partial evaluation approaches. The question naturally arises as to what particular features sets a supercompiler apart from the more general class of partial evaluators.

The prevailing view [Sørensen et al., 1994] is that supercompilation is distinguished from partial evaluation by its propagation of positive (and, optionally, negative) information gleaned from residual **case** expressions or equality comparisons. This extra information allows a supercompiler that propagates both positive and negative information (i.e. a "perfect supercompiler") to transform a naive string searching algorithm to an instance of the KMP string search algorithm [Knuth et al., 1977], given that the string to be found is fixed at compile time [Secher and Sørensen, 2000; Sørensen et al., 1993].[1]

[1] A positive supercompiler is not able to achieve a KMP-optimal algorithm, but its use of positive information does allow some extra reductions to be done compared to what can be achieved by partial

Another major difference between the supercompilation and partial evaluation school of thought has been the method of ensuring termination. Work on partial evaluators has traditionally used a Mix-like approach [Jones et al., 1989] based on an offline (i.e. prior to specialisation) *binding time analysis* which attempts to discover which function parameters are available at compile time and hence specialisable. In contrast, supercompilation has a strong tradition of using an online (i.e. during specialisation) termination test such as a well-quasi-order [Leuschel, 1998]. However, this criteria alone does not distinguish a supercompiler from a partial evaluator because online termination methods are also popular in partial evaluation work, an approach which was popularised by Fuse [Weise et al., 1991; Ruf, 1993]. Another interesting approach is that taken by deforestation: the deforestation algorithm is only guaranteed to terminate when used on treeless expressions, in which case termination is ensured by the memoisation aspect of the algorithm.

Relation to our work The symbolic evaluation of call-by-need programming languages has been considered by the partial evaluation community. Jørgensen has previously produced a compiler for call-by-need through partial evaluation of a Scheme partial evaluator with respect to an interpreter for the lazy language [Jørgensen, 1992]. His work made use of a partial evaluator capable of dealing with the *set*! primitive, which was used to implement updateable thunks. Our supercompiler takes a direct approach that avoids the need for any imperative features in the language being supercompiled. A direct approach is essential in our GHC-based implementation of the supercompiler, since the intermediate language which is transformed by optimisation passes is a simple call-by-need functional language, which intentionally does not contain primitives which directly manipulate the contents of the thunks.

8.6 Online termination tests

The use of the homeomorphic embedding well-quasi-order for termination testing in supercompilation was introduced in Sørensen and Glück [1995] and has become the prevailing approach in the field. A notable refinement of this idea is the well-quasi-order of HOSC [Klyuchnikov, 2010a], which proposes an extension of the homeomorphic embedding which is more suitable for terms that contain bound variables.

Testing the homeomorphic embedding can be a bottleneck in a supercompiler implementation [Mitchell and Runciman, 2008]. One approach to solving this is to reduce the constant factors involved such as by testing multiple embeddings in parallel [Jonsson, 2011], but another approach is to use a more efficient, but weaker embedding that is nonetheless sufficient for optimising many programs of practical interest [Mitchell, 2010].

Relation to our work In this thesis we have adopted a version of the tag-bag termination test of [Mitchell, 2010], adjusted to our setting where we supercompile abstract machine states rather than simple expressions.

In [Mitchell, 2010], Mitchell uses tag-bags in a similar way to us, but only associates tags with let-bound variables. In order to tag every subexpression, he keeps terms in a normal form where all subexpressions are let-bound (NB: this is more restrictive than ANF, where only subexpressions in *argument* positions are let-bound). Supercompiling *State*s and tagging subterms directly means that we can avoid let-floating and—because we distinguish between tags from subexpressions currently being evaluated (in the stack),

evaluation.

and those subexpressions that are not in the process of being forced (in the heap)—our termination criterion is a little more lenient.

Furthermore, we make use of an implementation of the termination test that exploits various properties of well-quasi-orders to speed up the test (Section D.5).

We find that this test gives excellent performance in practice: little (on the order of 6%) of the runtime of our implementation is spent testing the termination condition, and we have observed only a few examples where the termination test is too weak to achieve a desirable optimisation.

8.7 Other related work

Supero 2010 [Mitchell, 2010] used a generalisation mechanism for call-by-need supercompilers that uses the tag-bag termination mechanism. The idea is that when computing a child term for recursive supercompilation, the splitting process avoids pushing down any bindings that would cause the new child state to immediately fail the termination check. We found that Supero's generalisation method gives very similar results to our growing-tag generalisation method (Section 6.2.2) in almost all situations. However, our method has two merits:

- It is somewhat easier to implement, because the splitter does not need to try pushing down bindings in any particular order—the growing tags precisely indicate those bindings which may be pushed down.

- It is faster: Supero's method requires one termination test to be carried out for every binding one may wish to inline, whereas our method simply requires a set membership test per inlining.

Our use of rollback (Section 6.1) is present in standard supercompilers constructed around the idea of a "graph of configurations" [Sørensen and Glück, 1999; Klyuchnikov, 2009]. In such systems, when an earlier graph node a is embedded into a later node b, the a node (rather than the b node) is generalised. This has the effect of cutting off the part of the graph reachable via a, achieving rollback. We have described how this idea can not only be applied to a direct-style supercompiler, but have also extended the the concept to rolling back "loopy" reduction (Section 6.1.1) and heap speculation (Section 6.3).

Traditional graph-based approaches to supercompilation make a distinction between "upward" and "downward" generalisation steps, with "upward" steps rolling back to generalise and "downward" steps (triggered by instance matches) not rolling back. The equivalent distinction in our supercompiler arises when we come to choose whether to use a rollback opportunity or not (Section 6.2.4).

One clear advantage using an explicit partial process graph in supercompilation is that it allows the retention in the graph of disconnected nodes which have been made unreachable by rollback, but which we may later want to resume supercompilation of. In contrast, in Section 6.1 we described how our rollback mechanism has to carefully discard any work done towards fulfilling those promises we roll back past. In our formulation of supercompilation, this discarded work will be redone if the algorithm later needs to supercompile similar states.

Our decision to use speculative evaluation in the supercompiler was initially inspired by Ennals's work [Ennals and Jones, 2003] on an alternative evaluation strategies for Haskell which speculates fresh heap bindings but still has the same observational semantics as standard call-by-need.

Chapter 9

Further work and conclusions

There are a number of avenues which further work in call-by-need supercompilation could take. In this section we briefly survey both some obvious and less obvious directions.

9.1 Formal verification of module properties

Throughout the thesis, we have clearly defined the properties that we expected each of the modules (termination, memoisation, splitting and evaluation) of our modular supercompiler to have. Assuming that these properties hold, we have been able to prove that the supercompilation algorithm is correct. However, in some cases we have only been able to make informal arguments as to why our implementations of the modules obey the stated properties: this problem is particularly acute in case of the *split* function (Chapter 4), but also applies to the *msg* function (Chapter 5). Although we are confident that the properties hold, it would be useful to formally verify that our implementations are correct.

9.2 Binding structure in the termination test

We use a tag-bag termination test in our supercompiler. We found (Section 7.1.3) that this test is in practice almost always sufficiently powerful to avoid generalisation occurring too early, but sometimes causes the supercompiler to terminate earlier than a standard homeomorphic embedding would, to the detriment of the generated code.

The root of the problem is that a tag-bag termination test is insensitive to binding structure. For example, *reduce* cannot completely reduce away f in the following because the tag-bag termination test can't observe that with every iteration of f the argument becomes strictly smaller, since bs is kept alive by the use in the outermost **case**-frame:

$$
\begin{array}{l}
\mathbf{let}\, f\ [\,] \qquad\quad = \mathit{True} \\
\quad f\ (x : xs) = \mathbf{case}\, f\ xs\ \mathbf{of}\ \begin{array}{l} \mathit{True} \to \mathit{False} \\ \mathit{False} \to \mathit{True} \end{array} \\
\quad bs = [\,\mathit{True}, \mathit{False}\,] \\
\mathbf{in\ case}\, f\ bs\ \mathbf{of}\ \begin{array}{l} \mathit{True} \to bs \\ \mathit{False} \to [\,] \end{array}
\end{array}
$$

Interesting future work would be to combine the tag-bag termination test with a stronger test similar to a homeomorphic embedding in order to get very fast termination tests in the common case where the tag-bag test passes, but avoid early termination if the tag-bag test fails but the more powerful test still passes.

The main problem with using a standard homeomorphic embedding as the powerful termination test it that the we wish to use it with states, which due to their potentially mutually-recursive binding structure are perhaps better modelled by graphs or infinite trees rather than the finite trees used to model normal terms in previous work. Future work could explore how the graph minor well-quasi-order [Robertson and Seymour, 1990] or well-quasi-orders on infinite trees [Nash-Williams, 1965] could be combined with existing work on homeomorphic embeddings for trees containing bound variables [Klyuchnikov, 2010a] to obtain a termination test suitable for states.

9.3 Code explosion

Supercompiling a program can sometimes vastly increase the size of the programs that it optimises. Consider, for example, this program:

```
data Nat = Z | S Nat
bs :: [Bool]
bs = [True, False]
sequence :: [[a]] → [[a]]
sequence [] = [[]]
sequence (mx : mxs) = concatMap (λx → map (x:) (sequence mxs)) mx
count :: [Bool] → Nat
count = foldr (λx acc → if x then S acc else acc) Z
root1 = map count (sequence [bs])
root2 = map count (sequence [bs, bs])
root3 = map count (sequence [bs, bs, bs])
root3 = map count (sequence [bs, bs, bs, bs])
```

The term *rootn* enumerates all n-length lists of Booleans and then counts the number of *True*s in each list. As a result, there are 2^n items in the list returned by *rootn*. Most supercompiler implementations will statically evaluate *rootn* until it reveals the entire output list at compile time[1]. Therefore, as we increase n we linearly increase the size of the *rootn* term and exponentially increase the size of the supercompiled version of *rootn*.

Extreme bloat along these lines is by no means an unusual phenomenon, and the bloat not only wastes hard disk and code cache space, but also leads to the related problem of extremely long supercompilation run times, which can quickly exceed what is practically computable.

Post-hoc code size bounding techniques Jonsson [2011] which examine the output of supercompilation to guess if the code size increase from supercompilation (if any) is "worth it" may be part of the solution. However, it is also essential to find principled ways to avoid producing such large output programs at all, in order to reduce the time required to optimise a program with supercompilation and hence make it more practical as a compiler optimisation pass. Our implementation incorporates some mechanisms to avoid the worst excesses of code explosion (Section E.4), but they are distinctly ad-hoc.

This is an area which needs considerable work, and it is inevitable that any solution will involve heuristics which sometimes pessimise programs even when supercompiling normally would not have led to code bloat.

[1] In fact, our supercompiler is not able to do this because the tag-bag termination test is too weak. However, we can show similar behaviour by manually unrolling the *sequence* recursion and replacing each occurrence of *bs* with an explicit list [*True*, *False*]

An alternative approach would be to compose a suitable supercompiler with a just-in-time compilation algorithm that lazily consumes the syntax tree returned by the supercompiler. By taking this approach, it would be possible to avoid performing supercompilation for a term until it is actually required at runtime, so we would avoid exploring the exponential-time worst cases unless the program being run would itself require exponential time to complete. The main problems with this approach are the relative complexity of just-in-time compilation, and the fact that it would make it difficult or impossible to apply other optimisations (such as strictness analysis) to the result of supercompilation itself.

9.4 Predictable supercompiation

It is very difficult even for those with considerable experience with supercompilation implementations to predict what the result of supercompiling a program will be. This problem fundamentally arises not only from the large state space that supercompilation explores, but also the non-intuitive nature of the well-quasi-order based termination condition that is applied to bound that search.

It would be interesting to devise a supercompilation algorithm which trades away factors such as the amount of optimisation that is achieved in exchange for predictability of the algorithm. As part of this work, it might be fruitful to revisit offline termination tests (such as binding time analysis) whose results are more immediately explicable to the user of the transformation system. Another interesting approach is to only generalise when a user's annotation explicitly tells you to do so, rather than making use of any automatic termination test at all. Of course, such a system would necessarily invite non-termination, but it could be useful in practice when combined with sufficiently good debugging tools that the programmer can use to discover the source of specialiser non-termination.

Better tools for understanding the supercompilation process tree could be another path to predictablity. Existing approaches to debugging supercompilation rely on such primitive methods as inspecting a textual trace showing the states that are being supercompiled. A GUI which helps manage the huge volume of information generated during a typical supercompilation run would be extremely useful.

9.5 Reduction before match

An unfortunate feature of our supercompiler is that for a program such as the following, supercompilation will peel the call to *map* once:

```
let map f xs = case xs of [ ]      → [ ]
                          (y : ys) → f y : map f ys
in map
```

The reason for this is that the state $\mathcal{S}_1$ that will be recursively supercompiled is based on the code underneath the *map* λ, and as such looks like:

```
let map f xs = . . .
in case xs of [ ]      → [ ]
              (y : ys) → f y : map f ys
```

However, when we come to recursively supercompile from the $y : ys$ branch of the residual **case** expression, the state $\mathcal{S}_2$ we supercompile looks like:

```
let map f xs = ... in map f ys
```

Notice that this state will reduce into a state which is equivalent (up to renaming) to the earlier one, but the memoiser will be unable to detect this fact as it only performs exact syntactic matching. As a result, the optimised program will be:

```
let h0 = let map f xs = h1 f xs in map
    h1 f xs = case xs of [ ]         → [ ]
                         (y0 : ys0)  → f y0 : h2 f ys0
    h2 f ys0 = case ys0 of [ ]        → [ ]
                           (y1 : ys1) → f y1 : h2 f ys1
in h0
```

Note that the *map* loop has essentially been peeled once. One obvious approach to ameliorate this is to *reduce* the states to be *match*ed in the memoiser, exploiting the fact that *reduce* will generally be more normalising than a simple call to *normalise*. With this change, we will indeed be able to tieback the $\mathcal{S}_2$ state to the existing promise for $\mathcal{S}_1$.

However, with this modified memoiser, the supercompiler correctness argument no longer goes through. The reason for this is that we may end up matching a later state against an earlier one which is strictly slower than it before reduction, even though it appears to be just as fast after using *reduce*.

For example, we could have an earlier state **let** *f x* = *id True* **in** *f x* which fails the termination test and is *split*, so that we recursively drive the state *id True*. These two states appear to have the same meaning after *reduce*, but we do not want to return the optimised program **let** *h0 x* = **let** *f x* = *h0 x* **in** *f x* **in** *h0 x*!

It is desirable to have some way of avoiding this loop-peeling phenomenon, and the proposed strategy of using *reduce* before *match* seems promising, but more work is required to avoid correctness issues.

9.6 Instance matching

In Section 5.2 we defined the *match* function used by the memoiser. This function only reports *exact* matches between states: you might wonder what would happen if you defined a memoiser which instead used the companion function *instanceMatch* to tie back even if could only detect an instance match against a previous state.

Clearly, this feature can severely impair optimisation if it is uncontrolled. For example, if we supercompile a call *map f xs* first, then the danger is that the supercompilation of a later call *map f* (*map g xs*) will detect an instance match and so reuse the code for the earlier *map f xs* promise, which prevents deforestation of the *map f*.*map g* composition!

Nonetheless, some supercompilers do make restricted use of instance matching. For example, Jonsson and Nordlander [2010] allow instance matches against promises which are ancestors of the current node in the process tree (equivalently, unfulfilled promises). The theoretical justification for this is that parents of the current node will be present in the current termination history, and if the termination test is implemented with a homeomorphic embedding (Section D.6.2) then supercompilation of any instance of those ancestors will immediately cause the termination test to fail. Although this argument does not apply in a supercompiler using tag-bags to implement the termination test, this might still be a useful heuristic for allowing instance matches against ancestor promises.

Another possibility is to allow instance matching against non-ancestor promises *if the variables we will instantiate have previously been generalised.* As an example, consider supercompiling the following initial state with a generalising supercompiler:

$$\begin{array}{lll}
\mathbf{let}\ \mathit{foldl}\ c\ n\ \mathit{xs} = (\mathbf{case}\ \mathit{xs}^{t_1}\ \mathbf{of} & [\,] & \to n \\
 & (y : \mathit{ys}) & \to \mathbf{let}\ m = (c\ n\ y)^{t_4} \\
 & & \quad\ \ \mathbf{in}\ (((\mathit{foldl}^{t_2}\ c)^{t_3}\ m)^{t_5}\ \mathit{ys})^{t_6})^{t_7} \\
\quad \mathit{n0} = \mathit{True} & & \\
\quad f\ \mathit{xs} = \mathit{foldl}\ (\wedge)\ \mathit{n0}\ \mathit{xs} & & \\
\quad \mathit{n1} = \mathit{False} & & \\
\quad g\ \mathit{xs} = \mathit{foldl}\ (\wedge)\ \mathit{n1}\ \mathit{xs} & & \\
\mathbf{in}\ (f, g) & &
\end{array}$$

Two states will be recursively supercompiled (note that we explicitly write the kinding and typing environments):

$$\begin{array}{rcl}
\mathcal{S}_1 & = & \langle \mathit{foldl} \mapsto \ldots, \mathit{n0} \mapsto \mathit{True} \,|\, \mathit{foldl}\ (\wedge)\ \mathit{n0}\ \mathit{xs} \,|\, \epsilon \rangle_{\epsilon | \mathit{xs} : [\mathit{Bool}]} \\
\mathcal{S}_2 & = & \langle \mathit{foldl} \mapsto \ldots, \mathit{n1} \mapsto \mathit{False} \,|\, \mathit{foldl}\ (\wedge)\ \mathit{n0}\ \mathit{xs} \,|\, \epsilon \rangle_{\epsilon | \mathit{xs} : [\mathit{Bool}]}
\end{array}$$

After driving and splitting $\mathcal{S}_1$ the following state will be recursively supercompiled:

$$\mathcal{S}_3 = \langle \mathit{foldl} \mapsto \ldots, \mathit{n0} \mapsto \mathit{True}, \mathit{m0} \mapsto (\wedge)\ \mathit{n0}\ \mathit{y0} \,|\, \mathit{foldl}\ (\wedge)\ \mathit{m0}\ \mathit{ys0} \,|\, \epsilon \rangle_{\epsilon | \mathit{y0} : \mathit{Bool}, \mathit{ys0} : [\mathit{Bool}]}$$

Which in turn leads to driving:

$$\mathcal{S}_4 = \left\langle \begin{array}{l} \mathit{foldl} \mapsto \ldots, \mathit{n0} \mapsto \mathit{True}, \\ \mathit{m0} \mapsto (\wedge)\ \mathit{n0}\ \mathit{y0}, \mathit{m1} \mapsto (\wedge)\ \mathit{m0}\ \mathit{y1} \end{array} \middle| \mathit{foldl}\ (\wedge)\ \mathit{m1}\ \mathit{ys1} \middle| \epsilon \right\rangle_{\epsilon | \mathit{y0} : \mathit{Bool}, \mathit{y1} : \mathit{Bool}, \mathit{ys1} : [\mathit{Bool}]}$$

At this point, the termination test will fail and so (assuming the supercompiler does incorporate rollback) the generalisation heuristics will kick in and cause the *m1* heap binding to be generalised away. As a result we could recursively supercompile:

$$\mathcal{S}_5 = \langle \mathit{foldl} \mapsto \ldots \,|\, \mathit{foldl}\ (\wedge)\ \mathit{m1}\ \mathit{ys1} \,|\, \epsilon \rangle_{\epsilon | \overset{\text{gen}}{\mathit{m1}} : \mathit{Bool}, \mathit{ys1} : [\mathit{Bool}]}$$

Note the hypothetical syntax in the typing environment used to mark the fact that the *m1* variable became free due to generalisation. We will recursively supercompile:

$$\mathcal{S}_6 = \langle \mathit{foldl} \mapsto \ldots, \mathit{m2} \mapsto (\wedge)\ \mathit{m1}\ \mathit{y2} \,|\, \mathit{foldl}\ (\wedge)\ \mathit{m1}\ \mathit{ys2} \,|\, \epsilon \rangle_{\epsilon | \overset{\text{gen}}{\mathit{m1}} : \mathit{Bool}, \mathit{y2} : \mathit{Bool}, \mathit{ys2} : [\mathit{Bool}]}$$

Because $\mathcal{S}_6$ is an instance of $\mathcal{S}_5$ (via the variable *m1*) and *m1* is marked with the "gen" flag, by the proposed instance-matching heuristic we could tie back to $\mathcal{S}_5$ at this point. Turning our attention to the pending state $\mathcal{S}_2$, note that it is also an instance of $\mathcal{S}_5$ via the *m1* variable, so we can also discharge this promise by an instance match. The final code would therefore be:

$$\begin{array}{lll}
\mathbf{let}\ \mathit{h0} = \mathbf{let}\ f\ \mathit{xs} = \mathit{h1}\ \mathit{xs} & & \\
\qquad\qquad\qquad g\ \mathit{xs} = \mathit{h2}\ \mathit{xs} & & \\
\qquad\qquad \mathbf{in}\ (f, g) & & \\
\quad \mathit{h1}\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of} & [\,] & \to \mathit{True} \\
 & (\mathit{y0} : \mathit{ys0}) & \to \mathbf{let}\ \mathit{n0} = \mathit{True} \\
 & & \qquad\quad \mathit{m0} = (\wedge)\ \mathit{n0}\ \mathit{y0} \\
 & & \qquad\quad \mathit{m1} = (\wedge)\ \mathit{m0}\ \mathit{y1} \\
 & & \quad\ \ \mathbf{in}\ \mathit{h5}\ \mathit{m1}\ \mathit{ys1} \\
\quad \mathit{h5}\ \mathit{m1}\ \mathit{ys1} = \mathbf{case}\ \mathit{ys1}\ \mathbf{of} & [\,] & \to \mathit{m1} \\
 & (\mathit{y2} : \mathit{ys2}) & \to \mathit{h6}\ \mathit{m1}\ \mathit{y1}\ \mathit{ys2} \\
\quad \mathit{h6}\ \mathit{m1}\ \mathit{y1}\ \mathit{ys2} = \mathbf{let}\ \mathit{m2} = (\wedge)\ \mathit{m1}\ \mathit{y1} & &
\end{array}$$

```
                              in h5 m2 ys2
        h2 xs = let n0 = False
                       in h5 n0 xs
      in h0
```

This kind of idea seems like a promising basis for a strategy that reduces supercompilation time without sacrificing essential optimisations. However, the strategy still prevents us from achieving useful optimisation in certain cases, and so further work is required to refine it and gather experimental evidence for its effectiveness (or lack thereof).

9.7 Parametricity

Because Core is a typed language, it is possible to exploit the notion of parametricity and free theorems [Walder, 1989] to eagerly generalise away useless information. For example, if we have the following state:

let *x1* $:: \alpha$ = *g x0 y* **in** *foldl g x1 xs*

Because the *x1* heap binding has a type α which is a simple type variable, by parametricity we immediately know that the evaluation of the call *foldl g x1 xs* cannot benefit from knowing the definition of *x1*. As a result, it would not impede optimisation (but may speed up supercompilation) to "eagerly" generalise this term even before the termination test has failed, and drive *foldl g x1 xs* instead.

Likewise, it would not impede optimisation if the memoiser was allowed to tie back even if it only had an instance (rather than exact) match where the instantiating heap bindings all have types which are simple type variables (or applications to type variables of the form $\alpha\ \overline{\tau}$).

Our implementation does not exploit these observations because GHC's core language actually includes a mechanism known as coercions (Section E.3.7) which are used to implement features such as generalised algebraic data types (GADTs) [Xi et al., 2003] and type-level functions [Chakravarty et al., 2005]. Coercions and GADTs allow apparently polymorphic types to be refined by value-level information, which means that these observations no longer hold. Nonetheless, it may prove useful to explore how useful they are in supercompilers for languages which use simple parametric polymorphism.

9.8 Conclusions

This thesis has focused on the issues surrounding supercompilation for a call-by-need language with unrestricted **letrec**, similar to Haskell. The main result of the thesis is a supercompiler which is the first in the world capable of supercompiling and deforesting programs written in these languages, achieving optimisation without the risk of work duplication. Nonetheless, the supercompiler still suffers from the same code (and compile time) explosion problems which plague the field as a whole, and these problems seem to be particularly acute on examples involving **letrec** (such as the *wheel* − *sieve* benchmarks of Chapter 7). Worse, the complexity of implementing supercompilation in the context of GHC, a real-world compiler of considerable complexity, means that although we have considerable faith in the quality of our research contribution, our realisation of that contribution as software still contains at least one major bug, as demonstrated by the miscompilation of *queens* described in Section 7.2.

We believe that the supercompiler of this thesis is a solid basis for further work in the field of call-by-need supercompilation. In particular, we believe that our core contributions of algorithm structure (Chapter 3), call-by-need splitting (Chapter 4) and call-by-need matching and generalisation (Chapter 5) are all solid local maxima in the design space.

Some of our contributions are more peripheral and may prove to be dead ends in the supercompilation research program: in particular, the evidence for the efficacy of our attempts to improve the degree of optimisation a supercompiler can achieve (Chapter 6) is mixed.

Appendix A

Proof of normalisation

Theorem 2.2.1. *The operational semantics of Figure 2.3 without* BETA *and* TYBETA *is normalising.*

Proof. The proof proceeds by defining a measure on states that maps each state to a natural number, and then showing that each of the proposed rules of the operational semantics strictly decreases this measure. Because this relation is well-founded on natural numbers, this shows that the proposed ruleset is normalising.

The measure on states $\|\langle H \,|\, d \,|\, K \rangle\| \in \mathbb{N}$ is defined as follows:

$$\begin{array}{rcl}
\|\langle H \,|\, d \,|\, K \rangle\| & = & \|H\|_H\,(\mathit{values}(H)) + \|d\|_{\hat{d}}\,(\mathit{values}(H)) + \|K\|_K \\
\mathit{values}(H) & = & \{x \mid x{:}\tau \mapsto v^t \in H\}
\end{array}$$

$$\|e^t\|_{\hat{d}}\,(X) \;=\; \begin{cases} 0 & e \equiv x \wedge x \in X \\ \|e\|_e & \text{otherwise} \end{cases} \qquad\qquad \|e^t\|_d \;=\; \|e\|_e$$

$$\begin{array}{rcl}
\|\epsilon\|_H\,(X) & = & 0 \\
\|H, x{:}\tau \mapsto d\|_H\,(X) & = & \|H\|_H(X) + \|d\|_{\hat{d}}\,(X)
\end{array}
\qquad
\begin{array}{rcl}
\|\epsilon\|_K & = & 0 \\
\|\kappa^t, K\|_K & = & \|\kappa\|_\kappa + \|K\|_K
\end{array}$$

$$\begin{array}{rcl}
\|\mathbf{update}\ x\|_\kappa & = & 1 \\
\|\bullet\ x\|_\kappa & = & 1 \\
\|\mathbf{case} \bullet \mathbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau}\ \to\ d}\|_\kappa & = & 1 + \sum \overline{\|d\|_d} \\
\|\bullet\ \tau\|_\kappa & = & 1
\end{array}$$

$$\begin{array}{rcl}
\|x\|_e & = & 2 \\
\|\lambda x{:}\tau.\, d\|_e & = & \|d\|_d \\
\|d\ x\|_e & = & 2 + \|d\|_d \\
\|\mathbf{C}\ \overline{\tau}\ \overline{x}\|_e & = & 0 \\
\|\mathbf{case}\ d\ \mathbf{of}\ \overline{\mathbf{C}\ \overline{\alpha{:}\kappa}\ \overline{x{:}\tau}\ \to\ d_{\mathbf{C}}}\|_e & = & 2 + \|d\|_d + \sum \overline{\|d_{\mathbf{C}}\|_d} \\
\|\mathbf{let}\ \overline{x\ =\ d_x}\ \mathbf{in}\ d\|_e & = & 1 + \sum \overline{\|d_x\|_d} + \|d\|_d \\
\|\Lambda\alpha{:}\kappa.\, d\|_e & = & \|d\|_d \\
\|d\ \tau\|_e & = & 2 + \|d\|_d
\end{array}$$

For all the normalising rules except VAR, UPDATEV and UPDATE it is easy to show that the measure is strictly decreased by the rule (it is useful to observe in the LETREC case that for all d and X, $\|d\|_{\hat{d}}(X) \le \|d\|_d$). We consider the more complicated cases of VAR, UPDATEV and UPDATE explicitly.

Case VAR By this rule, $\langle H, x{:}\tau \mapsto d \,|\, x^t \,|\, K\rangle \rightsquigarrow \langle H \,|\, d \,|\, \textbf{update } x{:}\tau^t, K\rangle$, where d is not a value. Let $X = \mathit{values}(H, x{:}\tau \mapsto d)$ and $X' = \mathit{values}(H)$. Note that $x \notin X$, so we have that:

$$\begin{array}{rcl} \|\langle H, x{:}\tau \mapsto d \,|\, x^t \,|\, K\rangle\| & = & \|H\|_H\,(X) + \|d\|_{\hat{d}}(X) + 2 + \|K\|_K \\ \|\langle H \,|\, d \,|\, \textbf{update } x{:}\tau^t, K\rangle\| & = & \|H\|_H\,(X') + \|d\|_{\hat{d}}(X') + 1 + \|K\|_K \end{array}$$

By the definition of *values*, $\forall y . y \in X \implies y \in X'$ and so by inspection of the definition of $\|\|_{\hat{d}}$ we can see that $\|H\|_H\,(X') \leq \|H\|_H\,(X)$ and $\|d\|_{\hat{d}}(X') \leq \|d\|_{\hat{d}}(X)$. This establishes that $\|\langle H \,|\, d \,|\, \textbf{update } x{:}\tau^t, K\rangle\| < \|\langle H, x{:}\tau \mapsto d \,|\, x^t \,|\, K\rangle\|$, as required.

Case UPDATEV According to this rule,

$$\langle H \,|\, u \,|\, \textbf{update } x{:}\tau^{t_x}, K\rangle \rightsquigarrow \langle H, x{:}\tau \mapsto u \,|\, x^{t_x} \,|\, K\rangle$$

Let $X = \mathit{values}(H)$ and $X' = \mathit{values}(H, x{:}\tau \mapsto u)$. We have that:

$$\begin{array}{rcl} \|\langle H \,|\, u \,|\, \textbf{update } x{:}\tau^{t_x}, K\rangle\| & = & \|H\|_H(X) + \|u\|_{\hat{d}}(X) + 1 + \|K\|_K \\ \|\langle H, x{:}\tau \mapsto u \,|\, x^{t_x} \,|\, K\rangle\| & = & \|H\|_H(X') + \|u\|_{\hat{d}}(X') + \|K\|_K \end{array}$$

By a similar argument to before, $\|H\|_H\,(X') \leq \|H\|_H\,(X)$. We can also see that by the definition of $\|\|_{\hat{d}}$, $\|u\|_{\hat{d}}(X) = \|u\|_{\hat{d}}(X')$, which establishes that $\|\langle H, x{:}\tau \mapsto u \,|\, x^{t_x} \,|\, K\rangle\| < \langle H \,|\, u \,|\, \textbf{update } x{:}\tau^{t_x}, K\rangle$ as required.

Case UPDATE According to this rule,

$$\langle H[x{:}\tau \mapsto u] \,|\, x^{t_x} \,|\, \textbf{update } y{:}\tau^{t_y}, K\rangle \rightsquigarrow \langle H, y{:}\tau \mapsto x^{t_y} \,|\, x^{t_x} \,|\, K\rangle$$

Let $X = \mathit{values}(H)$ and $X' = \mathit{values}(H, y{:}\tau \mapsto x^{t_y})$. We have that:

$$\begin{array}{rcl} \|\langle H[x{:}\tau \mapsto u] \,|\, x^{t_x} \,|\, \textbf{update } y{:}\tau^{t_y}, K\rangle\| & = & \|H\|_H(X) + \|x^{t_x}\|_{\hat{d}}(X) + 1 + \|K\|_K \\ \|\langle H, y{:}\tau \mapsto x^{t_y} \,|\, x^{t_x} \,|\, K\rangle\| & = & \|H\|_H(X') + \|x^{t_y}\|_{\hat{d}}(X') + \|x^{t_x}\|_{\hat{d}}(X') + \|K\|_K \end{array}$$

We know that $x \in X$ and $x \in X'$ so therefore $\|x^{t_x}\|_{\hat{d}}(X) = \|x^{t_y}\|_{\hat{d}}(X') = \|x^{t_x}\|_{\hat{d}}(X') = 0$. This is enough to establish $\|\langle H, y{:}\tau \mapsto x^{t_y} \,|\, x^{t_x} \,|\, K\rangle\| < \|\langle H[x{:}\tau \mapsto u] \,|\, x^{t_x} \,|\, \textbf{update } y{:}\tau^{t_y}, K\rangle\|$ as required.

□

Appendix B

Proof of type generalisation

The syntax and static semantics of System Fω are reproduced in Figure B.1 and Figure B.2, respectively. To conduct our proof, we will require type substitutions, which we define along with their operations in Figure B.3. For our purposes, all substitutions are exhaustive and non-capturing. Note that in the following all type equality will be considered up to α-equivalence, and we will often silently α-convert during our proofs.

The reason that the non-coupling substitutions of Section 5.7 are interesting is that they allow us to prove that two terms are equal given only that they are equal *after* the non-coupling substitutions have been applied. This is codified in the following lemma:

Lemma B.0.1 (Non-coupling equality). *If θ_0 and θ_1 are non-coupling then $\tau\theta_0 = \upsilon\theta_0$ and $\tau\theta_1 = \upsilon\theta_1$ iff $\tau = \upsilon$.*

Proof. Consider the $(\tau = \upsilon) \implies \ldots$ direction first. This direction follows trivially because $\tau\theta_0 = \tau\theta_0$ and symmetrically for θ_1.

Now consider the other direction. We proceed by simultaneous induction on the structures on τ and υ.

The first case we consider is $\tau = \alpha$, $\upsilon = \beta$. By the assumption we know that $\alpha\theta_0 = \beta\theta_0$ and $\alpha\theta_1 = \beta\theta_1$. Note that if $\alpha = \beta$ then the non-coupling substitution property would be violated because we would have two distinct variables α and β mapping to α-equivalent pairs of types. Thus, $\alpha = \beta$, satisfying the goal of this case.

The next case we consider is $\tau = \alpha$, $\upsilon \neq \beta$. By the assumption, $\alpha\theta_0 = \upsilon\theta_0$ and $\alpha\theta_1 = \upsilon\theta_1$. To discharge this case, it is sufficient to prove *coupled* $(\upsilon\theta_0, \upsilon\theta_1)$ because doing so would contradict the non-coupling assumption for the variable α. We prove this by case analysis on υ:

- Case $\upsilon = (\to)$: *coupled* $((\to)\theta_0, (\to)\theta_1) =$ *coupled* $((\to), (\to))$ follows immediately.

- Case $\upsilon = \upsilon_1\ \upsilon_2$: *coupled* $((\upsilon_1\ \upsilon_2)\,\theta_0, (\upsilon_1\ \upsilon_2)\,\theta_1) =$ *coupled* $(\upsilon_1\theta_0\ \upsilon_2\theta_0, \upsilon_1\theta_1\ \upsilon_2\theta_1)$ follows immediately.

- Case $\forall\beta{:}\kappa.\,\hat{\upsilon}$: *coupled* $((\forall\beta{:}\kappa.\,\hat{\upsilon})\,\theta_0, (\forall\beta{:}\kappa.\,\hat{\upsilon})\,\theta_1) =$ *coupled*$(\forall\beta{:}\kappa.\,\hat{\upsilon}\,(\theta_0, \beta{:}\kappa \mapsto \beta)\,, \forall\beta{:}\kappa.\,\hat{\upsilon}\,(\theta_1, \beta{:}\kappa \mapsto \beta))$ follows immediately.

The case $\tau \neq \alpha$, $\upsilon = \beta$ can be discharged by a symmetric version of the previous case's argument. This leaves only cases involving types of the form $(\to)$, $\hat{\tau}\ \hat{\upsilon}$ and $\forall\alpha{:}\kappa.\,\hat{\tau}$. Note that since neither τ nor υ can be a variable, the only way our assumption $\tau\theta_0 = \upsilon\theta_0$ can be true is if the (outermost) forms of τ and υ match *before* substitution. Thus we only need consider three more cases: one for each possible form. As we will see, each case can be discharged by making use of the inductive hypothesis.

The easiest remaining case is $\tau = (\to)$, $\upsilon = (\to)$, which follows trivially.

Type Variables α, β **Term Variables** x, y, z

Kinds

$$\begin{array}{lcll} \kappa & ::= & \star & \text{Kind of term types} \\ & \mid & \kappa \to \kappa & \text{Kind of type constructors} \end{array}$$

Types

$$\begin{array}{lcll} \tau & ::= & \alpha & \text{Type variable} \\ & \mid & (\to) & \text{Function type constructor} \\ & \mid & \tau\ \tau & \text{Type application} \\ & \mid & \forall \alpha{:}\kappa.\, \tau & \text{Parametric polymorphism} \end{array}$$

Terms

$$\begin{array}{lcll} e & ::= & x & \text{Term variable} \\ & \mid & \lambda x{:}\tau.\, e & \text{Term abstraction} \\ & \mid & e\ e & \text{Term application} \\ & \mid & \Lambda \alpha{:}\kappa.\, e & \text{Type abstraction} \\ & \mid & e\ \tau & \text{Type application} \end{array}$$

Figure B.1: Syntax of System Fω

Kinding context $\Sigma ::= \overline{\alpha{:}\kappa}$ **Typing context** $\Gamma ::= \overline{x{:}\tau}$

$$\boxed{\Sigma \vdash^{\kappa} \tau \ :\ \kappa}$$

$$\frac{\alpha{:}\kappa \in \Sigma}{\Sigma \vdash^{\kappa} \alpha \ :\ \kappa}\ \textsc{TyVar} \qquad \frac{}{\Sigma \vdash^{\kappa} (\to) \ :\ \star \to \star \to \star}\ \textsc{TyFun}$$

$$\frac{\Sigma \vdash^{\kappa} \tau \ :\ \iota \to \kappa \quad \Sigma \vdash^{\kappa} \upsilon \ :\ \iota}{\Sigma \vdash^{\kappa} \tau\ \upsilon \ :\ \kappa}\ \textsc{TyApp} \qquad \frac{\Sigma, \alpha{:}\kappa \vdash^{\kappa} \tau \ :\ \star}{\Sigma \vdash^{\kappa} \alpha{:}\kappa.\tau \ :\ \star}\ \textsc{TyForall}$$

$$\boxed{\Sigma|\Gamma \vdash e \ :\ \tau}$$

$$\frac{x{:}\tau \in \Gamma}{\Sigma|\Gamma \vdash x \ :\ \tau}\ \textsc{Var}$$

$$\frac{\Sigma|\Gamma, x{:}\upsilon \vdash e \ :\ \tau \quad \Sigma \vdash^{\kappa} \upsilon \ :\ \star}{\Sigma|\Gamma \vdash \lambda x{:}\upsilon.\, e \ :\ \upsilon \to \tau}\ \textsc{Lam} \qquad \frac{\Sigma|\Gamma \vdash e_1 \ :\ \upsilon \to \tau \quad \Sigma|\Gamma \vdash e_2 \ :\ \upsilon}{\Sigma|\Gamma \vdash e_1\ e_2 \ :\ \tau}\ \textsc{App}$$

$$\frac{\Sigma, \alpha{:}\kappa|\Gamma \vdash e \ :\ \tau}{\Sigma|\Gamma \vdash \Lambda \alpha{:}\kappa.\, e \ :\ \forall \alpha{:}\kappa.\tau}\ \textsc{TyLam} \qquad \frac{\Sigma|\Gamma \vdash e_1 \ :\ \forall \alpha{:}\kappa.\tau \quad \Sigma \vdash^{\kappa} \upsilon \ :\ \kappa}{\Sigma|\Gamma \vdash e_1\ \upsilon \ :\ \tau[\upsilon/\alpha]}\ \textsc{TyApp}$$

Figure B.2: Type system of System Fω

We next consider $\tau = \tau_1\ \tau_2$, $\upsilon = \upsilon_1\ \upsilon_2$. By the assumption, $(\tau_1\ \tau_2)\,\theta_0 = \tau_1\theta_0\ \tau_2\theta_0 = \upsilon_1\theta_0\ \upsilon_2\theta_0 = (\upsilon_1\ \upsilon_2)\,\theta_0$ (and symmetrically for θ_1). By the definition of α-equivalence, $\tau_1\theta_0 = \upsilon_1\theta_0$ and $\tau_2\theta_0 = \upsilon_2\theta_0$ (and symmetrically for θ_1). Therefore by induction $\tau_1 = \upsilon_1$ and $\tau_2 = \upsilon_2$, which is sufficient to discharge this case.

Our final case is $\tau = \forall \alpha : \kappa.\, \hat{\tau}$, $\upsilon = \forall \alpha : \kappa.\, \hat{\upsilon}$. By the assumption, $(\forall \alpha{:}\kappa.\, \hat{\tau})\,\theta_0 = \forall \alpha : \kappa.\, \hat{\tau}\,(\theta_0, \alpha{:}\kappa \mapsto \alpha) = \forall \alpha : \kappa.\, \hat{\upsilon}\,(\theta_0, \alpha{:}\kappa \mapsto \alpha) = (\forall \alpha{:}\kappa.\, \hat{\upsilon})\,\theta_0$ (and symmetrically for θ_1). By the definition of α-equivalence, $\hat{\tau}\,(\theta_0, \alpha{:}\kappa \mapsto \alpha) = \hat{\upsilon}\,(\theta_0, \alpha{:}\kappa \mapsto \alpha)$ (and symmetrically for θ_1). Because α is fresh, our extended substitutions $(\theta_0, \alpha{:}\kappa \mapsto \alpha)$ and $(\theta_1, \alpha{:}\kappa \mapsto \alpha)$ are still non-coupling, and thus by induction we have $\hat{\tau} = \hat{\upsilon}$, completing the case and the proof. □

Type substitution $\theta ::= \overline{\alpha{:}\kappa \mapsto \tau}$

$$\boxed{\tau\theta = \upsilon}$$

$$\begin{array}{rcl}
\alpha\,(\theta[\alpha{:}\kappa \mapsto \tau]) & = & \tau \\
(\rightarrow)\theta & = & (\rightarrow) \\
(\tau\ \upsilon)\,\theta & = & \tau\theta\ \upsilon\theta \\
(\forall\alpha{:}\kappa.\,\tau)\,\theta & = & \forall\alpha{:}\kappa.\,\tau\,(\theta, \alpha{:}\kappa \mapsto \alpha)
\end{array}$$

$$\boxed{e_1\theta = e_2}$$

$$\begin{array}{rcl}
x\theta & = & x \\
(\lambda x{:}\tau.\,e)\,\theta & = & \lambda x{:}\tau\theta.\,e\theta \\
(e_1\ e_2)\,\theta & = & e_1\theta\ e_2\theta \\
(\Lambda\alpha{:}\kappa.\,e)\,\theta & = & \Lambda\alpha{:}\kappa.\,e\,(\theta, \alpha{:}\kappa \mapsto \alpha) \\
(e\ \tau)\,\theta & = & e\theta\ \tau\theta
\end{array}$$

Figure B.3: Type substitution for System Fω

Having established this, we are in a position to define the theorem that justifies the well-typedness of a generalisation that uses non-coupling substitutions:

Theorem 5.7.1. *In System Fω, if we have Σ_0, Σ_1, Σ, Γ_0, Γ_1, Γ, e_0, e_1, e, τ_0, τ_1, θ_0, θ_1 such that:*

- *The ungeneralised terms are well typed: $\Sigma_0|\Gamma_0 \vdash e_0 \;:\; \tau_0$ and $\Sigma_1|\Gamma_1 \vdash e_1 \;:\; \tau_1$*
- *θ_0 and θ_1 are non-coupling with common domain Σ*
- *The term is a generalisation of the originals: $e\theta_0 = e_0$ and $e\theta_1 = e_1$*
- *The type environment is a generalisation of the originals: $\Gamma\theta_0 = \Gamma_0$ and $\Gamma\theta_1 = \Gamma_1$.*

Then there exists a (unique) type τ such that:

- *The generalised term is well typed: $\Sigma|\Gamma \vdash e \;:\; \tau$*
- *The type is a generalisation of the originals: $\tau\theta_0 = \tau_0$ and $\tau\theta_1 = \tau_1$*

Proof. Case $e = x$: by assumptions, $x\theta_0 = x = e_0$ and $x\theta_1 = x = e_1$. We thus know that $\Sigma_0|\Gamma_0 \vdash x \;:\; \tau_0$ and $\Sigma_1|\Gamma_1 \vdash x \;:\; \tau_1$, and hence $x : \tau_0 \in \Gamma_0$ and $x : \tau_1 \in \Gamma_1$. Thus, by the assumption about the type environment, $\exists x : \tau \in \Gamma. \tau\theta_0 = \tau_0 \wedge \tau\theta_1 = \tau_1$. Well-typing is proven since $\Sigma|\Gamma \vdash x \;:\; \tau$.

Case $e = \lambda x{:}\hat{\upsilon}.\,\hat{e}$: by assumptions, $(\lambda x{:}\hat{\upsilon}.\,\hat{e})\,\theta_0 = \lambda x{:}\hat{\upsilon}\theta_0.\,\hat{e}\theta_0 = e_0$, and so $\Sigma_0|\Gamma_0 \vdash \lambda x{:}\hat{\upsilon}\theta_0.\,\hat{e}\theta_0 \;:\; \tau_0$. By inspecting the type rules, it must be the case that $\tau_0 = \hat{\upsilon}\theta_0 \rightarrow \hat{\tau}_0$ and $\Sigma_0|\Gamma_0, x : \hat{\upsilon}\theta_0 \vdash \hat{e}\theta_0 \;:\; \hat{\tau}_0$. The same things hold symmetrically with θ_1, and so we can induct to prove that for some $\hat{\tau}$ satisfying $\hat{\tau}\theta_0 = \hat{\tau}_0 \wedge \hat{\tau}\theta_1 = \hat{\tau}_1$, $\Sigma|\Gamma, x{:}\hat{\upsilon} \vdash \hat{e} \;:\; \hat{\tau}$. Well-typing follows since $\Sigma|\Gamma \vdash \lambda x{:}\hat{\upsilon}.\,\hat{e} \;:\; \hat{\upsilon} \rightarrow \hat{\tau}$.

Case $e = e_1\ e_2$: by assumptions, $(e_1\ e_2)\,\theta_0 = e_1\theta_0\ e_2\theta_0 = e_0$ and so $\Sigma_0|\Gamma_0 \vdash e_1\theta_0\ e_2\theta_0 \;:\; \tau_0$. By inspecting the type rules, we must have that $\Sigma_0|\Gamma_0 \vdash e_1\theta_0 \;:\; \upsilon_0 \rightarrow \tau_0$ and $\Sigma_0|\Gamma_0 \vdash e_2\theta_0 \;:\; \upsilon_0$. The same things hold symmetrically with θ_1, and so we can induct to prove that for some $\hat{\tau}$ and υ, $\Sigma|\Gamma \vdash e_1 \;:\; \hat{\tau}$ and $\Sigma|\Gamma \vdash e_2 \;:\; \upsilon$ with both $\hat{\tau}\theta_0 = \upsilon_0 \rightarrow \tau_0 \wedge \hat{\tau}\theta_1 = \upsilon_1 \rightarrow \tau_1$ and $\upsilon\theta_0 = \upsilon_0 \wedge \upsilon\theta_1 = \upsilon_1$. Now, observe that it must be the case that $\hat{\tau} = \hat{\upsilon} \rightarrow \tau$ for some $\hat{\upsilon}$ and τ. The only other possibility consistent with the equality we have just learnt is that $\hat{\tau} = \alpha$ for some α, but that would violate the non-coupling property of θ_0 and θ_1. Therefore, $(\hat{\upsilon} \rightarrow \tau)\,\theta_0 = \hat{\upsilon}\theta_0 \rightarrow \tau\theta_0 = \upsilon_0 \rightarrow \tau_0$ (and symmetrically for θ_1). This suffices to prove that $\hat{\upsilon} = \upsilon$ and $\tau\theta_0 = \tau_0 \wedge \tau\theta_1 = \tau_1$ and therefore we can complete this case with well-typing following from $\Sigma|\Gamma \vdash e_1\ e_2 \;:\; \tau$.

Case $e = \Lambda\alpha : \kappa.\,\hat{e}$: by assumptions, $(\Lambda\alpha{:}\kappa.\,\hat{e})\,\theta_0 = \Lambda\alpha : \kappa.\,\hat{e}\,(\theta_0, \alpha{:}\kappa \mapsto \alpha) = e_0$ and so $\Sigma_0|\Gamma_0 \vdash \Lambda\alpha : \kappa.\,\hat{e}\,(\theta_0, \alpha{:}\kappa \mapsto \alpha) \;:\; \tau_0$. By inspecting the type rules, we must have that $\Sigma_0, \alpha : \kappa|\Gamma_0 \vdash \hat{e}\,(\theta_0, \alpha{:}\kappa \mapsto \alpha) \;:\; \hat{\tau}_0$ where $\tau_0 = \forall\alpha : \kappa.\,\hat{\tau}_0$. The same things hold symmetrically with θ_1, so we can induct with the extended θ_0 and θ_1 (which are still non-coupling because α is fresh) to show that for some $\hat{\tau}$ we have $\Sigma, \alpha : \kappa|\Gamma \vdash \hat{e} \;:\; \hat{\tau}$. Furthermore, inductively we have that $\hat{\tau}\,(\theta_0, \alpha{:}\kappa \mapsto \alpha) = \hat{\tau}_0 \wedge \hat{\tau}\,(\theta_1, \alpha{:}\kappa \mapsto \alpha) = \hat{\tau}_1$, which implies that $(\forall\alpha{:}\kappa.\,\hat{\tau})\,\theta_0 = \forall\alpha : \kappa.\,\hat{\tau}_0 = \tau_0$ (and symmetrically for θ_1). Well-typing follows since $\Sigma|\Gamma \vdash \Lambda\alpha{:}\kappa.\,\hat{e} \;:\; \forall\alpha{:}\kappa.\,\hat{\tau}$.

Case $e = \hat{e}\ \hat{\tau}$: by assumptions, $(\hat{e}\ \hat{\tau})\,\theta_0 = \hat{e}\theta_0\ \hat{\tau}\theta_0 = e_0$ and so $\Sigma_0|\Gamma_0 \vdash \hat{e}\theta_0\ \hat{\tau}\theta_0 \;:\; \tau_0$. By inspecting the type rules, we must have that $\Sigma_0|\Gamma_0 \vdash \hat{e}\theta_0 \;:\; \forall\alpha{:}\kappa.\,\upsilon_0$, $\Sigma_0 \vdash^{\kappa} \hat{\tau}\theta_0 \;:\; \kappa$ and $\tau_0 = \upsilon_0[\hat{\tau}\theta_0/\alpha]$. The same thing holds symmetrically with θ_1 and so we can induct to prove that for some $\hat{\upsilon}$, $\Sigma|\Gamma \vdash \hat{e} \;:\; \hat{\upsilon}$ where $\hat{\upsilon}\theta_0 = \forall\alpha : \kappa.\,\upsilon_0 \wedge \hat{\upsilon}\theta_1 = \forall\alpha : \kappa.\,\upsilon_1$. Observe that it must be the case that $\hat{\upsilon} = \forall\alpha : \kappa.\,\upsilon$. The only other possibility consistent with the equality we learn by induction is that $\hat{\upsilon} = \beta$ for some β, but if there were the case the non-coupling property of θ_0 and θ_1 would be contradicted. Therefore, $(\forall\alpha{:}\kappa.\,\upsilon)\,\theta_0 = \forall\alpha : \kappa.\,\upsilon\,(\theta_0, \alpha{:}\kappa \mapsto \alpha) = \forall\alpha : \kappa.\,\upsilon_0$ (and symmetrically for θ_1). Well typing follows since clearly $\Sigma \vdash^{\kappa} \hat{\tau} \;:\; \kappa$ and so $\Sigma|\Gamma \vdash \hat{e}\ \hat{\tau} \;:\; \upsilon[\hat{\tau}/\alpha]$. The generalisation property holds because $\tau\theta_0 = \upsilon[\hat{\tau}/\alpha]\theta_0 = \upsilon\,(\theta_0, \alpha{:}\kappa \mapsto \alpha)\,[\hat{\tau}\theta_0/\alpha] = \upsilon_0[\hat{\tau}\theta_0/\alpha] = \tau_0$ (and symmetrically for θ_1).

□

Appendix C

Improvement theory

This appendix continues the discussion of improvement theory begun in Section 2.3, completing the development of the main theorems for our operational semantics.

We begin by stating a fundamental property about the reduction rules for Core (Figure 2.3), which we will make use of throughout this section:

Lemma C.0.2 (Extension). *If* $\langle H \,|\, d \,|\, K \rangle \rightsquigarrow^n \langle H' \,|\, d' \,|\, K' \rangle$ *then for all* $\hat{H}$ *and* $\hat{K}$*,*

$$\left\langle \hat{H}, H \,\middle|\, e \,\middle|\, K, \hat{K} \right\rangle \rightsquigarrow^n \left\langle \hat{H}, H' \,\middle|\, e' \,\middle|\, K', \hat{K} \right\rangle$$

Proof. Follows by observing that none of the reduction rules can be prevented from being applicable by extending the stack at its tail, or by adding additional bindings to the heap. □

C.1 Generalised contexts explored

The principle advantage that generalised contexts (as introduced in Section 2.3) have over standard contexts is that the variables that may be used by the term that is eventually used to fill the hole are explicitly recorded. This ensures that the usual operational semantics for the language in question can be extended smoothly to the reduction of terms containing holes. If we do not explicitly record the variables used by the hole it is difficult to avoid potentially problematic reduction sequences such as this one:

$$\begin{array}{ll}
 & \langle f : (\mathit{Int} \to \mathit{Int}) \mapsto \lambda x : \mathit{Int}.\, [\cdot] \,|\, f\ y \,|\, \epsilon \rangle \\
\rightsquigarrow & \langle \epsilon \,|\, \lambda x : \mathit{Int}.\, [\cdot] \,|\, \textbf{update}\ f : \mathit{Int} \to \mathit{Int}, \bullet\ y \rangle \\
=^{\alpha} & \langle \epsilon \,|\, \lambda y : \mathit{Int}.\, [\cdot] \,|\, \textbf{update}\ f : \mathit{Int} \to \mathit{Int}, \bullet\ y \rangle \quad (\ast) \\
\rightsquigarrow & \langle f : (\mathit{Int} \to \mathit{Int}) \mapsto \lambda y : \mathit{Int}.\, [\cdot] \,|\, [\cdot] \,|\, \epsilon \rangle
\end{array}$$

Note that in the pre-reduction state, filling the hole with the term x gives a state that can be rebuilt to $\textbf{let}\, f = \lambda(x :: \mathit{Int}) \to x\ \textbf{in}\, f\ y$, but after the state has undergone the (erroneous) reduction, filling the hole with x and rebuilding gives $\textbf{let}\, f = \lambda(y :: \mathit{Int}) \to x\ \textbf{in}\ x$, and these two terms are not denotationally equivalent in all closing contexts. This failure of hole-filling to commute with reduction fundamentally stems from the erroneous α-conversion on the line marked $(\ast)$. This α-conversion is invalid because standard contexts are not α-convertible without changing their meaning. In contrast, generalised contexts may be α-converted freely by simply renaming the occurrences of variables applied to a meta-variable as you would any other variable occurrence.

We will also make use of a version of the operational semantics of Figure 2.3, extended to states $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle$ made up of generalised contexts. This extension is mostly obvious, except $\xi \cdot \overline{x}$ should not be considered to be a non-value for the purposes of deciding whether the VAR rule can be used (so $\langle H, x{:}\tau \mapsto \xi \cdot \overline{x} \,|\, x \,|\, K \rangle$ is stuck). The normalisation result of Theorem 2.2.1 from the standard semantics carries over mostly unchanged, except that:

- Because of the addition of $\xi{\cdot}\overline{x}$ to the syntax, there is an additional form of normalised state: $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle$, where $\mathit{deref}(\mathbb{H}, \mathbb{C}) = \xi \cdot \overline{x}$
- The issue arises of what weight to assign to $\xi \cdot \overline{x}$ when extending the normalisation measure to contexts. Our approach is to have $\|\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle\|$ return a *triple* where the components are:

  1. The number of syntactic occurrences of the meta-variable ξ
  2. $\|\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle [(\overline{x}).\, y]\|$, where y is an arbitrary variable which is not bound in $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle$
  3. $\|\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle [(\overline{x}).\, \mathbf{C}]\|$, where $\mathbf{C}$ is an arbitrary data constructor

  These triples (n, o, p) are not compared by usual lexical order but instead by the well-founded ordering defined by $(n', o', p') < (n, o, p) \iff (n' \leq n) \wedge (o' < o) \wedge (p' < p)$.

  It is straightforward to show that the normalisation result of Theorem 2.2.1 still holds since the normalising reduction rules *never* duplicate occurrences of meta-variables (although they may remove occurrences) and so the measure on contexts is still strictly decreased by application of the normalising rules.

This choice of the measure on states allows us to show a useful lemma relating the measure on contexts and simple states:

Lemma C.1.1. *If* $\|\langle \mathbb{H}' \,|\, \mathbb{C}' \,|\, \mathbb{K}' \rangle\| < \|\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle\|$ *then for all hole-fillers* $\overline{x}$, e *we have* $\|\langle \mathbb{H}' \,|\, \mathbb{C}' \,|\, \mathbb{K}' \rangle [(\overline{x}).\, e]\| < \|\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle [(\overline{x}).\, e]\|$

Proof. By inspection of the definition of the measures, observing that:

- Measure-decrease on contexts means that the number of occurrences of the meta-variables has either stayed the same or lessened, and so substitution of an arbitrary term for those holes will not contribute more weight to the "smaller" context than it does to the "larger" context.
- Whether e is a value or a non-value, one of the two state measure-decreases witnessed by $\|\langle \mathbb{H}' \,|\, \mathbb{C}' \,|\, \mathbb{K}' \rangle\| < \|\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle\|$ will suffice to show that the hole-filled state will also experience measure-decrease.

□

Many useful lemmas about the standard operational semantics, such as Lemma C.0.2, carry over in the obvious way to the operational semantics on contexts.

C.2 Basic lemmas

In this section we will develop the basic lemmas of improvement theory: open uniform computation and a context lemma. These will be necessary to derive the interesting theorems of improvement theory in the next section.

Lemma C.2.1 (Open Uniform Computation)**.** *Given some* $\mathbb{H}$, $\mathbb{C}$ *and* $\mathbb{K}$, *if there exist some* $\hat{\mathbb{H}}$, $\hat{\mathbb{K}}$ *and* $(\overline{x}).\, e$ *such that* $\left\langle \hat{\mathbb{H}}, \mathbb{H} \middle| \mathbb{C} \middle| \mathbb{K}, \hat{\mathbb{K}} \right\rangle [(\overline{x}).\, e] \Downarrow$ *then* $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle$ *reduces to a state context of one of the following forms:*

1. $\langle \mathbb{H}' \,|\, \mathbb{C}' \,|\, \epsilon \rangle$, *deref*$(\mathbb{H}', \mathbb{C}') = \mathbb{V}'$

2. $\langle \mathbb{H}' \,|\, \mathbb{C}' \,|\, \mathbb{K}' \rangle$, *deref*$(\mathbb{H}', \mathbb{C}') = \xi \cdot \overline{y}$

3. $\langle \mathbb{H}' \,|\, x \,|\, \mathbb{K}' \rangle$, $x \in \mathit{bvs}(\hat{\mathbb{H}})$

Proof. We can assume that for some n, $\left\langle \hat{\mathbb{H}}, \mathbb{H} \middle| \mathbb{C} \middle| \mathbb{K}, \hat{\mathbb{K}} \right\rangle [(\overline{x}).\, e] \Downarrow^n$. Proceeding by induction on the pair $(n, \|\left\langle \hat{\mathbb{H}}, \mathbb{H} \middle| \mathbb{C} \middle| \mathbb{K}, \hat{\mathbb{K}} \right\rangle [(\overline{x}).\, e]\|)$, we take the context $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle$ and consider cases on $\mathbb{C}$. For space reasons, we show the proof for illustrative cases only:

Case $\mathbb{C} = \xi \cdot \overline{y}$ This is a type 2 context, so we are done.

Case $\mathbb{C} = x$ We know that the extended state eventually reduces to a value, so x must be bound either in $\mathbb{H}$ or $\hat{\mathbb{H}}$. In the latter case, this is a type 3 context, and we are done. In the former case, $\mathbb{H} = \mathbb{H}_0, x\!:\!\tau \mapsto \mathbb{C}'$. If $\mathbb{C}'$ is $\xi \cdot \overline{y}$ then this is a type 2 context and we are done. Therefore the only cases we need to consider are whether $\mathbb{C}'$ is a value or a non-value.

- If $\mathbb{C}'$ is a non-value we know

  $$\begin{array}{rcl} \langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle & \rightsquigarrow^0 & \langle \mathbb{H}_0 \,|\, \mathbb{C}' \,|\, \textbf{update } x\!:\!\tau, \mathbb{K} \rangle \\ \|\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle\| & > & \|\langle \mathbb{H}_0 \,|\, \mathbb{C}' \,|\, \textbf{update } x\!:\!\tau, \mathbb{K} \rangle\| \end{array}$$

  Therefore, by Lemma C.0.2, $\left\langle \hat{\mathbb{H}}, \mathbb{H}_0 \middle| \mathbb{C}' \middle| \textbf{update } x\!:\!\tau, \mathbb{K}, \hat{\mathbb{K}} \right\rangle [(\overline{x}).\, e] \Downarrow^n$. Furthermore, by Lemma C.1.1 and inspection of the definition of $\|\,\|$, we can see that

  $$\|\left\langle \hat{\mathbb{H}}, \mathbb{H}_0 \middle| \mathbb{C}' \middle| \textbf{update } x\!:\!\tau, \mathbb{K}, \hat{\mathbb{K}} \right\rangle [(\overline{x}).\, e]\| < \|\left\langle \hat{\mathbb{H}}, \mathbb{H} \middle| \mathbb{C} \middle| \mathbb{K}, \hat{\mathbb{K}} \right\rangle [(\overline{x}).\, e]\|$$

  Because of the measure decrease, we can safely apply the inductive hypothesis to show that the reduced context $\langle \mathbb{H}_0 \,|\, \mathbb{C}' \,|\, \textbf{update } x\!:\!\tau, \mathbb{K} \rangle$ eventually reduces to a context of the required form.

- If $\mathbb{C}'$ is a value there are 5 cases to consider depending on the form of $\mathbb{K}$. If $\mathbb{K}$ is ϵ then this as a type 1 context, so we are done. Otherwise, there are 4 possible cases depending on what the uppermost stack frame is. For brevity, we only consider the case where the uppermost stack frame is such that BETA applies.

  In this case $\mathbb{C}' = \lambda x : \upsilon.\, \mathbb{C}''$, $\mathbb{K} = \bullet\ x, \mathbb{K}_0$ and so $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle \rightsquigarrow^1 \langle \mathbb{H} \,|\, \mathbb{C}'' \,|\, \mathbb{K}_0 \rangle$. By Lemma C.0.2, $\left\langle \hat{\mathbb{H}}, \mathbb{H} \middle| \mathbb{C}'' \middle| \mathbb{K}_0, \hat{\mathbb{K}} \right\rangle [(\overline{x}).\, e] \Downarrow^{n-1}$. Applying the inductive hypothesis we can immediately discharge this case.

Case $\mathbb{C} = \mathbb{V}$ There are 5 cases to consider depending on the form of $\mathbb{K}$: either $\mathbb{K}$ is ϵ (in which case this is a type 1 context, so we are done), or one of the reduction rules that have a value in focus will apply. For brevity, we only the case in which the form of the stack means that BETAV applies.

In this case we know that $\mathbb{V} = \lambda x\!:\!\tau.\, \mathbb{C}'$, $\mathbb{K} = \bullet\ x, \mathbb{K}_0$ and so $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle \rightsquigarrow^0 \langle \mathbb{H} \,|\, \mathbb{C}' \,|\, \mathbb{K}_0 \rangle$. By Lemma C.0.2, $\left\langle \hat{\mathbb{H}}, \mathbb{H} \middle| \mathbb{C}' \middle| \mathbb{K}_0, \hat{\mathbb{K}} \right\rangle [(\overline{x}).\, e] \Downarrow^n$ and clearly

$$\|\left\langle \hat{\mathbb{H}}, \mathbb{H} \middle| \mathbb{C}' \middle| \mathbb{K}_0, \hat{\mathbb{K}} \right\rangle [(\overline{x}).\, e]\| < \|\left\langle \hat{\mathbb{H}}, \mathbb{H} \middle| \mathbb{C} \middle| \mathbb{K}, \hat{\mathbb{K}} \right\rangle [(\overline{x}).\, e]\|$$

so we can once again discharge this case via the inductive hypothesis.

□

The definition of improvement suggests that in order to prove that one term improves another, we have to consider them in all (closed) contexts $\mathbb{C}$. Instead, a context lemma shows that it is sufficient to consider only *evaluation* contexts. To prove this, we first require an auxiliary lemma:

Lemma C.2.2 (Reference Equivalence)**.** *If for all H, K such that $\langle H \,|\, e \,|\, K \rangle$ and $\langle H \,|\, e' \,|\, K \rangle$ are closed,*

$$\langle H \,|\, e \,|\, K \rangle \Downarrow^{n} \implies \langle H \,|\, e' \,|\, K \rangle \Downarrow^{\leq n}$$

then for all H, K, x, τ such that $\langle H, x{:}\tau \mapsto e \,|\, x \,|\, K \rangle$ and $\langle H, x{:}\tau \mapsto e' \,|\, x \,|\, K \rangle$ are closed,

$$\langle H, x{:}\tau \mapsto e \,|\, x \,|\, K \rangle \Downarrow^{n} \implies \langle H, x{:}\tau \mapsto e' \,|\, x \,|\, K \rangle \Downarrow^{\leq n}$$

Proof. First consider the case where e is a non-value, so that $\langle H, x{:}\tau \mapsto e \,|\, x \,|\, K \rangle \leadsto^{0} \langle H \,|\, e \,|\, \mathbf{update}\ x{:}\tau, K \rangle$ and so $\langle H \,|\, e \,|\, \mathbf{update}\ x{:}\tau, K \rangle \Downarrow^{n}$. By the assumptions, we immediately find that $\langle H \,|\, e' \,|\, \mathbf{update}\ x{:}\tau, K \rangle \Downarrow^{\leq n}$.

Now consider the case where e' is furthermore a non-value, so that $\langle H, x{:}\tau \mapsto e' \,|\, x \,|\, K \rangle \leadsto^{0} \langle H \,|\, e' \,|\, \mathbf{update}\ x{:}\tau, K \rangle$ and so by applying the assumption we immediately find that $\langle H, x{:}\tau \mapsto e' \,|\, x \,|\, K \rangle \Downarrow^{\leq n}$ as required. In the case where e' is a value, we know that $\langle H \,|\, e' \,|\, \mathbf{update}\ x{:}\tau, K \rangle \leadsto^{0} \langle H, x{:}\tau \mapsto e' \,|\, x \,|\, K \rangle$ which in conjunction with the assumption is also sufficient to establish that $\langle H, x{:}\tau \mapsto e' \,|\, x \,|\, K \rangle \Downarrow^{\leq n}$ as required.

For the case where e is a value, notice that $\langle H \,|\, e \,|\, \mathbf{update}\ x{:}\tau, K \rangle \leadsto^{0} \langle H, x{:}\tau \mapsto e \,|\, x \,|\, K \rangle$ which establishes once again that $\langle H \,|\, e \,|\, \mathbf{update}\ x{:}\tau, K \rangle \Downarrow^{n}$, and the rest of the argument for the other can be reused wholesale. □

Lemma C.2.3 (Context Lemma)**.** *If for all H, K such that $\langle H \,|\, e_0 \,|\, K \rangle$ and $\langle H \,|\, e_1 \,|\, K \rangle$ are closed,*

$$\langle H \,|\, e_0 \,|\, K \rangle \Downarrow^{n} \implies \langle H \,|\, e_1 \,|\, K \rangle \Downarrow^{\leq n}$$

then $e_0 \gtrsim e_1$.

Proof. By the definition of $\gtrsim$ we need to show that for all $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle$ such that $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle [(\overline{x}).\, e_0]$ and $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle [(\overline{x}).\, e_1]$ are closed,

$$\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle [(\overline{x}).\, e_0] \Downarrow^{n} \implies \langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle [(\overline{x}).\, e_1] \Downarrow^{\leq n}$$

Assume the hypothesis $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle [(\overline{x}).\, e_0] \Downarrow^{n}$. Proceed by induction on the pair:

$$(n, \| \langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle [(\overline{x}).\, e_0] \|)$$

By Lemma C.2.1, $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle$ reduces in $k \geq 0$ steps to one of the following cases:

- $\langle \mathbb{H}' \,|\, \mathbb{C}' \,|\, \epsilon \rangle$ such that $\mathit{deref}(\mathbb{H}', \mathbb{C}') = \mathbb{V}$, in which case this lemma follows immediately as the hole does not participate in the computation at all.

- $\langle \mathbb{H}' \,|\, x \,|\, \mathbb{K}' \rangle$ such that $x \notin \mathit{bvs}(\mathbb{H}')$. This cannot occur because of the closedness precondition.

- $\langle \mathbb{H}' \,|\, \mathbb{C}' \,|\, \mathbb{K}' \rangle$ such that $\mathit{deref}(\mathbb{H}', \mathbb{C}') = \xi \cdot \overline{x}$

We need only consider the final case, where $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle [(\overline{x}).\, e_1] \leadsto^{k} \langle \mathbb{H}' \,|\, \mathbb{C}' \,|\, \mathbb{K}' \rangle [(\overline{x}).\, e_1]$ and so $\langle \mathbb{H}' \,|\, \mathbb{C}' \,|\, \mathbb{K}' \rangle [(\overline{x}).\, e_1] \Downarrow^{n-k}$. We consider the two situations in which the *deref* condition can hold:

1. $\mathbb{C}' = \xi \cdot \overline{x}$. In this case, let $\mathbb{H}'_0 = \mathbb{H}'_1 = \mathbb{H}'$, $\mathbb{C}'_0 = e_0$ and $\mathbb{C}'_1 = e_1$

2. $\mathbb{C}' = x$, $\mathbb{H}' = x : \tau \mapsto \xi \cdot \overline{x}, \mathbb{H}'_{rest}$. In this case, let $\mathbb{H}'_0 = x : \tau \mapsto e_0, \mathbb{H}'_{rest}$, $\mathbb{H}'_1 = x : \tau \mapsto e_1, \mathbb{H}'_{rest}$ and $\mathbb{C}'_0 = \mathbb{C}'_1 = x$

In either case of *deref*, we can prove that:

- For $i \in \{0,1\}$, $\mathit{deref}(\mathbb{H}'_i, \mathbb{C}'_i) = e_i$
- For $i \in \{0,1\}$, $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle \left[(\overline{x}).\, e_i \right] \rightsquigarrow^k \langle \mathbb{H}'_i \,|\, \mathbb{C}'_i \,|\, \mathbb{K}' \rangle \left[(\overline{x}).\, e_i \right]$
- For any m, if $\langle \mathbb{H}'_0 \,|\, \mathbb{C}'_0 \,|\, \mathbb{K}' \rangle \left[(\overline{x}).\, e_1 \right] \Downarrow^m$ then $\langle \mathbb{H}'_1 \,|\, \mathbb{C}'_1 \,|\, \mathbb{K}' \rangle \left[(\overline{x}).\, e_1 \right] \Downarrow^{\leq m}$. In the first case, this can be shown by a direct application of the assumption. In the second case, it follows from combining the assumption with Lemma C.2.2.

Now apply Lemma C.2.1 to the state $\langle \mathbb{H}'_0 \,|\, \mathbb{C}'_0 \,|\, \mathbb{K}' \rangle$. Again, bearing in mind the closedness precondition, this state will be reduced in $l \geq 0$ steps to one of two forms, which we consider separately.

Case $\langle \mathbb{H}'_0 \,|\, \mathbb{C}'_0 \,|\, \mathbb{K}' \rangle \rightsquigarrow^l \langle \mathbb{H}'' \,|\, \mathbb{C}'' \,|\, \epsilon \rangle$ **where** $\mathit{deref}(\mathbb{H}'', \mathbb{C}'') = \mathbb{V}$ It immediately follows that $\langle \mathbb{H}'_0 \,|\, \mathbb{C}'_0 \,|\, \mathbb{K}' \rangle \left[(\overline{x}).\, e_1 \right] \rightsquigarrow^l \langle \mathbb{H}'' \,|\, \mathbb{C}'' \,|\, \epsilon \rangle \left[(\overline{x}).\, e_1 \right]$ and so $\langle \mathbb{H}'_0 \,|\, \mathbb{C}'_0 \,|\, \mathbb{K}' \rangle \left[(\overline{x}).\, e_1 \right] \Downarrow^l$.

By the assumptions, it follows that $\langle \mathbb{H}'_1 \,|\, \mathbb{C}'_1 \,|\, \mathbb{K}' \rangle \left[(\overline{x}).\, e_1 \right] \Downarrow^{\leq l}$. Combined with the fact we already know that $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle \left[(\overline{x}).\, e_1 \right] \rightsquigarrow^k \langle \mathbb{H}'_1 \,|\, \mathbb{C}'_1 \,|\, \mathbb{K}' \rangle \left[(\overline{x}).\, e_1 \right]$, this shows that $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle \left[(\overline{x}).\, e_1 \right] \Downarrow^{\leq (k+l=n)}$ as required.

Case $\langle \mathbb{H}'_0 \,|\, \mathbb{C}'_0 \,|\, \mathbb{K}' \rangle \rightsquigarrow^l \langle \mathbb{H}'' \,|\, \mathbb{C}'' \,|\, \mathbb{K}'' \rangle$ **where** $\mathit{deref}(\mathbb{H}'', \mathbb{C}'') = \xi \cdot \overline{x}$ It immediately follows that $\langle \mathbb{H}'' \,|\, \mathbb{C}'' \,|\, \mathbb{K}'' \rangle \left[(\overline{x}).\, e_0 \right] \Downarrow^{n-(k+l)}$. It is now safe to apply the induction hypothesis since e_0 is ξ-free, so we must have applied at least one reduction rule within these l steps, and so using Lemma C.1.1 we know that either $k + l > 0$ or $\| \langle \mathbb{H}'' \,|\, \mathbb{C}'' \,|\, \mathbb{K}'' \rangle \left[(\overline{x}).\, e_0 \right] \| < \| \langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle \left[(\overline{x}).\, e_0 \right] \|$. The induction hypothesis shows that $\langle \mathbb{H}'' \,|\, \mathbb{C}'' \,|\, \mathbb{K}'' \rangle \left[(\overline{x}).\, e_1 \right] \Downarrow^{\leq (n-(k+l))}$ and so $\langle \mathbb{H}'_0 \,|\, \mathbb{C}'_0 \,|\, \mathbb{K}' \rangle \left[(\overline{x}).\, e_1 \right] \Downarrow^{\leq (n-k)}$. Applying the assumptions, we find that, as required, $\langle \mathbb{H}'_1 \,|\, \mathbb{C}'_1 \,|\, \mathbb{K}' \rangle \left[(\overline{x}).\, e_1 \right] \Downarrow^{\leq (n-k)}$ and so $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle \left[(\overline{x}).\, e_1 \right] \Downarrow^{\leq n}$.

□

C.3 Value-β: an example use of improvement theory

We can use our newly-developed lemmas to prove some interesting laws of the "tick algebra" for Core. The value-β law is an interesting example that illustrates how the standard theorems can be adapted and proved in our setting:

Theorem C.3.1 (Value-β)**.** $\textbf{let}\ x = v;\ \overline{y = \mathbb{C}[x]}\ \textbf{in}\ \mathbb{C}[x] \gtrsim \textbf{let}\ x = v;\ \overline{y = \mathbb{C}[v]}\ \textbf{in}\ \mathbb{C}[v]$

Proof. By Lemma C.2.3, it suffices to show that for all $\mathbb{H}$, $\mathbb{K}$ and $\mathbb{C}$ such that $(\{x\} \cup \mathrm{fvs}(v)) \subseteq \overline{x}$ and the states formed are closed, then

$$\langle \mathbb{H}[(\overline{x}).\, x], x \mapsto v \,|\, \mathbb{C}[(\overline{x}).\, x] \,|\, \mathbb{K}[(\overline{x}).\, x] \rangle \Downarrow^n \implies \langle \mathbb{H}[(\overline{x}).\, v], x \mapsto v \,|\, \mathbb{C}[(\overline{x}).\, v] \,|\, \mathbb{K}[(\overline{x}).\, v] \rangle \Downarrow^{\leq n}$$

Assume the premise $\langle \mathbb{H}[(\overline{x}).\, x], x \mapsto v \,|\, \mathbb{C}[(\overline{x}).\, x] \,|\, \mathbb{K}[(\overline{x}).\, x] \rangle \Downarrow^n$ and proceed by induction on the pair of natural numbers $(n, \| \langle \mathbb{H}[(\overline{x}).\, x], x \mapsto v \,|\, \mathbb{C}[(\overline{x}).\, x] \,|\, \mathbb{K}[(\overline{x}).\, x] \rangle \|)$. Apply Lemma C.2.1 to $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle$ and consider each of the three cases separately:

Case $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle \rightsquigarrow^k \langle \mathbb{H}' \,|\, \mathbb{C}' \,|\, \mathbb{K}' \rangle$ **where** $\mathit{deref}(\mathbb{H}', \mathbb{C}') = \mathbb{V}$ In this case $n = k$ and we have trivially proved the goal.

Case $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle \rightsquigarrow^k \langle \mathbb{H}' \,|\, \mathbb{C}' \,|\, \mathbb{K}' \rangle$ **where** $\mathit{deref}(\mathbb{H}', \mathbb{C}') = \xi \cdot \overline{x}$ To proceed further, we need to consider cases on *deref*:

1. $\mathbb{C}' = \xi \cdot \overline{x}$, in which case

$$\begin{aligned} \langle \mathbb{H}[(\overline{x}).\, x], x \mapsto v \,|\, \mathbb{C}[(\overline{x}).\, x] \,|\, \mathbb{K}[(\overline{x}).\, x] \rangle &\rightsquigarrow^k \langle \mathbb{H}'[(\overline{x}).\, x], x \mapsto v \,|\, \mathbb{C}'[(\overline{x}).\, x] \,|\, \mathbb{K}'[(\overline{x}).\, x] \rangle \\ &= \langle \mathbb{H}'[(\overline{x}).\, x], x \mapsto v \,|\, x \,|\, \mathbb{K}'[(\overline{x}).\, x] \rangle \end{aligned}$$

2. $\mathbb{C}' = y$, $\mathbb{H}' = y \mapsto \xi \cdot \overline{x}, \mathbb{H}'_{\mathit{rest}}$, in which case

$$\begin{aligned} \langle \mathbb{H}[(\overline{x}).\, x], x \mapsto v \,|\, \mathbb{C}[(\overline{x}).\, x] \,|\, \mathbb{K}[(\overline{x}).\, x] \rangle &\rightsquigarrow^k \langle \mathbb{H}'[(\overline{x}).\, x], x \mapsto v \,|\, \mathbb{C}'[(\overline{x}).\, x] \,|\, \mathbb{K}'[(\overline{x}).\, x] \rangle \\ &= \langle \mathbb{H}'_{\mathit{rest}}[(\overline{x}).\, x], y \mapsto x, x \mapsto v \,|\, y \,|\, \mathbb{K}'[(\overline{x}).\, x] \rangle \\ &\rightsquigarrow^0 \langle \mathbb{H}'_{\mathit{rest}}[(\overline{x}).\, x], y \mapsto x, x \mapsto v \,|\, x \,|\, \mathbb{K}'[(\overline{x}).\, x] \rangle \\ &= \langle \mathbb{H}'[(\overline{x}).\, x], x \mapsto v \,|\, x \,|\, \mathbb{K}'[(\overline{x}).\, x] \rangle \end{aligned}$$

This shows that both possibilities are confluent and we can consider them together. We now need to consider cases on $\mathbb{K}'$. For brevity, we restrict attention to when the BETA rule is applicable, in which case $\mathbb{K}' = \bullet\ z, \mathbb{K}'_{\mathit{rest}}$ and by termination of the extended state $v = \lambda z{:}\tau.\, e$, so $\langle \mathbb{H}'[(\overline{x}).\, x], x \mapsto v \,|\, x \,|\, \mathbb{K}'[(\overline{x}).\, x] \rangle \rightsquigarrow^1 \langle \mathbb{H}'[(\overline{x}).\, x], x \mapsto v \,|\, e \,|\, \mathbb{K}'_{\mathit{rest}}[(\overline{x}).\, x] \rangle$.

Since $\langle \mathbb{H}'[(\overline{x}).\, x], x \mapsto v \,|\, e \,|\, \mathbb{K}'_{\mathit{rest}}[(\overline{x}).\, x] \rangle \Downarrow^{n-(k+1)}$, we can apply the inductive hypothesis to obtain $\langle \mathbb{H}'[(\overline{x}).\, v], x \mapsto v \,|\, e \,|\, \mathbb{K}'_{\mathit{rest}}[(\overline{x}).\, v] \rangle \Downarrow^{\leq (n-(k+1))}$. It follows directly that, as required, $\langle \mathbb{H}[(\overline{x}).\, v], x \mapsto v \,|\, \mathbb{C}[(\overline{x}).\, v] \,|\, \mathbb{K}[(\overline{x}).\, v] \rangle \Downarrow^{\leq n}$.

Case $\langle \mathbb{H} \,|\, \mathbb{C} \,|\, \mathbb{K} \rangle \rightsquigarrow^k \langle \mathbb{H}' \,|\, x \,|\, \mathbb{K}' \rangle$ By cases on v and $\mathbb{K}'$, just as in the previous case.

□

Note that in Moran and Sands' original presentation of the tick algebra for call-by-need [Moran and Sands, 1999a], the value-β rule specified a cost equivalence rather than a simple improvement, i.e. $\mathbf{let}\ x = v;\ \overline{y = \mathbb{C}[x]}\ \mathbf{in}\ \mathbb{C}[x] \underset{\sim}{\lhd\rhd} \mathbf{let}\ x = v;\ \overline{y = \mathbb{C}[\checkmark v]}\ \mathbf{in}\ \mathbb{C}[\checkmark v]$. (Recall that a tick $\checkmark e$ slows down the execution of a term by exactly one step.) This cost equivalence does not hold in our system: note that

$$\begin{aligned} &\mathbf{let}\ x = \mathit{True} \\ &\mathbf{in}\ \mathbf{if}\ x\ \mathbf{then}\ \mathit{True}\ \mathbf{then}\ \mathit{False} \end{aligned}$$

reduces to a value in 0 steps, whereas

$$\begin{aligned} &\mathbf{let}\ x = \mathit{True} \\ &\mathbf{in}\ \mathbf{if}\ \checkmark\mathit{True}\ \mathbf{then}\ \mathit{True}\ \mathbf{then}\ \mathit{False} \end{aligned}$$

requires a single non-normalising step to reach a value. Equally, the "tickless" law $\mathbf{let}\ x = v;\ \overline{y = \mathbb{C}[x]}\ \mathbf{in}\ \mathbb{C}[x] \underset{\sim}{\lhd\rhd} \mathbf{let}\ x = v;\ \overline{y = \mathbb{C}[v]}\ \mathbf{in}\ \mathbb{C}[v]$ does not hold in general: consider that

$$\begin{aligned} &\mathbf{let}\ f = \lambda y.\ \mathit{True} \\ &\mathbf{in}\ f\ y \end{aligned}$$

reduces with 1 use of a non-normalising reduction step to *True*, but

$$\begin{aligned} &\mathbf{let}\ f = \lambda y.\ \mathit{True} \\ &\mathbf{in}\ (\lambda y \to \mathit{True})\ y \end{aligned}$$

does not require any non-normalising steps to reach that same value.

These special cases do hold, however:

$$\begin{aligned} \mathbf{let}\ x = \lambda z.\, e;\ \overline{y = \mathbb{C}[x]}\ \mathbf{in}\ \mathbb{C}[x] &\underset{\sim}{\lhd\rhd} \mathbf{let}\ x = \lambda z.\, e;\ \overline{y = \mathbb{C}[\checkmark \lambda z.\, e]}\ \mathbf{in}\ \mathbb{C}[\checkmark \lambda z.\, e] \\ \mathbf{let}\ x = \Lambda\alpha.\, e;\ \overline{y = \mathbb{C}[x]}\ \mathbf{in}\ \mathbb{C}[x] &\underset{\sim}{\lhd\rhd} \mathbf{let}\ x = \Lambda\alpha.\, e;\ \overline{y = \mathbb{C}[\checkmark \Lambda\alpha.\, e]}\ \mathbf{in}\ \mathbb{C}[\checkmark \Lambda\alpha.\, e] \\ \mathbf{let}\ x = \mathbf{C}\ \overline{\tau}\ \overline{z};\ \overline{y = \mathbb{C}[x]}\ \mathbf{in}\ \mathbb{C}[x] &\underset{\sim}{\lhd\rhd} \mathbf{let}\ x = \mathbf{C}\ \overline{\tau}\ \overline{z};\ \overline{y = \mathbb{C}[\mathbf{C}\ \overline{\tau}\ \overline{z}]}\ \mathbf{in}\ \mathbb{C}[\mathbf{C}\ \overline{\tau}\ \overline{z}] \end{aligned}$$

C.4 The improvement theorem

Finally, we will show how the (very useful) "improvement theorem" can be adapted to our setting. Once again, we require a simple lemma before the main proof:

Lemma C.4.1. *If* $\mathbf{let}\ f\ =\ v\ \mathbf{in}\ e \mathrel{\underset{\sim}{\triangleright}} \mathbf{let}\ f\ =\ v\ \mathbf{in}\ e'$ *then for all closing contexts* H *and* K*,*

$$\langle H, f \mapsto v \,|\, e \,|\, K \rangle \Downarrow^{n} \Longrightarrow \langle H, f \mapsto v \,|\, e' \,|\, K \rangle \Downarrow^{\leq n}$$

Proof. By the definition of $\underset{\sim}{\triangleright}$, we know that:

$$\langle H \,|\, \mathbf{let}\ f\ =\ v\ \mathbf{in}\ e \,|\, K \rangle \Downarrow^{m} \Longrightarrow \langle H \,|\, \mathbf{let}\ f\ =\ v\ \mathbf{in}\ e' \,|\, K \rangle \Downarrow^{\leq m}$$

By the reduction rules, we know that both:

$$\begin{array}{rcl} \langle H \,|\, \mathbf{let}\ f\ =\ v\ \mathbf{in}\ e \,|\, K \rangle & \leadsto^{0} & \langle H, f \mapsto v \,|\, e \,|\, K \rangle \\ \langle H \,|\, \mathbf{let}\ f\ =\ v\ \mathbf{in}\ e' \,|\, K \rangle & \leadsto^{0} & \langle H, f \mapsto v \,|\, e' \,|\, K \rangle \end{array}$$

So the goal follows immediately. □

Theorem C.4.2 (Improvement Theorem)**.**

$$\frac{\mathbf{let}\ f\ =\ v\ \mathbf{in}\ v \mathrel{\underset{\sim}{\triangleright}} \mathbf{let}\ f\ =\ v\ \mathbf{in}\ v'}{\mathbf{let}\ f\ =\ v\ \mathbf{in}\ e \mathrel{\underset{\sim}{\triangleright}} \mathbf{let}\ f\ =\ v'\ \mathbf{in}\ e}$$

Proof. By Lemma C.2.3 it suffices to show that for all e and closing H and K, if $\langle H, f \mapsto v \,|\, e \,|\, K \rangle \Downarrow^{n}$ then $\langle H, f \mapsto v' \,|\, e \,|\, K \rangle \Downarrow^{\leq n}$.

Assume the premise $\langle H, f \mapsto v \,|\, e \,|\, K \rangle \Downarrow^{n}$ and proceed by induction on the pair of natural numbers $(n, \|\langle H, f \mapsto v \,|\, e \,|\, K \rangle\|)$. We apply Lemma C.2.1 to $\langle H \,|\, e \,|\, K \rangle$ and consider the two possible cases (since we are reducing a simple state, not a context, we can't encounter a meta-variable ξ).

Case $\langle H \,|\, e \,|\, K \rangle \leadsto^{k} \langle H' \,|\, e' \,|\, K' \rangle$ **where** $\mathit{deref}(H', e')v$ In this case we have by Lemma C.0.2 that $\langle H, f \mapsto v' \,|\, e \,|\, K \rangle \leadsto^{k} \langle H', f \mapsto v' \,|\, e' \,|\, K' \rangle$ and since $k = n$, $\langle H, f \mapsto v' \,|\, e \,|\, K \rangle \Downarrow^{\leq n}$ as required.

Case $\langle H \,|\, e \,|\, K \rangle \leadsto^{k} \langle H' \,|\, f \,|\, K' \rangle$ (Note that no other variable than f can appear in the context in this case due to the closedness condition.). In this case, for both $\hat{v} \in \{v, v'\}$

$$\langle H, f \mapsto \hat{v} \,|\, e \,|\, K \rangle \quad \leadsto^{k} \quad \langle H', f \mapsto \hat{v} \,|\, f \,|\, K' \rangle \quad \textit{Lemma C}.0.2$$

To proceed further we need to consider cases on K'. For brevity, we restrict attention to those cases in which the BETA rule is applicable, in which case $K' = \bullet\ y, K'_{\mathit{rest}}$, and by well-typedness of the state we know that $v = \lambda y{:}\tau.\, \hat{e}$ and $v' = \lambda y{:}\tau.\, \hat{e}'$, so

$$\begin{array}{rcl} \langle H', f \mapsto v \,|\, f \,|\, K' \rangle & \leadsto^{1} & \langle H', f \mapsto v \,|\, \hat{e} \,|\, K'_{\mathit{rest}} \rangle \\ \langle H', f \mapsto v' \,|\, f \,|\, K' \rangle & \leadsto^{1} & \langle H', f \mapsto v' \,|\, \hat{e}' \,|\, K'_{\mathit{rest}} \rangle \end{array}$$

We have assumed that $\mathbf{let}\ f\ =\ v\ \mathbf{in}\ v \mathrel{\underset{\sim}{\triangleright}} \mathbf{let}\ f\ =\ v\ \mathbf{in}\ v'$, and it is easy to see that this implies that $\mathbf{let}\ f\ =\ v\ \mathbf{in}\ \hat{e} \mathrel{\underset{\sim}{\triangleright}} \mathbf{let}\ f\ =\ v\ \mathbf{in}\ \hat{e}'$.

Combining this fact, Lemma C.4.1 and $\langle H', f \mapsto v \,|\, e \,|\, K'_{\mathit{rest}} \rangle \Downarrow^{n-(k+1)}$, we can derive $\langle H', f \mapsto v \,|\, \hat{e}' \,|\, K'_{\mathit{rest}} \rangle \Downarrow^{\leq(n-(k+1))}$. We can now apply the induction hypothesis to discover that $\langle H', f \mapsto v' \,|\, \hat{e}' \,|\, K'_{\mathit{rest}} \rangle \Downarrow^{\leq(n-(k+1))}$, which allows us to derive $\langle H, f \mapsto v' \,|\, e \,|\, K \rangle \Downarrow^{\leq n}$ as required.

□

Appendix D

Termination combinators

The question of termination arises over and over again when building compilers, theorem provers, or program analysers. For example, a compiler may inline a recursive function once, or twice, but should not do so forever. One way to extract the essence of the problem is this:

> **The online termination problem.** Given a finite or infinite sequence of terms (often syntax trees), $x_0, x_1, x_2, ...$, with the elements presented one by one, shout “stop” if the sequence looks as if it is diverging. Try not to shout “stop” for any finite sequence; but guarantee to shout “stop” at some point in every infinite sequence.

The test is “online” in the sense that the terms are presented one by one, and the entity producing the terms is a black box. In contrast, static, offline termination checkers analyse the producing entity and try to prove that it will never generate an infinite sequence.

Termination is a well-studied problem and many termination tests are known. But building *good* online termination tests is hard. A good test is

- *Sound*: every infinite sequence is caught by the test.
- *Lenient*: it does not prematurely terminate a sequence that is actually finite. As an extreme example, shouting ”stop” immediately is sound, but not very lenient.
- *Vigilant*: sequences of terms that are clearly growing in an “uninteresting” way are quickly reported as such—the termination test “wait for a million items then say stop” is not what we want.

These properties are in direct conflict: making a test more lenient risks making it less vigilant, or indeed unsound. Termination tests are typically tailored for a particular application, and it is all too easy to inadvertently build tests that are either unsound or too conservative.

Our contribution is to describe how to *encapsulate termination tests in a library.* Termination tests built using our library are guaranteed sound, and the library embodies standard (but tricky) techniques that support leniency and vigilance. Our specific contributions are these:

- We give the API of a combinator library that allows the client to construct sound, lenient, and vigilant termination tests (Section D.1). Our API is *modular* and *compositional*: that is, you can build complex tests by combining simpler ones.

- An API is not much good unless you can implement it. Building on classical work we show how to implement a termination test in terms of a so-called *well-quasi-order* (WQO) on the underlying type (Section D.2). WQOs compose well, and we give combinators for sums, products, finite maps, and so on.
- Termination tests for recursive types are particularly interesting (Section D.4). We generalise the classic homeomorphic embedding to our setting, and show what proof obligations arise.
- We show that some useful improvements to termination tests can be incorporated, once and for all, in our library: Section D.5.
- We show that our library subsumes several well-studied termination tests, including homeomorphic embedding [Kruskal, 1960], and tag-bags [Mitchell, 2010] (Section D.6). We further show how our combinators can capture a novel and slightly stronger version of the tag-bag termination test (Section D.6.4).

To our knowledge, our development is the first time that anyone has even identified an online termination tester as a separable abstraction, let alone provided a library to let you build such a thing. Yet an online termination-testing heuristic is built into the guts of many symbolic programs, including compilers (don't inline recursive functions forever) and theorem provers (don't explore unproductive proofs forever). We do not claim that our termination testers are better than any particular competing ones; rather, our library is a domain-specific language that makes it easy to explore a rich variety of online termination testers, while still guaranteeing that each is sound.

D.1 The client's eye view: tests and histories

Our goal is to define termination tests over terms whose type is under the control of the user. Recall that the client produces successive terms $x_0, x_1, x_2...$, and the business of the termination test is to shout "stop" if the sequence looks as if it might diverge. (In the literature the term "blow the whistle" is often used instead of "shout stop".) A possible API is thus:

```
data TTest a   -- Abstract
testSequence :: TTest a → [ a ] → Bool
```

Here a *TTest A* is a termination tester for type A. If we have such a tester, we can test a sequence of values of type $[A]$ using the function *testSequence*; a result of *True* means that the tester shouts "stop".

We will return to the question of construction of *TTest* values shortly, but we can already see one problem with the API. The client produces values $x_0, x_1, ...$, one a time. As each value is produced we want to ask "should we stop now". We can certainly do this with *testSequence*, by calling it on arguments $[x_0]$, $[x_0, x_1]$, $[x_0, x_1, x_2]$, and so on, but it could be terribly inefficient to do so. Each call to *testSequence* may have to check the entire sequence in case earlier elements have changed, rather than just looking at the most recent addition. A better API would allow you to say "here is one new value to add to the ones you already have". Thus:

```
data History a   -- Abstract
initHistory :: TTest a → History a
```

```
test :: History a → a → TestResult a
data TestResult a = Stop | Continue (History a)
```

A *History A* is an abstract type that embodies the knowledge about the terms (of type *A*) seen so far. The function *initHistory* creates an empty history from a termination test. Given such a history, a client can use *test* to extend the history with one new term. The *test* indicates that it has blown the whistle by returning *Stop*. Otherwise it returns a new history, augmented with the new term.

That leaves the question of how one creates a termination test in the first place. The exact test you want to use will depend greatly on the application, and so it is crucial that there is significant flexibility in defining them. Our library is therefore structured as a number of composable combinators to allow flexibility and rapid experimentation.

Our combinator library uses a type directed approach. A subset of the API is as follows:

```
intT    :: TTest Int
boolT   :: TTest Bool
pairT   :: TTest a → TTest b → TTest (a, b)
eitherT :: TTest a → TTest b → TTest (Either a b)
cofmap  :: (a → b) → TTest b → TTest a
```

We provide built-in tests for *Int* and *Bool*, and a way to compose simple tests together to make more complex ones. (We will tackle the question of recursive types in Section D.4.)

Note that *TTest* is abstract, so that the client can only construct termination tests using the combinators of the library. That is the basis for our guarantee that the termination test is sound.

As an example, here is how a client could make a *History* that (via *test*) can be use to monitor sequences of (*Int*, *Bool*) pairs:

```
myHistory :: History (Int, Bool)
myHistory = initHistory (intT `pairT` boolT)
```

An artificial example of how this *History* could be used to implement an online termination test follows. Let's say that we have a possibly-infinite list *vals* :: [(*Int*, *Bool*)] from which we would like to take the last item. However, the list is potentially infinite, and we would like to give up and return an intermediate element if we don't reach the end of the list promptly. A suitable value *vals_last* can be obtained as follows:

```
vals_last :: (Int, Bool)
vals_last = go myHistory init_lst init_rst
  where
    (init_lst : init_rst) = vals
    go hist lst rst = case test hist lst of
      Continue hist′ | (lst′ : rst′) ← rst
        → go hist′ lst′ rst′
      _ → lst
```

We know that *vals_last* will be defined (if the elements of *vals* are, and *vals* has at least one item in it) because the termination test promises to eventually shout "stop". As long as our termination test *intT* `*pairT*` *boolT* is reasonably lenient we can expect to extract the final value from a truly finite *vals* list with high probability, while still gracefully failing for "bad" infinite lists.

More realistic (but more complicated) examples can be found in Section D.6.

D.2 Termination tests and well-quasi-orders

Now that we have sketched the API for our library, we turn to the question of implementing it. The way that humans intuitively look for termination is to find a totally-ordered, well-founded "measure" and check that it is decreasing. For example, if each member of a sequence of syntax trees has strictly fewer nodes than the preceding member, the sequence cannot be infinite; here the measure is the number of nodes in the tree.

The trouble is that it can be difficult to find a simple, strictly-decreasing measure, except ones that are absurdly conservative, especially when the elements are trees. For example, the size-reduction criterion on syntax trees is sound, but far too conservative: in a compiler, inlining a function often increases the size of the syntax tree, even though progress is being made.

This is a well-studied problem [Leuschel, 1998]. The most widely-used approach is to use a so-called *well-quasi-order* (WQO) instead of a well-founded order. In this section we'll explore what WQOs are, why they are good for termination testing, and how to build WQOs using our library.

D.2.1 What is a WQO?

Definition D.2.1. A *well-quasi-order* on A is a transitive binary relation $\trianglelefteq \in A \times A$, such that for any infinite sequence $\overline{x}^\infty \in A^\infty$, there exist $i, j > i$ such that $x_i \trianglelefteq x_j$.

For example $\leq$ is a WQO on the natural numbers; in any infinite sequence of natural numbers there must be an x_i, x_j with $i < j$, and $x_i \leq x_j$. However, a WQO $\trianglelefteq$ is not *total*; that is, there may be pairs of elements of A that are not related by $\trianglelefteq$ in either direction. A WQO is transitive by definition, and is necessarily reflexive:

Lemma D.2.1. *All well-quasi-orders are reflexive.*

Proof. For any $x \in A$, form the infinite sequence $x, x, x, \ldots$. By the well-quasi-order property it immediately follows that $x \trianglelefteq x$. □

The significance of a WQO is that every infinite sequence has at least one pair related by the WQO. (In fact, infinitely many such pairs, since the sequence remains infinite if you delete the pair thus identified.) We say that a sequence $\overline{x}$ is *rejected by* $\trianglelefteq$ if there exists such a pair:

Definition D.2.2. A finite or infinite sequence $\overline{x} \in \overline{A}$ is *rejected by relation R* if $\exists i, j > i.\ R(x_i, x_j)$. A sequence is *accepted* if it is not rejected.

The relation $\trianglelefteq$ is a WQO if and only if every infinite sequence is rejected by $\trianglelefteq$[1]. Hence, given an implementation of *TTest* that uses WQOs, it is easy to implement a *History*:

```
data TTest a = WQO {(⊴) :: a → a → Bool}
newtype History a = H {test :: a → TestResult a}
initHistory :: ∀a.TTest a → History a
initHistory (WQO (⊴)) = H (go [ ])
  where
    go :: [a] → a → TestResult a
    go xs x
```

[1] In the literature, a sequence is "good for $\trianglelefteq$" iff it is rejected by $\trianglelefteq$. This terminology seems back to front in our application, so we do not use it.

```
    | any (⊴ x) xs = Stop
    | otherwise    = Continue (H (go (x : xs)))
```

A termination test, of type *TTest*, is represented simply by a WQO. A *History* closes over both the WQO $\trianglelefteq$ and a list *xs* of all the values seen so far. The invariant is that *xs* is accepted by $\trianglelefteq$. When testing a new value, we compare it with all values in *xs*; if any are related to it by *wqo*, we blow the whistle by returning *Stop*; otherwise we extend *xs* and *Continue*.

Notice that basing a termination test on a WQO is somewhat less efficient than basing it on a total, well-founded measure, because in the latter case we could maintain a single monotonically-decreasing value, and blow the whistle if the newly presented value is not smaller. In exchange WQOs are simpler, more composable, and more lenient. In Section D.5.1, we will show how we can use the fact that well-quasi-orders are transitive to reduce the length of history, which would otherwise get extended by one element each and every time *test* is called.

D.2.2 Why WQOs are good for termination tests

WQOs make it easier to construct good termination tests. For example, suppose we are interested in termination of a sequence of finite strings, consisting only of the 26 lower-case letters; for example

$$\begin{array}{ll}
[\,abc, ac, a\,] & (1) \\
[\,a, b, c\,] & (2) \\
[\,c, b, a\,] & (3) \\
[\,aa, ccc, bbbbaa, ca\,] & (4)
\end{array}$$

One can invent a total order on such strings, based on their length, or on their lexicographic ordering, but it is not altogether easy to think of one for which all the above sequences are strictly decreasing.

Here is a WQO on such strings, inspired by Mitchell [2010]:

$$s_1 \trianglelefteq_s s_2 \quad \text{iff} \quad set(s_1) = set(s_2) \text{ and } \#s_1 \leq \#s_2$$

where $set(s)$ is the set of characters mentioned in s, and $\#s$ is the length of s. Notice that strings for which $set(s_1) \neq set(s_2)$ are unrelated by $\trianglelefteq_s$, which makes it harder for $\trianglelefteq_s$ to hold, and hence makes the corresponding termination test more lenient. For example, all the sequences (1-4) above are good for this WQO.

But is this relation really a WQO? The reader is invited to pause for a moment, to prove that it is. Doing so is not immediate—which is a very good reason for encapsulating such proofs in a library and do them once rather than repeatedly for each application. Anyway, here is a proof:

Theorem D.2.2. *The relation $\trianglelefteq_s$ is a well-quasi-order.*

Proof. Transitivity of $\trianglelefteq_s$ is straightforward, but we must also check that every infinite sequence is rejected by $\trianglelefteq_s$. Suppose we have an infinite sequence of strings. Partition the sequence into at most 2^{26} sub-sequences by set equality. At least one of these sequences must also be infinite, say $\overline{x}^{\infty}$. The length of the strings in this sequence cannot be strictly decreasing (since lengths are bounded below by zero). So we can find two elements x_i, x_j with $i < j$ and $x_i \trianglelefteq_s x_j$. □

It is often useful to find a relation that is as sparse as possible, while still remaining a WQO. For example, when solving the online termination problem we wish to delay signalling possible divergence for as long as we reasonably can.

Following this principle, me can make our string example sparser still like this:

$$\begin{aligned} s_1 \trianglelefteq_t s_2 \quad \text{iff} \quad & set(s_1) = set(s_2) \text{ and} \\ & \forall c \in [a...z].\ N(s_1, c) \leq N(s_2, c) \end{aligned}$$

where $N(s,c)$ is the number of occurrences of letter c in string s. So $s_1 \trianglelefteq_t s_2$ only if s_1 has no more a's than s_2, *and* no more b's, *and* no more c's, etc. These conjunctions make it even harder for $s_1 \trianglelefteq_t s_2$ to hold. Exercise: prove that this too is a WQO.

We can quantify how lenient a WQO is by asking how long a sequence it can tolerate. One measure of lenience is something we call the *characteristic index*.

Definition D.2.3 (Characteristic index)**.** The characteristic index $K(\trianglelefteq, \overline{x}^\infty)$ of a WQO $\trianglelefteq$, relative to a finite or infinite sequence $\overline{x}^\infty$, is the largest index n for which $x_0, \ldots, x_n$ is accepted by $\trianglelefteq$.

One WQO is (strictly) more lenient than another if it always has a bigger characteristic index:

Definition D.2.4 (Lenience)**.** A WQO $\trianglelefteq_1$ is more lenient than $\trianglelefteq_2$ if $K(\trianglelefteq_1, \overline{x}) > K(\trianglelefteq_2, \overline{x})$ for every infinite sequence $\overline{x}$.

This is a rather strong definition of lenience: in practice, we are also interested in well-quasi-orders that *tend* to be more lenient than others on commonly-encountered sequences. However, this definition will suffice for this paper.

D.3 Termination combinators

In this section we describe the primitive combinators provided by our library, and prove that they construct correct WQOs.

D.3.1 The trivial test

The simplest WQO is one that relates everything, and hence blows the whistle immediately:

$$\begin{aligned} &alwaysT :: TTest\ a \\ &alwaysT = WQO\ (\lambda x\ y.\ True) \end{aligned}$$

This *alwaysT* is trivially correct, and not at all lenient. Nonetheless, it can be usefully deployed as a "placeholder" well-quasi-order when we have yet to elaborate a well-quasi-order, or a natural well-quasi-order does not exist (e.g. consider well-quasi-ordering values of type *IO Int*).

D.3.2 Termination for finite sets

Our next combinator deals with termination over finite sets:

$$\begin{aligned} &finiteT :: \forall a. Finite\ a \Rightarrow TTest\ a \\ &finiteT = WQO\ (\equiv) \end{aligned}$$

```
class Eq a ⇒ Finite a where
  elements :: [ a ]   -- Members of the type
```

This WQO relates equal elements, leaving unequal elements unrelated. Provided all the elements are drawn from a finite set, (≡) is indeed a WQO:

Proof. Consider an arbitrary sequence $\overline{x}^{\infty} \in A^{\infty}$ where there are a finite number of elements of A. Since A is finite, the sequence must repeat itself at some point—i.e. $\exists jk.j \neq k \wedge x_j = x_k$. The existence of this pair proves that *finiteT* defines a well-quasi-order. Meanwhile, transitivity follows trivially from the transitivity of (≡). □

Using *finiteT*, we can trivially define the *boolT* combinator used in the introduction:

```
boolT :: TTest Bool
boolT = finiteT
```

The combinator *finiteT* is polymorphic. The fact that the element type *a* must be finite using the "*Finite a* ⇒" constraint in *finiteT*'s type. But there is clearly something odd here. First, 'finiteT' does not use any methods of class *Finite*, and second, it is the the client who makes a new type T into an instance of *Finite*, and the library has no way to check that the instance is telling the truth. For example, a client could bogusly say:

```
instance Finite Integer where
  elements = [ ]
```

Moreover, the user could give a bogus implementation of equality:

```
data T = A | B
instance Eq T where
  (≡) p q = False
instance Finite T where
  elements = [ A, B ]
```

Here the new type T is finite, but since the equality function always returns *False*, the whistle will never blow.

So our library guarantees the soundness of the termination testers *under the assumption that the instances of certain classes at the element type A satisfy corresponding correctness conditions.* Specifically:

- (≡) must be reflexive and transitive at type A.
- The type A must have only a finite number of distinct elements (distinct according to (≡), that is).

Another way to say this is that the instances of *Eq* and *Finite* form part of the trusted code base. This is not unreasonable. On the one hand, these proof obligations are simple for the programmer to undertake—much, much simpler than proving that a particular Boolean-valued function is a WQO.

On the other hand, it is unrealistic for the library to check that *elements* is a finite list and that the two values we compare are elements of that finite list, for instance, by using runtime assertions. In the example of Section D.2.2 there are 2^{26} elements of the type *Set Char*, so making these checks at runtime would be a very bad idea.

D.3.3 Termination for well-ordered sets

Another very useful primitive well-quasi-order is that on elements drawn from well-ordered sets: every well-order is a well-quasi-order (but clearly not vice-versa):

```
wellOrderedT :: WellOrdered a ⇒ TTest a
wellOrderedT = WQO (≤)
class Ord a ⇒ WellOrdered a
```

Similar to *Finite*, the *WellOrdered* predicate picks out types with least elements; that is ones have a total order (hence the *Ord* superclass) and a least element. The client's proof obligations about instances of a type A are:

- $(\le)$ defines a total order (i.e. it is antisymmetric, transitive and total)
- For every (possibly infinite) non-empty set $X \subseteq A$ of elements, $\exists (y :: A) \in X. \forall (x :: A) \in X. y \le x$.

Under these conditions, $(\le)$ is a WQO:

Proof. Transitivity is immediate by assumption. Now consider an arbitrary sequence $\overline{x}^{\infty}$. Each pair of adjacent elements x_j, x_{j+1} in the sequence is either shrinking (so $\neg(x_j \le x_{j+1})$) or non-decreasing (so $x_j \le x_{j+1}$). If we have at least one pair of the latter kind, the well-quasi-order property holds. The dangerous possibility is that all our pairs may be of the former sort.

Because we have that $\forall j. \neg(x_j \le x_{j+1})$, by the reflexivity of $\le$ we know that $\forall j. \neg(x_j < x_{j+1})$—i.e. we have an infinitely descending chain. However, this fact contradicts the assumption that $\le$ is a well-order. □

Given *wellOrderedT* and an instance *WellOrdered Int*, it is trivial to define a suitable *intT* (as used in the introduction):

```
intT :: TTest Int
intT = wellOrderedT
```

D.3.4 Functorality of termination tests

Now that we have defined a number of primitive termination tests, we are interested in defining some combinators that let us combine these tests into more powerful ones. The first of these shows that *TTest* is a contravariant functor:

```
class Cofunctor f where
  cofmap :: (b → a) → f a → f b
instance Cofunctor TTest where
  cofmap f (WQO (⊴)) = WQO $ λx y.f x ⊴ f y
```

So, for example, here is how a client could build a (not very good) termination test for labelled rose trees:

```
data Tree = Node Label [Tree]
size :: Tree → Int
size (Tree n ts) = 1 + sum (map size ts)
```

$$
\begin{aligned}
&\mathit{treeT} :: \mathit{TTest}\ \mathit{Tree}\\
&\mathit{treeT} = \mathit{cofmap}\ \mathit{size}\ \mathit{wellOrderedT}
\end{aligned}
$$

Here we use *size* to take the size of a tree, and use the fact that *Int* is well-ordered by $\leq$ as the underlying termination test.

The defining laws of contravariant functors (cofunctors) are:

1. Identity: $\mathit{cofmap}\ \mathit{id} = \mathit{id}$

2. Composition: $\mathit{cofmap}\ f.\mathit{cofmap}\ g = \mathit{cofmap}\ (g.f)$

These two laws are easy to verify for *TTest* instance above. Similarly, it is easy to to show that $(\mathit{cofmap}\ f\ t)$ is a well-quasi-order if t is.

Intuitively, the reason that *TTest* is a *contravariant* functor is that it $\mathit{TTest}\ a$ is a *consumer* rather than a producer of values of type a. For the same reason, the arrow type $(\rightarrow)$ is contravariant in its first type argument.

In section Section D.5.2, we show how this definition of $\mathit{cofmap}\ f$ can be improved.

D.3.5 Termination for sums

We are able to build termination test for sum types, given tests for the components:

$$
\begin{aligned}
&\mathit{eitherT} :: \mathit{TTest}\ a \rightarrow \mathit{TTest}\ b \rightarrow \mathit{TTest}\ (\mathit{Either}\ a\ b)\\
&\mathit{eitherT}\ (\mathit{WQO}\ (\trianglelefteq_a))\ (\mathit{WQO}\ (\trianglelefteq_b)) = \mathit{WQO}\ (\trianglelefteq)\\
&\quad \textbf{where}\\
&\qquad (\mathit{Left}\ a_1)\ \trianglelefteq (\mathit{Left}\ a_2) \quad = a_1 \trianglelefteq_a a_2\\
&\qquad (\mathit{Right}\ b_1) \trianglelefteq (\mathit{Right}\ b_2) = b_1 \trianglelefteq_b b_2\\
&\qquad \_ \qquad\qquad \trianglelefteq \_ \qquad\qquad = \mathit{False}
\end{aligned}
$$

The ordering used here treats elements from the same side of the sum (i.e. both *Left* or both *Right*) using the corresponding component ordering, and otherwise treats them as unordered.

Does this test define a WQO? Yes:

Proof. Consider an arbitrary sequence $\overline{x}^{\infty} \in (\mathit{Either}\ A\ B)^{\infty}$. Form the subsequences $\overline{a}^{\infty} = \{a_i \in A \mid \mathit{Left}\ a_i \in \overline{x}^{\infty}\}$ and $\overline{b}^{\infty} = \{b_i \in B \mid \mathit{Right}\ b_i \in \overline{x}^{\infty}\}$. Since the x sequence is infinite, at least one of these subsequences must be infinite. Without loss of generality, assume that the $\overline{a}^{\infty}$ sequence is infinite. Now, the fact that $\mathit{eitherT}\ (\trianglelefteq_a)\ (\trianglelefteq_b)$ is a well-quasi-order follows directly from the fact that $(\trianglelefteq_a)$ is a well-quasi-order. □

Incidentally, notice that if the component types are both $()$, the test boils down to the same as the finite-set test for *Bool* in Section D.3.2. Conversely, it is straightforward (albeit inefficient) to define *finiteT* by iterating *eitherT* once for each item in the *elements* list, and the reader is urged to do so as an exercise.

The test $\mathit{eitherT}\ (\trianglelefteq_a)\ (\trianglelefteq_b)$ is at least as lenient as $(\trianglelefteq_a)$ or $(\trianglelefteq_b)$ (in the sense of Definition D.2.4), and is often strictly more lenient. Specifically, if $\overline{x} \in \mathit{Either}\ A\ B$, and $L(\overline{x}) = \{x \mid \mathit{Left}\ x \in \overline{x}\}$, and similarly for $R(\overline{x})$, then

$$
\begin{aligned}
&\mathit{min}(K((\trianglelefteq_a), L(\overline{x})),\ K((\trianglelefteq_b), R(\overline{x})))\\
&\qquad \leq\ K(\mathit{eitherT}\ (\trianglelefteq_a)\ (\trianglelefteq_b), \overline{x})\\
&\quad \leq\ K((\trianglelefteq_a), L(\overline{x})) + K((\trianglelefteq_b), R(\overline{x}))
\end{aligned}
$$

Both the upper and lower bounds of this inequality can actually be realised. For example, with the test

$$\mathit{eitherT}\ \mathit{finiteT}\ \mathit{finiteT} :: \mathit{TTest}\ (\mathit{Either}\ ()\ \mathit{Bool})$$

the lower bound is realised by $\overline{x}^{\infty} = L\ (), L\ (), L\ (), \ldots$, and the upper bound by $\overline{x}^{\infty} = L\ (), R\ \mathit{True}, R\ \mathit{False}, L\ (), R\ \mathit{True}, \ldots$.

Although we haven't defined many combinators, we already have enough to be able to define natural well-quasi-orders on many simple data types. For example, we can well-quasi-order *Maybe T* if we can well-quasi-order *T* itself:

```
maybeT :: TTest a → TTest (Maybe a)
maybeT wqo = cofmap inject (eitherT alwaysT wqo)
  where
    inject Nothing = Left ()
    inject (Just x) = Right x
```

To define *maybeT* we have adopted a strategy—repeated later in this document—of "injecting" the *Maybe* data type (which our combinators cannot yet handle) into a simpler data type which is handled by a primitive combinator—in this case, *Either*[2].

Note that we use *alwaysT* from Section D.3.1 to well-quasi-order values of unit type—there really is no non-trivial way to order a type with only one value.

D.3.6 Termination for products

Just like we could for sum types, we can define a combinator for well-quasi-ordering product types, given WQOs on the component types:

```
pairT :: TTest a → TTest b → TTest (a, b)
pairT (WQO (⊴_a)) (WQO (⊴_b)) = WQO (⊴)
  where
    (a_1, b_1) ⊴ (a_2, b_2) = (a_1 ⊴_a a_2) ∧ (b_1 ⊴_b b_2)
```

The fact that *pairT* defines a WQO is quite surprising. We can assume that $\trianglelefteq_a$ and $\trianglelefteq_b$ are WQOs, but that only means that given input sequences $\overline{a}^{\infty}$ and $\overline{b}^{\infty}$ respectively, there exists some $i < j. a_i \trianglelefteq_a a_j$ and $k < l. b_k \trianglelefteq_b b_l$. Yet for *pairT* to define a WQO there must exist a $p < q$ such that $a_p \trianglelefteq_a a_q$ and *simultaneously* $b_p \trianglelefteq_b b_q$. How can we know that the related elements of the two sequences will ever "line up"?

Nonetheless, it is indeed the case, as the following proof demonstrates. First we need a lemma:

Lemma D.3.1. *For any well-quasi-order* $\trianglelefteq \in A \times A$ *and* $\overline{x}^{\infty} \in A^{\infty}$*, there exists some* $n \geq 0$ *such that* $\forall j > n. \exists k > j. x_j \trianglelefteq x_k$.

This lemma states that, beyond some some threshold value n, *every* element x_j (where $j > n$) has a related element x_k somewhere later in the sequence.

Proof. This lemma can be shown by a Ramsey argument. Consider an arbitrary sequence $\overline{x}^{\infty}$. Consider the sequence

$$\overline{y} = \{x_i \mid x_i \in \overline{x}^{\infty}, \forall j > i. \neg(x_i \trianglelefteq x_j)\}$$

of elements of $\overline{x}^{\infty}$ which are embedded into no later element. If this sequence was infinite it would violate the well-quasi-order property, since by definition none of the elements of the sequence are related by $\trianglelefteq$. Hence we have a constructive proof of the proposition if we take n to be $\max\{i \mid x_i \in \overline{y}\}$. □

[2] In this and many other examples, the GHC's optimisation passes ensure that the intermediate *Either* value is not actually constructed at runtime.

A proof of the fact that *pairT* defines a well-quasi-order as long as its two arguments does—a result that e.g. Kruskal [1960] calls the Cartesian Product Lemma— now follows:

Proof. Consider an arbitrary sequence $\overline{(a,b)}^{\infty} \in (A \times B)^{\infty}$. By Lemma D.3.1, there must be a n such that $\forall j > n. \exists k > j. a_j \trianglelefteq_a a_k$. Hence there must be at least one infinite subsequence of $\overline{a}^{\infty}$ where adjacent elements are related by $\trianglelefteq_a$—i.e. $a_n \trianglelefteq_a a_{l_0} \trianglelefteq_a a_{l_1} \trianglelefteq_a \ldots$ where $n < l_0 < l_1 < \ldots$.

Now form the infinite sequence $b_j, b_{l_0}, b_{l_1} \ldots$. By the properties of $\trianglelefteq_b$, there must exist some m and n such that $m < n$ and $b_{l_m} \trianglelefteq_b b_{l_n}$. Because $\trianglelefteq_a$ is transitive, we also know that $a_{l_m} \trianglelefteq_a a_{l_n}$.

This inference, combined with the fact that $\trianglelefteq_a$ and $\trianglelefteq_b$ are valid WQOs, and that transitivity follows by the transitivity of both the component WQOs, proves that *pairT* $\trianglelefteq_a$ $\trianglelefteq_b$ is a well-quasi-order. □

From a leniency point of view, we have a lower bound on the leniency of a test built with *pairT*:

$$\max \begin{pmatrix} K((\trianglelefteq_a), \{a_i \mid (a_i, b_i) \in \overline{x}^{\infty}\}), \\ K((\trianglelefteq_b), \{b_i \mid (a_i, b_i) \in \overline{x}^{\infty}\}) \end{pmatrix} \leq K(pairT\ (\trianglelefteq_a)\ (\trianglelefteq_b), \overline{x}^{\infty})$$

However, there is no obvious upper bound on the characteristic index. Not even

$$K((\trianglelefteq_a), \{a_i \mid (a_i, b_i) \in \overline{x}^{\infty}\}) * K((\trianglelefteq_b), \{b_i \mid (a_i, b_i) \in \overline{x}^{\infty}\})$$

is an upper bound for the characteristic index of *pairT* $(\trianglelefteq_a)$ $(\trianglelefteq_b)$—for example, the proposed upper bound is violated by the well-quasi-order *pairT finiteT wellOrderedT* and the sequence $(T, 100), (F, 100), (T, 99), (F, 99), \ldots, (F, 0)$, which has characteristic index 300, despite the component characteristic indexes being 2 and 1 respectively.

We now have enough combinators to build the string termination test from Section D.2.2:

```
stringT :: TTest String
stringT = cofmap inject (pairT finiteT wellOrderedT)
  where inject cs = (mkSet cs, length cs)
```

We assume a type of sets with the following interface:

```
instance (Ord a, Finite a) ⇒ Finite (Set a) where...
mkSet :: Ord a ⇒ [a] → Set a
```

(We use the bounded *Int* length of a string in our *stringT*, but note that this would work equally well with a hypothetical type of unbounded natural numbers *Nat*, should you define a suitable *WellOrdered Nat* instance.)

The big advantage in defining *stringT* with our combinator library is that Theorem D.2.2 in Section D.2.2 is not needed: the termination test is sound by construction, provided only that (a) there are only a finite number of distinct sets of characters, and (b) the *Int*s are well ordered.

D.3.7 Finite maps

It is often convenient to have termination tests over finite mappings, where the domain is a finite type — for example, we will need such a test in Section D.6.4. One way to implement such a test is to think of the mapping as a large (but bounded) arity tuple.

To compare *m1* and *m2*, where *m1* and *m2* are finite maps, you may imagine forming two big tuples

```
(lookup k1 m1, lookup k2 m1, ..., lookup kn m1)
(lookup k1 m2, lookup k2 m2, ..., lookup kn m2)
```

where *k1*...*kn* are all the elements of the key type. The *lookup* returns a *Maybe* and, using the rules for products (Section D.3.6), we return *False* if any of the constructors differ; that is, if the two maps have different domains. If the domains are the same, we will simply compare the corresponding elements pairwise, and we are done.

We can implement this idea as a new combinator, *finiteMapT*. We assume the following standard interface for finite maps:

```
assocs  :: Ord k ⇒ Map k v → [(k, v)]
keysSet :: Ord k ⇒ Map k v → Set k
elems   :: Ord k ⇒ Map k v → [v]
lookup  :: Ord k ⇒ k → Map k v → Maybe v
```

From which the combinator follows:

```
finiteMapT :: ∀k v.(Ord k, Finite k)
           ⇒ TTest v → TTest (Map k v)
finiteMapT (WQO (⊴)) = WQO test
  where
    test :: Map k v → Map k v → Bool
    test m1 m2 = keysSet m1 ≡ keysSet m2
               ∧ all (ok m1) (assocs m2)
    ok :: Map k v → (k, v) → Bool
    ok m1 (k2, v2) = case lookup k2 m1 of
      Just v1 → v1 ⊴ v2
      Nothing → error "finiteMapT"
```

In fact, the *finiteMapT* combinator can be defined in terms of our existing combinators, by iterating the *pairT* combinator (we also make use of *maybeT* from Section D.3.5):

```
finiteMapT_indirect :: ∀k v.(Ord k, Finite k)
                    ⇒ TTest v → TTest (Map k v)
finiteMapT_indirect wqo_val
   = go (const ()) finiteT elements
  where
    go :: ∀vtup.(Map k v → vtup) → TTest vtup → [k]
       → TTest (Map k v)
    go acc test [] = cofmap acc test
    go acc test (key : keys)
       = go acc′ (pairT (maybeT wqo_val) test) keys
      where acc′ mp = (lookup key mp, acc mp)
```

Unfortunately, this definition involves enumerating all the elements of the type (via the call to *elements*), and there might be an unreasonably large number of such elements, even though any particular *Map* might be small. For these reasons we prefer the direct implementation.

D.4 Termination tests for recursive data types

Now that we have defined well-quasi-order combinators for both sum and product types, you may very well be tempted to define a WQO for a data type such as lists like this:

```
list_bad :: ∀a.TTest a → TTest [a]
list_bad test_x = test_xs
  where
    test_xs :: TTest [a]
    test_xs = cofmap inject (eitherT finiteT
                                    (pairT test_x test_xs))
    inject []       = Left ()
    inject (y : ys) = Right (y, ys)
```

Unfortunately the *list_bad* combinator would be totally bogus. Notice that *list_bad* only relates two lists if they have exactly the same "spines" (i.e. their lengths are the same) —but unfortunately, there are infinitely many possible list spines. Thus in particular, it would be the case that the following infinite sequence would be accepted by the (non!) well-quasi-order *list_bad finite*:

$$[\,],[()],[(),()],[(),(),()],\ldots$$

We would like to prevent such bogus definitions, to preserve the safety property of our combinator library. The fundamental problem is that *list_bad* isn't well-founded in some sense: our proof of the correctness of *cofmap*, *eitherT* and so on are sufficient to show only that *test_xs* is a well-quasi-order if and only if *test_xs* is a well-quasi-order—a rather uninformative statement! This issue fundamentally arises because our mathematics is set-theoretical, whereas Haskell is a language with complete partial order (cpo) semantics.

Our approach is to rule out such definitions by making all of our combinators *strict in their well-quasi-order arguments*. Note that we originally defined *TTest* using the Haskell **data** keyword, rather than **newtype**, which means that all the combinator definitions presented so far are in fact strict in this sense. This trick means that the attempt at recursion in *list_bad* just builds a loop instead—$\forall w.\mathit{list\_bad}\ w = \bot$.

It is clear that making our well-quasi-order combinators non-strict—and thus allowing value recursion—immediately makes the combinator library unsafe. However, we still need to be able to define well-quasi-orders on recursive data types like lists and trees, which—with the combinators introduced so far—is impossible without value-recursion. To deal with recursive data types, we need to introduce an explicit combinator for reasoning about fixed points in a safe way that is *lazy* in its well-quasi-order argument, and hence can be used to break loops that would otherwise lead to divergence.

D.4.1 Well-quasi-ordering any data type

You might wonder if it is possible to naturally well-quasi-order recursive data types at all. To show that we can, we consider well-quasi-ordering a "universal data type", *UnivDT*:

```
data UnivDT = U String [UnivDT]
```

The idea is that the *String* models a constructor name, and the list the fields of the constructor. By analogy with real data types, we impose the restrictions that there are only a finite number of constructor names, and for any given constructor the length of the associated list is fixed. In particular, the finite list of constructors will contain `"Nil"` (of arity 0) and "Cons" (of arity 2), with which we can model the lists of the previous section.

We can impose a well-quasi-order on the suitably-restricted data type *UnivDT* like so:

```
univT :: TTest UnivDT
univT = WQO test
  where test u1@(U c1 us1) (U c2 us2)
          = (c1 ≡ c2 ∧ and (zipWith test us1 us2)) ∨
            any (u1`test`) us2
```

Elements *u1* and *u2* of *UnivDT* are related by the well-quasi-order if either:

- The constructors *c1* and *c2* match, and all the children *us1* and *us2* match (remember that the length of the list of children is fixed for a particular constructor, so *us1* and *us2* have the same length). When this happens, the standard terminology is that *u1* and *u2* *couple*.
- The constructors don't match, but *u1* is related by the well-quasi-order to one of the children of *u2*. The terminology is that *u1* *dives* into *u2*.

Although not immediately obvious, this test does indeed define a well-quasi-order on these tree-like structures (the proof is similar to that we present later in Section D.4.2), and it is this well-quasi-order (sometimes called the "homeomorphic embedding") which is used in most classical supercompilation work (see e.g. [Turchin, 1988]).

Once again, we stress that for this test to be correct, the constructor name must determine the number of children: without this assumption, given at least two constructors *F* and *G* you can construct a chain such as

```
U "F" [], U "F" [U "G" []], U "F" [U "G" [], U "G" []], ...
```

which is not well-quasi-ordered by the definition above.

D.4.2 Well-quasi-ordering functor fixed points

We could add the well-quasi-order on our "universal data type" as a primitive to our library. This would be sufficient to allow the user to well-quasi-order their own data types—for example, we could define an ordering on lists as follows:

```
list_univ :: TTest [UnivDT]
list_univ = cofmap to_univ univT

to_univ :: [UnivDT] → UnivDT
to_univ []       = U "Nil" []
to_univ (x : xs) = U "Cons" [x, to_univ xs]
```

However, this solution leaves something to be desired: for one, we would like to be able to well-quasi-order lists [*a*] for an arbitrary element type *a*, given a well-quasi-ordering on those elements. Furthermore, with this approach there is scope for the user to make an error in writing *to_univ* which violates the invariants on the *UnivDT* type. This would break the safety promises of the well-quasi-order library.

We propose a different solution that does not suffer from these problems. The first step is to represent data types as fixed points of functors in the standard way. For example, lists are encoded as follows:

```
newtype Fix t = Roll {unroll :: t (Fix t)}
data ListF a rec = NilF | ConsF a rec
  deriving (Functor, Foldable, Traversable)
fromList :: [a] → Fix (ListF a)
fromList [ ]      = Roll NilF
fromList (y : ys) = Roll (ConsF y (fromList ys))
```

The *fixT* combinator Our library then provides a single primitive that can be used to well-quasi-order any data type built out of this sort of explicit fixed point scheme:

```
fixT :: ∀t.Functor t
     ⇒ (∀rec.t rec → [rec])
     → (∀rec.t rec → t rec)
     → (∀rec.TTest rec → TTest (t rec))
     → TTest (Fix t)
fixT kids p f = wqo
  where
    wqo = WQO (λ(Roll a) (Roll b) → test a b)
    test a b = (⊴) (f wqo) (p a) (p b) ∨
               any (test a.unroll) (kids b)
```

The arguments of *fixT* are as follows:

- A type constructor $t :: * \to *$ that is equipped with the usual functorial lifting function $fmap :: \forall a\ b.(a \to b) \to t\ a \to t\ b$. (By chance, we do not in fact use *fmap* in our definition, though it will show up in our proof that *fixT* is correct. Alternative representations for *TTest*—such as that discussed in Section D.5.2—may indeed use *fmap* in their definition of *fixT*.)
- A function *kids* with which to extract the (or some of the) "children" of a functor application.
- A function *p* that we will call the *calibrator* whose purpose is to map elements of type *t rec* to elements of type *t rec* but where the holes in the returned shape are filled in with elements returned from the *kids* function. We explain this in detail later in this section.
- Finally, a function which determines how we will create a well-quasi-order *t rec* given a well-quasi-order for some arbitrary *rec*. The only invariant we require on this is that if given a correct well-quasi-order it returns a correct well-quasi-order. This invariant will be trivially satisfied as long as the user constructs all *TTest*s using the combinators of our library.

The definition of *test* in *fixT* is analogous to the test we saw in *univT*—the first argument of ∨ tests whether the left side couples with the right, and the second argument determines whether the left side dives into one of the *kids* of the right. The coupling case is actually slightly more general than the coupling we have seen until now, due to the calibrator *p* being applied to *a* and *b* before we compare them.

We now present the preconditions for *fixT* to define a well-quasi-order.

Definition D.4.1 (*fixT* preconditions)**.** For a particular type constructor $t :: * \to *$ equipped with the usual $fmap :: \forall\ a\ b.(a \to b) \to t\ a \to t\ b$, and functions *kids*, *p* and *f* (suitably typed) we say that they jointly satisfy the *fixT* preconditions if:

- All elements x of type *Fix t* must be finite, in the sense that *size x* is defined, where *size* is as follows:

```
size :: Fix t → Integer
size = (1+).sum.map size.kids.unroll
```

- The calibrator function p must satisfy the (non-Haskell) dependent type:

$$g :: (y : t\ a) \to t\ \{x : a \mid x \in \mathit{kids}\ y\}$$

The first condition is not interesting[3]—it ensures that we can't be calling *kids* forever while comparing two elements. The second condition is the interesting one. Typically one thinks of *kids* as returning *all* the children of a functor. For instance, consider the *BTreeF* functor below, that defines labelled binary trees:

```
data BTreeF a rec = BNil | BNode a rec rec
kids_tree :: ∀a rec.BTreeF a rec → [rec]
kids_tree BNil              = [ ]
kids_tree (BNode _ x y) = [x, y]
```

In this case, a valid calibrator is simply the identity

```
p :: ∀a rec.BTreeF a rec → BTreeF a rec
p BNil              = BNil
p (BNode a x y) = BNode a x y
```

since both x and y are returned by *kids_tree*. Consider however, a different version of *kids* that only returns the left branch of a node:

```
kids_tree_alt :: ∀a rec.BTreeF a rec → [rec]
kids_tree_alt BNil              = [ ]
kids_tree_alt (BNode _ x y) = [x]
```

A valid calibrator for this *kids_tree_alt* can only plug in the holes of the functor elements that can be returned from *kids_tree_alt*. Consider:

```
p_ok, p_bad :: BTreeF a rec → BTreeF a rec
p_ok BNil              = BNil
p_ok (BNode a x y) = BNode a x x
p_bad BNil              = BNil
p_bad (BNode a x y) = BNode a x y
```

In this example *p_ok* is a valid calibrator, as it only uses x, which belongs in the list *kids_tree_alt* (*BNode a x y*). However *p_bad* is not a valid calibrator as it uses y, which is not returned by *kids_tree_alt*. So, the role of the calibrator is to correct the behaviour of the test, depending on the implementation of *kids*.

Arguably, the extra generality of a *kids* function that does not return all kids or may have even more exotic behaviour is rarely used but provides for an elegant generic proof of correctness of *fixT*.

[3] Again, this constraint arises from our attempt to use Haskell (a language with cpo semantics) as if it had set semantics.

Using *fixT* with lists One correct way to use the *fixT* combinator is with the following *kids_list* function

```
kids_list :: ∀a rec.ListF a rec → [rec]
kids_list NilF           = [ ]
kids_list (ConsF _ xs) = [xs]
```

along with the identity calibrator to define a correct-by-construction well-quasi-order for lists (realising the "Finite Sequence Theorem" of Kruskal [1960]):

```
listT :: ∀a.TTest a → TTest [a]
listT wqo_elt
  = cofmap fromList (fixT kids_list id wqo_fix)
  where
    wqo_fix :: ∀rec.TTest rec → TTest (ListF a rec)
    wqo_fix wqo_tail
      = cofmap inject $
        eitherT finiteT (wqo_elt `pairT` wqo_tail)
    inject :: ∀rec.ListF a rec → Either () (a, rec)
    inject NilF           = Left ()
    inject (ConsF y ys) = Right (y, ys)
```

Is *fixT* correct? Now we have seen an example of the use of *fixT*, we are in a position to tackle the important question as to whether it actually defines a well-quasi-order:

Theorem D.4.1 (Correctness of *fixT*). *If the preconditions of fixT (Definition D.4.1) are satisfied then fixT kids p f defines a well-quasi-order.*

Proof. By contradiction, assume that under our assumptions, there exists at least one accepted infinite sequence $\in (\mathit{Fix}\ t)^\infty$ for the relation $(\trianglelefteq)$ $(\mathit{fixT}\ \mathit{kids}\ p\ f)$.

We pick the minimal such accepted sequence $\overline{t}^\infty$, such that for all $n \in \mathbb{N}$ and accepted sequences $\overline{s}^\infty$ such that $\forall i.0 \le i < n.t_i = s_i$, we have that $\mathit{size}\ t_n \le \mathit{size}\ s_n$.

We now form the possibly infinite set of children, D:

$$D = \{k \mid i \in \mathbb{N}, k \in \mathit{kids}\ (\mathit{unroll}\ t_i)\}$$

As a subgoal, we claim that $\mathit{fixT}\ \mathit{kids}\ p\ f :: \mathit{TTest}\ D$ is a WQO. In other words, the union of all children of the minimal sequence is well-quasi ordered by $\mathit{fixT}\ \mathit{kids}\ p\ f$. To see this, we proceed by contradiction: assume there is some accepted infinite sequence $\overline{r}^\infty \in D^\infty$. Because each $\mathit{kids}\ (\mathit{unroll}\ t_i)$ is finite (since $\mathit{size}\ t_i$ is finite), the accepted sequence $\overline{r}^\infty$ must have an infinite subsequence $\overline{q}^\infty$ such that $q_i \in \mathit{kids}\ (\mathit{unroll}\ t_{f(i)})$ for some f such that $\forall j.f(0) \le f(j)$. Given such a $\overline{q}^\infty$, we can define a new infinite sequence $\overline{s}^\infty \in (\mathit{Fix}\ t)^\infty$:

$$\overline{s}^\infty = t_0, t_1, \ldots, t_{f(0)-1}, q_{f(0)}, q_{f(1)}, \ldots$$

The sequence $\overline{s}^\infty$ must be accepted because otherwise, by the definition of *fixT* the original $\overline{t}^\infty$ would be rejected (by the "dive" rule). But if it is accepted then we have a contradiction to the minimality of $\overline{t}^\infty$ since $\mathit{size}\ q_{f(0)} < \mathit{size}\ t_{f(0)}$, $q_{f(0)} \in \mathit{kids}\ (\mathit{unroll}\ t_{f(0)})$, and the children of an element have smaller size than their parent. We conclude that $\mathit{fixT}\ \mathit{kids}\ p\ f$ is a WQO.

This fact means that $f\ (\mathit{fixT}\ \mathit{kids}\ p\ f) :: \mathit{TTest}\ (t\ D)$ is a WQO. Consider now the infinite minimal sequence $\overline{t}^\infty$ again and the mapping of each element through the calibrator

p: $u_i = p\ (unroll\ t_i)$. Each u_i has type: $u_i :: t\ \{x \mid x \in kids\ t_i\}$. Furthermore, because t is a functor and $\forall i.kids\ t_i \subseteq D$, we have that $u_i :: t\ \{x \mid x \in D\}$ and hence we have an infinite sequence of elements of type $t\ D$. Hence there exist two elements $p\ (unroll\ t_i)$ and $p\ (unroll\ t_j)$ such that they are related in the WQO $f\ (fixT\ kids\ p\ f)$. By the definition of *fixT*, this contradicts the initial assumption that the sequence $\bar{t}^{\infty}$ is accepted by *fixT kids p f*. □

Our proof is essentially a proof of the Tree Theorem [Kruskal, 1960] to our setting, though the proof itself follows the simpler scheme in Nash-Williams [1963].

Generality is good, but the calibrator has an complex type which may be somewhat hard for Haskell programmers to check. In the next section we show how *kids* and the calibrator p can be written generically, and hence can be entirely eliminated from the preconditions for *fixT*.

Further remarks on lists Inlining our combinators and simplifying, we find that our earlier definition of *listT* is equivalent to the following:

```
listT′ :: TTest a → TTest [a]
listT′ (WQO (⊴)) = WQO go
  where
    go (x : xs) (y : ys)
      | x ⊴ y, go xs ys = True
      | otherwise       = go (x : xs) ys
    go (_: _)  []       = False
    go []      []       = True
    go []      (_: ys)  = go [] ys
```

It is interesting to note that *listT′* could be more efficient:

- By noticing that $\forall ys.go\ [\,]\ ys = True$, the last clause of *go* can be replaced with $go\ [\,]\ (\_ : ys) = True$. This avoids a redundant deconstruction of the list in the second argument (at the cost of changing the meaning if the second argument is in fact infinite—a possibility we explicitly excluded when defining *fixT*).

- By noticing that $\forall x, xs, ys.go\ (x : xs)\ ys \implies go\ xs\ ys$, the first clause of *go* can avoid falling through to test $go\ (x : xs)\ ys$ if it finds that $go\ xs\ ys \equiv False$.

Both of these observations are specific to the special case of lists: for other data types (such as binary trees) *fixT* will generate an implementation that does not have any opportunity to apply these "obvious" improvements.

D.4.3 From functors to *Traversable*s

As we have presented it, the user of *fixT* still has the responsibility of providing a correct *kids* and a calibrator p with a strange dependent type (which Haskell does not even support!). Happily, we can greatly simplify things by combining the recently-added ability of GHC to automatically derive *Traversable* instances. The *Traversable* [Gibbons and d. S. Oliveira, 2009] type class allows us to write the following:

```
kids_traverse :: ∀t a.Traversable t ⇒ t a → [a]
kids_traverse = unGather.traverse (λx. Gather [x])
newtype Gather a b = Gather {unGather :: [a]}
```

```
instance Functor (Gather a) where
  fmap _ (Gather xs) = Gather xs
instance Applicative (Gather a) where
  pure x = Gather [ ]
  Gather xs ⟨*⟩ Gather ys = Gather (xs ++ys)
```

It follows from the *Traversable* laws that $kids_{traverse}$ collects "all the children" of *t rec*, and as a consequence (See Section 4.1 of [Gibbons and d. S. Oliveira, 2009]) the corresponding projector is just *id*. We can therefore satisfy the preconditions of Definition D.4.1 by setting:

$$\begin{aligned} kids &:= kids_{traverse} \\ p &:= id \end{aligned}$$

The corresponding generic definition *gfixT* becomes:

```
gfixT :: Traversable t
      ⇒ (∀rec.TTest rec → TTest (t rec))
      → TTest (Fix t)
gfixT = fixT kids_traverse id
```

Therefore, if the user of the library has a correct *Traversable* instance (possibly compiler-generated), they need not worry about the calibrator or *kids* functions at all, and cannot violate the safety guarantees of the library.

D.5 Optimisation opportunities

Having defined our combinators, we pause here to consider two optimisations we can apply to our definitions. Thanks to our clearly-defined abstract interface to the *TTest* and *History* types these optimisations are entirely transparent to the user.

D.5.1 Pruning histories using transitivity

In this section we consider an improvement to the definition of *initHistory* in Section D.2.1.

Normally, whenever a *History a* receives a new element $x' :: a$ to compare against its existing $\overline{x}^n$, we test all elements to see if $\exists i < n. x_i \trianglelefteq x'$. If we do not find such an i, we append x' to form the new sequence $\overline{x'}^{n+1}$ which will be tested against subsequently. Thus at every step the number of tests that need to be done grows by one.

There is an interesting possibility for optimisation here: we may in fact exclude from $\overline{x'}$ any element x_j $(0 \leq j < n)$ such that $x' \trianglelefteq x_j$. The reason is that if a later element $x'' :: a$ is tested against $\overline{x'}$, then by transitivity of $\trianglelefteq$, $x_j \trianglelefteq x'' \implies x' \trianglelefteq x''$—thus it is sufficient to test x'' only against x', skipping the test against the "older" element x_j entirely.

To actually make use of this optimisation in our implementation, our implementation must (for all $0 \leq j < n$), test $x' \trianglelefteq x_j$ as well as $x_j \trianglelefteq x'$. To make this test more efficient, we could redefine *TTest* so when evaluated on x and y it returns a pair of *Bool* representing $x \trianglelefteq y$ and $y \trianglelefteq x$ respectively:

```
data TTest a = WQO {(⊴) :: a → a → (Bool, Bool)}
```

Returning a pair of results improves efficiency because there is almost always significant work to be shared across the two "directions".

A version of the core data types improved by this new *TTest* representation and the transitivity optimisation is sketched below:

```
data TTest a = WQO (a → a → (Bool, Bool))
newtype History a = H {test :: a → TestResult a}
initHistory :: ∀a.TTest a → History a
initHistory (WQO (⊴)) = H (go [ ])
  where
    go :: [a] → a → TestResult a
    go xs x
      | or gts      = Stop
      | otherwise = Continue (H (go (x : [x | (False, x) ← lts `zip` xs])))
        where (gts, lts) = unzip (map (⊴ x) xs)
```

It is unproblematic to redefine all of our later combinators for the elaborated *TTest* type so we can take advantage of this transitivity optimisation.

D.5.2 Making *cofmap* more efficient

The alert reader may wonder about how efficient the definition of *cofmap* in Section D.3.4 is. Every use of a WQO of the form *cofmap f wqo* will run *f* afresh on each of the two arguments to the WQO. This behaviour might lead to a lot of redundant work in the implementation of *test* (Section D.2.1), as repeated uses of *test* will repeatedly invoke the WQO with the same first argument. By a change of representation inside the library, we can help ensure that this per-argument work is cached and hence only performed once for each value presented to *test*:

```
data TTest a where
  TT :: (a → b) → (b → b → Bool) → TTest a
newtype History a = H {test :: a → TestResult a}
initHistory :: TTest a → History a
initHistory (TT f (⊴)) = H (go [ ])
  where
    go fxs x
      | any (⊴ fx) fxs = Stop
      | otherwise       = Continue (H (go (fx : fxs)))
     where fx = f x
```

A *History* now includes a function *f* mapping the client's data *a* to the maintained history list [*b*]. When testing, we apply the function to get a value *fx* :: *b*, which we compare with the values seen so far.

With this new representation of *TTest*, *cofmap* may be defined as follows:

```
instance Cofunctor TTest where
  cofmap f (WQO prep (⊴)) = WQO (prep.f) (⊴)
```

The ability to redefine *TTest* to be more than simply a WQO is one of the reasons why we distinguish "termination tests", which the client builds using the combinators, and "WQOs" which are part of the implementation of a termination test, and are hidden from the client.

All the *TTest*-using code we present is easily adapted for the above, more efficient, representation of *TTest*. Furthermore, this technique can further be combined with the optimisation described in Section D.5.1 with no difficulties.

D.6 Supercompilation termination tests

Now that we have defined a combinator library for termination tests, you might wonder whether it is actually general enough to capture those tests of interest in supercompilation. In this section, we demonstrate that this is so.

D.6.1 Terminating evaluators

Before we discuss those well-quasi-orders used for supercompilation, we would like to motivate them with an example of their use.

A supercompiler is, at its heart, an evaluator, and as such it implements the operational semantics for the language being supercompiled. However, the language in question is usually Turing complete, and we would like our supercompiler to terminate on all inputs—therefore, a termination test is required to control the amount of evaluation we perform. We would like to evaluate as much as possible (so the test should be lenient). Equally, if evaluation appears to start looping without achieving any simplification, then we would like to stop evaluating promptly (so the test should be vigilant).

Clearly, any test of this form will prevent us reducing some genuinely terminating terms to values (due to the Halting Problem), so all we can hope for is an approximation which does well in practice.

Concretely, let us say that we have a small-step evaluator for some language:

```
step :: Exp → Maybe Exp
```

The small-step evaluator is a partial function because some terms are already values, and hence are irreducible. Given this small-step semantics we wish to define a big step semantics that evaluates an *Exp* to a value:

```
reduce :: Exp → Exp
```

We would like *reduce* to be guaranteed to execute in finite time. How can we build such a function for a language for which strong normalisation does not hold? Clearly, we cannot, because many terms will never reduce to a value even if *step*ped an infinite number of times. To work around this problem, supercompilers relax the constraints on *reduce*: instead of returning a value, we would like *reduce* to return a value, *except when it looks like we will never reach one.*

Assuming a well-quasi-order *test* :: *TTest Exp* It is easy to define *reduce*:

```
reduce = go (initHistory test)
  where
    go hist e = case hist ‘test‘ e of
      Continue hist′ | Just e′ ← step e → go hist′ e′
      _                                 → e
```

The choice of the *test* well-quasi-order is what determines which heuristic is used for termination. The following three sections demonstrate how our combinators can capture the two most popular choices of termination test: the homeomorphic embedding on syntax trees (used in e.g. Klyuchnikov [2009]; Jonsson and Nordlander [2009]; Hamilton [2007]), and the tag-bag well-quasi-order (used in e.g. Mitchell [2010]; Bolingbroke and Peyton Jones [2010]).

D.6.2 Homeomorphic embedding on syntax trees

The homeomorphic embedding—previous alluded to in Section D.4.1—is a particular relation between (finite) labelled rose trees. The proof that it does indeed define a well-quasi-order is the famous "Tree Theorem" of Kruskal [1960]. We can define it straightforwardly for the *Tree* type using our *gfixT* combinator:

```
type Tree a = Fix (TreeF a)
data TreeF a rec = NodeF a [rec]
                   deriving (Functor, Foldable, Traversable)
node :: a → [Tree a] → Tree a
node x ys = Roll (NodeF x ys)
treeT :: ∀a. TTest a → TTest (Tree a)
treeT wqo_elt = gfixT wqo_fix
  where
    wqo_fix :: ∀rec. TTest rec → TTest (TreeF a rec)
    wqo_fix wqo_subtree
      = cofmap inject (pairT wqo_elt (listT wqo_subtree))
    inject :: ∀rec. TreeF a rec → (a, [rec])
    inject (NodeF x ts) = (x, ts)
```

Now we have *treeT*—the homeomorphic embedding on rose trees—we can straightforwardly reuse it to define a homeomorphic embedding on *syntax* trees. To show how this test can be captured, we first define a simple data type of expressions, *Exp*:

```
data FnName = Map | Foldr | Even
              deriving (Enum, Bounded, Eq)


instance Finite FnName where
  elements = [minBound .. maxBound]
data Exp = FnVar FnName | Var String
         | App Exp Exp | Lam String Exp
         | Let String Exp Exp
```

As is standard, we identify a finite set of function names *FnName* that occur in the program to be supercompiled, distinct from the set of variables bound by lambdas or **let**s. The purpose of this distinction is that we usually wish that $\neg(map \trianglelefteq foldr)$ but (since we assume an infinite supply of bound variables) we need that $x \trianglelefteq y$ within $\lambda x \to x \trianglelefteq \lambda y \to y$.

Our goal is to define a termination test *test1* :: *TTest Exp*. We proceed as follows:

```
data Node = FnVarN FnName | VarN
          | AppN | LamN | LetN
          deriving (Eq)
instance Finite Node where
  elements = VarN : AppN : LamN : LetN :
             map FnVarN elements
test1 :: TTest Exp
test1 = cofmap inject (treeT finiteT)
  where
    inject (FnVar x)    = node (FnVarN x) [ ]
inject (Var _)        = node VarN []
inject (App e1 e2)    = node AppN [inject e1, inject e2]
inject (Lam _ e)      = node LamN [inject e]
inject (Let _ e1 e2)  = node LetN [inject e1, inject e2]
```

The correctness of the *Finite Node* predicate is easy to verify, and thus this termination test is indeed a WQO. This test captures the standard use of the homeomorphic embedding in supercompilation.

More typically, the *FnName* data type will be a string, and the supercompiler will ensure that in any one execution of the supercompiler only a finite number of strings (the function names defined at the top level of the program to supercompile) will be placed into a *FnVar* constructor. In this case, the code for the termination test remains unchanged—but it is up to the supercompiler programmer to ensure that the new **instance** *Finite Node* declaration is justified.

D.6.3 Quasi-ordering tagged syntax trees

Observing that typical supercompiler implementations spent most of their time testing the termination criteria, Mitchell [2010] proposed a simpler termination test based on "tag bags". Our combinators are sufficient to capture this test, as we will demonstrate.

The idea of tags is that the syntax tree of the initial program has every node tagged with a unique number. As supercompilation proceeds, new syntax trees derived from the input syntax tree are created. This new syntax tree contains tags that may be copied and moved relative to their position in the original tree—but crucially the supercompiler will never tag a node with a totally new tag that comes "out of thin air". This property means that in any one run of the supercompiler we can assume that there are a finite number of tags.

We first require a type for these tags, for which we reuse Haskell's *Int* type. Crucially, *Int* is a bounded integer type (unlike *Integer*), so we can safely make the claim that *Tag* is *Finite*:

```
newtype Tag = Tag {unTag :: Int} deriving (Eq, Ord)
instance Finite Tag where
  elements = map Tag [minBound .. maxBound]
```

As there are rather a lot of distinct *Int*s, the well-quasi-order *finiteT* :: *TTest Tag* may potentially not reject sequences until they become very large indeed (i.e. it is not very vigilant). In practice, we will only have as many *Int* tags as we have nodes in the input program. Furthermore, most term sequences observed during supercompilation only use a fraction of these possible tags. For these reasons, these long sequences are never a problem in practice.

Continuing, we define the type of syntax trees where each node in the tree has a tag:

```
type TaggedExp = (Tag, TaggedExp′)
data TaggedExp′
  = TFnVar FnName | TVar String
  | TApp TaggedExp TaggedExp
  | TLam String TaggedExp
  | TLet String TaggedExp TaggedExp
```

We also need some utility functions for gathering all the tags from a tagged expression. There are many different subsets of the tags that you may choose to gather—one particular choice that closely follows Mitchell is as follows:

```
type TagBag = Map Tag Int
gather :: TaggedExp → TagBag
gather = go False
  where
    go lazy (tg, e) = singleton tg 1 ‘plus‘ go′ lazy e
    go′ lazy (TFnVar _)     = empty
    go′ lazy (TVar _)       = empty
    go′ lazy (TApp e1 e2)   = go lazy e1 ‘plus‘ go_lazy lazy e2
    go′ lazy (TLam _ e)     = empty
    go′ lazy (TLet _ e1 e2) = go_lazy lazy e1 ‘plus‘ go lazy e2
    go_lazy True (tg, _) = singleton tg 1
    go_lazy False e      = go True e
    plus :: TagBag → TagBag → TagBag
    plus = unionWith (+)
```

We have assumed the following interface for constructing finite maps, with the standard meaning:

```
unionWith :: Ord k ⇒ (v → v → v)
          → Map k v → Map k v → Map k v
empty     :: Ord k ⇒ Map k v
singleton :: Ord k ⇒ k → v → Map k v
```

We can now define the tag-bag termination test of Mitchell [2010] itself, *test2*:

```
test2 :: TTest TaggedExp
test2 = cofmap (summarise.gather)
               (pairT finiteT wellOrderedT)
  where
    summarise :: TagBag → (Set Tag, Int)
    summarise tagbag
      = (keysSet tagbag, sum (elems tagbag))
```

D.6.4 Improved tag bags for tagged syntax trees

In fact, there is a variant of the tag-bag termination test that is more lenient than that of Mitchell [2010]. Observe that the tag bag test as defined above causes the supercompiler to terminate when the domain of the tag bag is equal to a prior one and where the total number of elements in the bag has not decreased. However, since there are a finite number of tags, we can think of a tag-bag as simply a very large (but bounded) arity tuple—so by the Cartesian Product Lemma we need only terminate if *each of the tags considered individually* occur a non-decreasing number of times.

Our more lenient variant of the test can be defined in terms of the *finiteMapT* combinator of Section D.3.7 almost trivially by reusing *gather*. It is straightforward to verify that if the *finiteMap* well-quasi-order relates two maps, those maps have exactly the same domains—so one of the parts of the original tag-bag termination test just falls out:

```
test3 :: TTest TaggedExp
test3 = cofmap gather (finiteMapT wellOrderedT)
```

All three of these termination tests—*test1*, *test2* and *test3*—are sound by construction, and straightforward to define using our library.

Appendix E

Practical considerations

In this appendix, we discuss some of the engineering considerations arising from adding supercompilation to a standard compiler, and the approach our implementation takes to the issues they raise.

E.1 Supercompilation and separate compilation

One major source of engineering challenge in a supercompiler implementation is separate compilation. In this section we discuss the issues relating to the module-by-module compilation method used by GHC.

E.1.1 Supercompiling modules instead of terms

So far, we have only discussed supercompilation of an *expression*, not a whole module: our *supercompile* function has type *Term* → *Term*. In order to use this to supercompile a module, we first have to transform the module into a term. Given a module such as:

```
module FooMod (bar, baz) where
bar = spqr.baz
baz = λxs. map inc xs
spqr = λxs. map dec xs
```

We transform it as follows:

```
module FooMod (bar, baz) where
bar = case res of (bar, _) → bar
baz = case res of (_, baz) → baz
res = let bar = spqr.baz
          baz = λxs. map inc xs
          spqr = λxs. map dec xs
      in (bar, baz)
```

So a program is simply a term, in which the top-level function definitions appear as possibly-recursive let bindings. After this transformation we can then *supercompile* the *expression* on the RHS of the *res* binding.

Note that we did *not* include the unexported function *spqr* in the tuple bound to *res*. It is unnecessary to do so. Furthermore, by omitting it we potentially get better results from supercompilation because the splitter may freely push down even a *non-value* unexported bindings as long as it is transitively used by at most one of the exported bindings, where the unexported binding might be deforested away entirely. In contrast if a non-value unexported binding *spqr* was mentioned in the *res* tuple then we would be forced to residualise it, and it could not possibly be fused together with other bindings in the module.

Controlling supercompilation targets The scheme we describe above has us supercompiling the definition of *every* exported function. In our implementation, we instead make use of a *SUPERCOMPILE* pragma that the user uses to mark those definitions which should be supercompiled. For example, if *spqr* and *baz* in the module above were marked by the pragma, the module would be transformed to:

```
module FooMod (bar, baz) where
spqr = case res of (spqr, _) → spqr
baz = case res of (_, baz) → baz
bar = spqr.baz
res = let baz = λxs. map inc xs
          spqr = λxs. map dec xs
      in (spqr, baz)
```

Where once again the definition of *res* would be supercompiled, so the unmarked *bar* definition would not be supercompiled at all.

As a shorthand, we also allow the whole module to be marked with *SUPERCOMPILE* if you wish to supercompile all exported definitions.

E.1.2 Interface files

When supercompiling, (unlike in control flow analysis [Shivers, 1988]), it is by no means required that all of the code of the program available—dealing with a call to a function whose name is known but code is unknown is no more challenging that dealing with a call to a normal free variable.

However, supercompilation clearly works best when all of the code that can be executed *is* available to the compiler. At first, this may seem incompatible with separate compilation of program modules, where any given invocation of the compiler is only asked to compile a single module, and thus presumably only has access to the code for that module.

In fact, compilers for modern high-level languages (such as Oracle's *javac* Java compiler [Jav]) rarely implement true separate compilation, where modules can be compiled in any order (as is the case for e.g. the GNU C compiler *gcc* [GCC]). Instead, when compiling a module they produce not only object code suitable for execution, but also metadata necessary when compiling dependent modules. (As a result, modules must be compiled in dependency order.)

GHC is no exception in this regard, and it records metadata in a so-called "interface file" which is created as a side-effect of compiling the object file. This metadata may include Core-language definitions of any exported names: these definitions are also called "unfoldings" of those names. This feature is used by GHC to implement cross-module inlining. The key issues surrounding unfoldings are *which* exported names should have definitions recorded, and *what* definition to record. GHC makes the following choice:

- If, after optimisation, an exported name has a definition which is "small" in some sense, and the definition is non-recursive, that definition is recorded in the interface file. These constraints fundamentally reflect the fact that GHC's inlining algorithm will only inline non-recursive definitions where the code-size cost of doing the inlining is compensated for by the benefits of the inlining [Peyton Jones and Marlow, 2002], and so even if definitions for large or recursive functions in imported modules *are* available in the interface file, GHC will never make use of them.

- If an exported name was explicitly marked by the user with an *INLINE* pragma then GHC is much more eager than usual to inline it, and as a result it will always have a definition recorded in the interface file. The twist is that GHC records the *definition as written by the user in the input program*, not the definition as it is after optimisation. One reason for this is that the optimised definition may be much larger than the original definition, and so there will be less immediate code size impact from inlining this smaller original definition into all use sites in other modules than there would be from inlining the larger optimised definition.

Our supercompiler implementation changes these heuristics in that we record post-optimisation definitions for all exported names not marked with the *INLINE* pragma, even if they are recursive. The reason for this is that (unlike GHC's inliner) the supercompiler does not risk non-termination when inlining recursive functions, and by making more definitions available we will have more specialisation opportunity.

Ideally our implementation we would treat the *SUPERCOMPILE* pragma like the *INLINE* pragma in that we would record the definition of the marked function *as written by the user* in the interface file, rather than recording the definition of the function *after supercompilation*. This would prevent any dependent modules which get supercompiled from compounding the problem of supercompilation code explosion by supercompiling already-supercompiled code. However, our implementation does not currently implement this idea. Instead, our tests only mark the *Main* module of the program of interest with the *SUPERCOMPILE* pragma. Since we do not supercompile the libraries, unfoldings of imported names will not themselves be supercompiled.

E.1.3 Sharing specialisations across module boundaries

In our implementation, the accumulated environment of promises is discarded after supercompilation is complete for the module being compiled. However, it would be possible to imagine instead recording all the promises in the corresponding interface file. Then, supercompilation of later modules could start with an environment of promises initialised from the stored promise lists in all imported modules, which would allow those later supercompilation runs to tie back directly to the h-functions created by supercompilation of those imported modules.

This mechanism would allow some the work associated with using supercompilation to specialise library code to be done once and for all when compiling the library, and then reused in all users of the library. This is analogous to GHC's existing *SPECIALISE* pragma, which is used to create specialisations of library functions on type-class dictionaries along with associated rewrite rules [Peyton Jones et al., 2001] that ensure that users of the library make use of the specialisation if one exists.

E.1.4 Making unfoldings available to supercompilation

In Section E.1.2, we described how GHC used unfoldings recorded in interface files to record the definitions of exported names, so that those definitions could be inlined in dependent modules. Given that we have access to these unfoldings, the question arises as to how to expose them to the supercompilation algorithm so that e.g. the evaluator is able to β-reduce calls to imported functions.

The approach we take is to form an initial heap H_{unfold} containing a mapping $x \mapsto e$ for every imported name x if e is the unfolding of x. We speculate this heap, and discard any of the bindings in the final heap which the speculator failed to reduce to values This means that the only bindings that are left are those which are either bind values or bind variables which refer to bindings which bind values.

The speculated, filter heap is wrapped around the initial term to supercompile by the top-level *supercompile* function. The cheapness check ensures that we don't duplicate work, such as could happen if we allowed inlining of an unfolding like $x = \mathit{fib}\ 100$ into all dependent modules.

The scheme as outlined up to this point works fine, but it leads to considerable code duplication because it means that a new copy of every transitively reachable imported function will be created in the module being supercompiled, even if those functions end up being completely unspecialised. Ideally, we would like to reuse the existing unspecialised code for the imported functions if possible.

In order to achieve this, we *preinitialise the memoiser's state*: before we begin supercompilation, for every imported name x with unfolding e we record a promise which says that the state $\langle H_{\mathit{unfold}} \,|\, e \,|\, \epsilon \rangle$ can be tied back to the "h function" x. In fact, in order to share more code we will create such a promise for all η-expansions of the unfolding, so for an imported name f with an unfolding $\lambda x\ y.\, e$ we will create promises for:

- **let** $f = \lambda x\ y.\, e$ **in** f, tying back to f
- **let** $f = \lambda x\ y.\, e$ **in** $f\ x$, tying back to $f\ x$
- **let** $f = \lambda x\ y.\, e$ **in** $f\ x\ y$, tying back to $f\ x\ y$
- $\lambda x\ y.\, e$, tying back to f
- $\lambda y.\, e$, tying back to $f\ x$
- e, tying back to $f\ x\ y$

This means that when the algorithm tries to supercompile a state which matches one of the imported functions, it just ties back to the existing imported name for that function rather than creating a new copy.

Memoiser preinitialisation can potentially cause a loss of optimisation if the exported function was not fully optimised by the earlier compiler invocation which compiled the module in which the imported functions were defined. For example, we could supercompile a module M which depends on a module *Library*:

> **module** *Library* (*mapmap*) **where**
> *mapmap f g* = *map f* .*map g*

If we did not mark *Library* with *SUPERCOMPILE*, or compiled it with a flag to turn off supercompilation, then the generated machine code for *mapmap* may allocate an intermediate list. With memoiser preinitalisation, if we import the *mapmap* function in module M which makes the call *mapmap f g*, then we will just reuse that inefficient exported code rather than supercompiling *mapmap* anew and hence achieving deforestation.

We are happy to use memoiser preinitalisation (and hence risk this problem) because we make the assumption that previously compiled modules have already optimised as much as possible given the information they had available. With this philosophy, the job of the supercompiler is just to try to optimise given the *new* information we have in the current module being supercompiled.

E.1.5 Unavailable unfoldings

Sometimes, a name may be imported by the program being supercompiled, but we do not have an available unfolding that we can add to H_{unfold}. This may be because the unfolding is non-cheap, because unfolding is explicitly disallowed by a *NOINLINE* pragma, or an unfolding simply was not recorded in the corresponding interface file (even if we modify the compiler to record unfoldings for all exports when it creates an interface file, unfoldings may be missing due to loops in the module dependency graph).

For such an unfoldingless name x, the supercompilation algorithm as described will λ-abstract *every* h-function whose corresponding state mentions x over the x variable, as it will regard x as a free variable. So if for some reason there is no unfolding available for the library function *runST*, the following module:

```
module Looper (f) where
g n = ...
f n = case n of Z   → [ ]
                S m → runST (g m) : f m
```

Will be transformed into a program like the following:

```
module Looper (f) where
f = let h0 runST = let f n = h1 runST n
                   in f
        h1 runST n = case n of Z   → [ ]
                               S m → h2 m : h1 runST m
        h2 m = ...
    in h0 runST
```

This program completely hides the fact that *runST* is an imported name from each of its use sites. This is a critical problem because even if we aren't able to inline the definition of *runST* at those use sites, the machine code generated by GHC for a call to a function whose definition site is known is far more efficient than what it generates for a call to an λ-bound function with unknown definition [Marlow and Peyton Jones, 2004]. Furthermore, *runST* might have associated rewrite rules [Peyton Jones et al., 2001] which can be exploited by GHC's optimisation mechanisms: if we λ-abstract over *runST* then these rules cannot be used.

One approach to fixing this problem would be to apply the static-argument transformation [Santos, 1995] to the supercompiled code, and hope that it manages to discover all such problems and undo them. However, we prefer a more robust solution which avoids introducing these spurious λ-abstractions in the first place.

The approach we take is to use a special form of "dummy" heap binding, which we write $x : \tau \mapsto \mathbf{let}^t$, to record which variables should actually bound by imported names rather than λ-abstractions in the corresponding h-function. These heap bindings are useless to the evaluator since they do not include any information about the definition of the corresponding name, but they prevent the variable x that they bind from being reported as free for the purposes of deciding what variables the h-function should be abstracted over. With this technique we can drive our example above in an initial heap containing the binding $runST : \forall a.ST\ a \to a \mapsto \mathbf{let}^t$ for some tag t, and hence can obtain the desired output program for our example above:

```
module Looper (f) where
f = let h0 = let f n = h1 n
```

```
            in f
        h1 n = case n of Z    → [ ]
                         S m → h2 m : h1 m
        h2 m = ...
    in h0
```

It is important to note that the variables bound by these dummy heap bindings cannot be freely renamed, because they are really variable *occurrences* rather than binders. This is particularly important to consider when modifying *msg* to deal with **let**-marked variables.

To see why, observe that we cannot *msg* $\mathcal{S}_0 = \langle runST : \forall a.ST\ a \to a \mapsto \textbf{let} \mid runST \mid \epsilon \rangle$ (where *runST* is **let**-marked) and $\mathcal{S}_1 = \langle \epsilon \mid x \mid \epsilon \rangle$ to the generalised state $\langle \epsilon \mid y \mid \epsilon \rangle$. If *msg* did behave that way, it would cause our *match* implementation to claim that you can tie back the state $\mathcal{S}_1$ to an earlier promise for the state $\mathcal{S}_0$ if you use the substitution $\{runST \mapsto x\}$, which is not in fact true because the h-function for $\mathcal{S}_0$ would not have been λ-abstracted over *runST*. Our implementation of *msg* therefore only MSGs together references to **let**-marked variables if they have *exactly* the same bound name on both the left and right, reflecting the fact that they both refer to the exact same imported name.

Note that the new form of heap binding has an associated tag, just like a normal one. This is useful because it means that the termination test will not fail if we happen to supercompile two states that differ only in the imported names to which they refer.

E.2 Primops

Primitive operations such as the arithmetic operators (+) and numeric literals ℓ can be easily incorporated into the Core language as extra expression productions: $e ::= \ldots \mid d \oplus d \mid \ell$. The operational semantics can be likewise extended to evaluate primops, with the syntax of the abstract machine growing to incorporate new types of stack frame: $\kappa ::= \ldots \mid \bullet \oplus d \mid u \oplus \bullet$. When reducing a state such as $\langle \epsilon \mid 20^{t_2} \mid 10^{t_1} \oplus \bullet^{t_0} \rangle$, a choice exists as to which tag should be used for the result of executing the primop, and the choice we arbitrarily make is to use the tag of the primop itself, so one of the new rules of the extended operational semantics is:

$$\left\langle H \,\middle|\, \ell_2^{t_2} \,\middle|\, \left(\ell_1^{t_1} \otimes \bullet\right)^{t_0}, K \right\rangle \rightsquigarrow \left\langle H \,\middle|\, (\otimes(\ell_1, \ell_2))^{t_0} \,\middle|\, K \right\rangle$$

All the new syntax can be handled in the splitter and MSG/matcher straightforwardly. However, there is something interesting about primops in supercompilation: their termination treatment. A standard supercompiler that uses a homeomorphic embedding will typically make a choice between:

- Not executing primops at compile time, or
- Treating all literals as identical for the purposes of the termination test (i.e. having the homeomorphic embedding embed all literals into each other)

The reason for this choice is that if primops are not executed at compile time, you may assume that the supercompiler will only encounter a fixed finite set of literals during supercompilation of any given program (i.e. those literals that were syntactically present in that program), and so distinct literals may be treated as distinct for termination purposes.

Because our termination test is tag-based, our supercompiler takes a compromise position where literals that occurred syntactically in the input program are treated as distinct from each other, while any literals generated by executing primops are treated by the termination test as being identical to one of the literal arguments to the primop. This position allows us to avoid failure of the termination test for longer in programs using literals, but still execute primops at compile time.

E.2.1 Interaction between primops and generalisation

One subtle point about primops is that they interact poorly with the combination of speculation (Section 6.3) and growing-tags generalisation (Section 6.2.2). A simple example of the problem occurs when supercompiling a version of *length* defined with the non-strict *foldl* defined in Section 6.2. Consider this term:

$$\textbf{let}\ \mathit{next}\ \_\ n = n + 1; z = 0^{t_e}\ \textbf{in}\ ((\mathit{foldl}^{t_a}\ \mathit{next}^{t_b}\ z)^{t_c}\ \mathit{xs})^{t_d}$$

Supercompiling, we see the following sequence of normalised *States*:

$$\begin{array}{lcl}
\mathcal{S}_1 & = & \left\langle H, z \mapsto 0^{t_e} \,\middle|\, \mathit{foldl}^{t_8} \,\middle|\, (\bullet\ \mathit{next})^{t_b}, (\bullet\ z)^{t_c}, (\bullet\ \mathit{xs})^{t_d} \right\rangle \\
\mathcal{S}_2 & = & \left\langle H, z \mapsto 0^{t_e}, \mathit{n1} \mapsto (\mathit{next}\ y\ z)^{t_4} \,\middle|\, \mathit{foldl}^{t_8} \,\middle|\, (\bullet\ \mathit{next})^{t_3}, (\bullet\ n1)^{t_5}, (\bullet\ \mathit{ys})^{t_6} \right\rangle \\
\mathcal{S}_3 & = & \left\langle H, \mathit{n1} \mapsto 1^{t_4}, \mathit{n2} \mapsto (\mathit{next}\ \mathit{y1}\ \mathit{n1})^{t_4} \,\middle|\, \mathit{foldl}^{t_8} \,\middle|\, (\bullet\ \mathit{next})^{t_3}, (\bullet\ n2)^{t_5}, (\bullet\ \mathit{ys}1)^{t_6} \right\rangle \\
\mathcal{S}_4 & = & \left\langle H, \mathit{n2} \mapsto 2^{t_4}, \mathit{n3} \mapsto (\mathit{next}\ \mathit{y2}\ \mathit{n2})^{t_4} \,\middle|\, \mathit{foldl}^{t_8} \,\middle|\, (\bullet\ \mathit{next})^{t_3}, (\bullet\ n3)^{t_5}, (\bullet\ \mathit{ys}2)^{t_6} \right\rangle
\end{array}$$

Up until $\mathcal{S}_4$, supercompilation has proceeded untroubled by termination test failure. This changes when we reach $\mathcal{S}_4$, as the latest tag bag is identical to that for $\mathcal{S}_3$ and so to ensure termination the supercompiler has to *generalise* or *split* without reduction. Unfortunately, because we have two *equal* tag-bags, the supercompiler will be unable to identify a growing tag suitable for generalisation, and so will have to fall back on standard *split* rather than generalising away the n heap bindings as we would hope.

If we had not speculated, we would not have the problem as *n1* and *n2* would remain unreduced. This would have turn caused the heap of $\mathcal{S}_3$ to contain z—thus triggering the termination test at the point we supercompiled $\mathcal{S}_3$, and fingering t_4 as the growing tag. Likewise, the problem would not occur if we had Peano numbers in place of literals and primops, because the growing accumulator would have shown up as an increasingly deeply nested application of S that is readily detectable by the growing-tags heuristic.

E.2.2 Multiplicity-tagging for better primop generalisation

Our implementation implements a solution to this problem which is straightforward and non-invasive. The idea is that a binding such as *n2* should "weigh" more heavily in the mind of the growing-tags generaliser than a binding like *n1* because its literal value embodies more computational history. To this end, we add a multiplicity to the type of tags, modelling a tag-bag with several repetitions of a single tag, vaguely approximating the effect of Peano arithmetic:

type *Multiplicity* = *Int*
type *Tag* = (*Int*, *Multiplicity*)

Tags are initialised with an *Multiplicity* of 1. The tagged reduction rules for Core remain largely the same, but rules which execute primops change to sum multiplicities:

$$\left\langle H \,\middle|\, \ell_2^{(x_2,c_2)} \,\middle|\, \left(\ell_1^{(x_1,c_1)} \otimes \bullet\right)^{(x_0,c_0)}, K \right\rangle \rightsquigarrow \left\langle H \,\middle|\, (\otimes(\ell_1,\ell_2))^{(x_0,c_0+c_1+c_2)} \,\middle|\, K \right\rangle$$

This rule differs from the standard one in that it produces an output tag with an *Multiplicity* that has grown to be strictly greater than the tags of each of the inputs.

The final, crucial step is to make use of the *Multiplicity* information when constructing the *TagBag* for the termination test (Section 3.2):

$$\begin{array}{rcl} \mathit{tagBag}\,\langle H \,|\, e^{(x,c)} \,|\, K \rangle & = & \{\!\{ x \,|\, x \mapsto e^{(x,c)} \in H, 0 \leq i < c \}\!\} \\ & \cup & \{\!\{ x \,| 0 \leq i < c \}\!\} \\ & \cup & \{\!\{ x \,|\, \kappa^{(x,c)} \in K, 0 \leq i < c \}\!\} \end{array}$$

No further changes are required—because we construct *TagBag*s with several "copies" of a tag with an *Multiplicity* greater than one, the growing tags heuristic identifies them as growing in just the situations where we need it to. Consider our *length* function example above. In $\mathcal{S}_3$, the tag t_4 on *n1* will have an multiplicity of 3, and that on *n2* will be 1. By the time we reach $\mathcal{S}_4$ the tag t_4 on *n2* will have a count of 5 while the tag on *n3* will be 1. Looking at the resulting tag bags, it is clear that t_4 is the growing tag and hence should serve as the basis of residualisation by the growing-tags heuristic.

E.3 Differences between Core and GHC Core

As our supercompiler has been implemented as a part of GHC, it operates on GHC's own internal Core language which is different to the Core language we describe in Section 2.1 and use throughout the thesis. In this section, we will summarise the differences and describe how they affect the supercompilation process.

E.3.1 Unlifted types

GHC's core language separates base types (i.e. those of a non-arrow kind) into two kinds rather than the conventional single kind $\star$ that we use. In GHC's system, types of kind $\star$ are called *lifted* types and may be inhabited by $\bot$, as usual, but types of kind # are called *unlifted* types and may not be. This is used to define types such as the unlifted 32-bit integer type *Int32#* which correspond closely to the type of values that can be held in a single machine register, and hence manipulated very efficiently without the burden of heap lookup that is required by usual implementations of laziness on stock hardware [Launchbury and Paterson, 1996; Jones and Launchbury, 1991].

There are two principle consequences to the supercompiler from this feature.

Unlifted let bindings A (non-recursive) let binding may bind a value of unlifted type, in which case the usual **let** evaluation rule of Section 3.3 is incorrect. Instead, the normalising small-step operational semantics must include new evaluation rules and corresponding stack frame:

$$\begin{array}{rcl} \left\langle H \,\middle|\, (\mathbf{let}\ x{:}\tau\ =\ d_x\ \mathbf{in}\ d)^t \,\middle|\, K \right\rangle & \rightsquigarrow & \langle H \,|\, d_x \,|\, \mathbf{let}\ x{:}\tau\ = \bullet\ \mathbf{in}\ d^t, K \rangle \\ \langle H \,|\, v \,|\, \mathbf{let}\ x{:}\tau\ = \bullet\ \mathbf{in}\ d^t, K \rangle & \rightsquigarrow & \langle x{:}\tau \mapsto v^t, H \,|\, d \,|\, K \rangle \\ \langle H \,|\, v \,|\, \mathbf{update}\ y{:}\tau, \mathbf{let}\ x{:}\tau\ = \bullet\ \mathbf{in}\ d^t, K \rangle & \rightsquigarrow & \langle x{:}\tau \mapsto v^t, y{:}\tau \mapsto x, H \,|\, d \,|\, K \rangle \end{array}$$

The new strict-**let** stack frame has to be handled everywhere in the supercompiler, but is generally straightforward to deal with. Of particular interest is its interaction with the splitter, where it should be treated similarly to a single-branch **case** frame, including the potential to have following stack frames inlined into the "branch" of the strict-**let** frame.

One other awkward issue arises with unlifted **let** bindings: because the heap may contain a mix of lifted and unlifted bindings, we can no longer bind heap bindings with a simple single recursive **let** when residualising a state. Instead, we must perform a strongly-connected-components analysis on the heap and then use a non-recursive **let** to bind non-recursive SCCs (any unlifted heap bindings will become a non-recursive SCC), and recursive **let**s to bind recursive SCCs.

Non-abstractability of type information Normally, if we were to compute the most-specific generalisation (Section 5.1) of the two terms $\lambda(x :: \mathit{Int}).\, x$ and $\lambda(x :: \mathit{Bool}).\, x$ we would derive the type-generalised term $\lambda(x :: a).\, x$. At this point, we would be justified (by Theorem 5.7.1) in performing a transformation such as taking this program:

$$\begin{array}{l} \textbf{let}\, f = \lambda(x :: \mathit{Int}).\, x \\ \quad\;\; g = \lambda(x :: \mathit{Bool}).\, x \\ \textbf{in}\,(f, g) \end{array}$$

And replacing it with this:

$$\begin{array}{l} \textbf{let}\, f = h\ \mathit{Int} \\ \quad\;\; g = h\ \mathit{Bool} \\ \quad\;\; h = \Lambda(a :: \star).\, \lambda(x :: a).\, x \\ \textbf{in}\,(f, g) \end{array}$$

However, unlifted types complicate things. We *cannot* replace this program:

$$\begin{array}{l} \textbf{let}\, f = \lambda(x :: \mathit{Int32\#}).\, x \\ \quad\;\; g = \lambda(x :: \mathit{Int16\#}).\, x \\ \textbf{in}\,(f, g) \end{array}$$

With this one:

$$\begin{array}{l} \textbf{let}\, f = h\ \mathit{Int32\#} \\ \quad\;\; g = h\ \mathit{Int16\#} \\ \quad\;\; h = \Lambda(a :: \#).\, \lambda(x :: a).\, x \\ \textbf{in}\,(f, g) \end{array}$$

The reason is that unlifted types will generally also be unboxed, so the code that manipulates values of unlifted types may not be polymorphic in the unlifted type. This means we may never abstract over a type variable of a kind that might ever end up constructing an unlifted type, so abstraction of the form $\Lambda(a :: \#).\, e$, $\Lambda(a :: \star \to \#).\, e$, $\Lambda(a :: (\star \to \star) \to \#).\, e$ and so on must be disallowed. Avoiding inadvertently creating an abstraction of this form requires an ad-hoc check in the type MSG (Section 5.8).

E.3.2 Wildcard case branches

In GHC, **case** expressions need not exhaustively list all of the possible constructors that the scrutinee may evaluate to if it does not need to. Instead, it is allowed to include a single wildcard branch that is used if none of the explicitly listed constructors match. For example, the following evaluates to *False*:

$$\textbf{case}\ \mathit{True}\ \textbf{of}\ \mathit{False} \to \mathit{True}; \_ \to \mathit{False}$$

This poses no problem for the supercompiler, but it can hurt the quality of positive information that is propagated, since if we have a residual **case** branch that binds a wildcard there is no way for a positive supercompiler to express the information we have learned about the scrutinee. For example, with naive handling of wildcard branches the following program:

$$\textbf{case}\ x\ \textbf{of}\ \_ \to \textbf{case}\ x\ \textbf{of}\ \mathit{False} \to \mathit{True}; \mathit{True} \to \mathit{False}$$

Will not be further optimised, even though it is easy to see that we need only scrutinise x once.

To prevent the issues introduced by wildcards, we could potentially perform a pre-pass on the program before supercompilation to expand wildcard branches to an exhaustive list:

$$\begin{aligned}&\textbf{case}\ x\ \textbf{of}\ \mathit{True} \to \textbf{case}\ x\ \textbf{of}\ \mathit{False} \to \mathit{True}; \mathit{True} \to \mathit{False}\\ &\qquad\qquad\ \mathit{False} \to \textbf{case}\ x\ \textbf{of}\ \mathit{False} \to \mathit{True}; \mathit{True} \to \mathit{False}\end{aligned}$$

After supercompiling this expanded program, positive information propagation will ensure we get the desired result—a program that scrutinises x only once:

$$\textbf{case}\ x\ \textbf{of}\ \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True}$$

Unfortunately, this expansion process is not practical because for it may increase the size of the program by a multiplicative factor of m^n, where m in the size of the largest algebraic data family scrutinised in the program and n the depth of the deepest case nest in the program. An alternative scheme, "λ-lifting" the wildcard branch to a new function abstracted over the scrutinised variable, is more practical. If using that scheme, our expanded example program would look as follows:

$$\begin{aligned}&\textbf{let}\ k\ x = \textbf{case}\ x\ \textbf{of}\ \mathit{False} \to \mathit{True}; \mathit{True} \to \mathit{False}\\ &\textbf{in}\ \textbf{case}\ x\ \textbf{of}\ \mathit{True} \to k\ \mathit{True}; \mathit{False} \to k\ \mathit{False}\end{aligned}$$

Although this solves the *input program* code size explosion problem, it is still often not desirable because of the tendency of supercompilation to specialise terms with regards to information even when that information gives no tangible runtime benefit: the output of supercompilation on programs treated this way is almost always much larger with no runtime gain at all.

Our implementation compromises by expanding only those wildcards which would be replaced by at most one new branch[1], which is occasionally effective at improving positive information propagation and never increases the size of the input program.

E.3.3 Scrutinee binder of case

In GHC, the scrutinee of a **case** is bound to a variable that may occur freely in any of the branches of the case. For example, the following program will evaluate to *True*:

$$\textbf{case}\ \mathit{True}\ \textbf{of}\ x\ \{\mathit{True} \to x; \mathit{False} \to \mathit{False}\}$$

This feature is unproblematic for the supercompiler except that the scrutinee binder must also be considered when propagating positive information.

E.3.4 Polymorphic *seq*

In GHC, **case** expressions are allowed to scrutinise values of types other than algebraic data types, in which case they may have at most one branch, and that branch will always be a wildcard branch.

Such **case** expressions pose few problems: they can be treated exactly as a standard **case** except that (as usual for wildcard branches) no positive information can be gleaned from them.

[1] When scrutinising a GADT, type information can sometimes be used to reduce the number of explicit branches that a wildcard branch expands to.

E.3.5 Type lambdas are not values

In GHC's core language, type lambdas are entirely erased before generated code, and so they cannot be values. However, in *Core* we treat type lambdas as values. The difference is observable in GHC core, where it is possible to do a *seq* that is not only polymorphic but also "impredicative"! In particular, the following GHC core program should diverge instead of returning *True*:

$$\begin{array}{l}\mathbf{let}\, f = \Lambda(a :: *).\, f\ Int \\ \mathbf{in\ case}\, f\ \mathbf{of}\ \_ \to True\end{array}$$

Unfortunately, our supercompiler optimises this program to a simple *True* because it believes f to be a value. This is a correctness bug, as the supercompiler should not make a program "more defined". We accept this as a known bug in the supercompiler because:

- Although it is very important for practical use that the supercompiler preserves standard polymorphic *seq*s, *seq*s on *terms of polymorphic type* are almost nonexistent in practice, so the bug is unlikely to hurt anyone.
- Fixing the problem (by modifying the evaluator to evaluate underneath type lambdas and changing the definition of values to include only $\Lambda\alpha{:}\kappa.\, v$ rather than $\Lambda\alpha{:}\kappa.\, e$) imposes a small but significant cost on the implementation.

E.3.6 Unsaturated data constructors

In GHC's core language, data constructors may occur saturated, but Core expects data constructors to be saturated. Therefore, when we convert a Haskell program into our Core language, we have to introduce wrappers to deal possible partial applications. For example, the following GHC program:

$$((:), (:)\ x, x : xs)$$

Will be translated into the following Core:

$$\begin{array}{l}\mathbf{let}\ (:)_{\text{wrap}} = (\lambda y.\, (\lambda ys.\, (y : ys)^{t_2})^{t_1})^{t_0} \\ \quad a1\ = (:)^{t_4}_{\text{wrap}}\ x^{t_3} \\ \quad a2\ = ((:)^{t_7}_{\text{wrap}}\ x^{t_6}\ xs)^{t_5} \\ \mathbf{in}\ ((:)_{\text{wrap}}, a1, a2)^{t_8}\end{array}$$

Notice that after β-reducing uses of the data constructor wrappers, all saturated occurrences of data constructors in the input program will have the *same* tag: for example, in the above program all saturated occurrences of the (:) data constructor will have tag t_2.

This is a desirable property to have, because it means that the termination test will be better able to spot functions that are e.g. being specialised multiple times on the same data constructor argument, even when those data constructor originate from distinct places in the input program.

E.3.7 Coercions

GHC's core language is based on System FC [Sulzmann et al., 2007; Vytiniotis et al., 2012], whose principal interesting difference from System Fω is the presence of *coercions* in the syntax of expressions:

$$e ::= \ldots \mid e \triangleright \varphi \mid \varphi$$

Without going into too much detail, suffice to say that a coercion φ of type $\tau \sim \upsilon$ is passed around as evidence that the runtime representation for the type τ and υ are identical, and a case $e \triangleright \varphi$ performs an unchecked cast from τ (the type of e) to υ. A cast is intended to be implemented at runtime by a no-op.

Coercions are integral to the encoding of some of GHC's most cutting-edge features in GHC's core language in a way which still allows the core to be trivially type checked and inferred, but they are also used to implement features as humble as the **newtype** declaration. For this reason, they tend to be ubiquitous in programs of practical interest and cannot be ignored, which is unfortunate as they impose considerable costs on the supercompiler implementation:

- The definition of a value has to take into account coercions: $v ::= \ldots \mid v \triangleright \varphi \mid \varphi$
- Positive information propagation must be prepared to propagate information about scrutinees that are not simple variables but possibly variables that have been cast by one or more coercions.
- Most importantly, the normalising small-step operational semantics has to include a new stack frame for casts, as well as modify all existing reduction rules to be able to deal with (potentially several consecutive) casts on the stack between the context and the stack frames of real interest (e.g. update frames or application frames).

E.3.8 Kind polymorphism

Recent versions of GHC [Yorgey et al., 2012] use an intermediate language that allows kinds to be abstracted over, making programs such as the following possible ($\square$ is the (unique) superkind i.e. all kinds are classified by $\square$):

$$\begin{array}{l}\textbf{let}\, f = \Lambda(k :: \square).\, \Lambda(a :: k).\, 1 \\ \textbf{in}\, f\ {*}\ \mathit{Int} + f\ ({*} \to {*})\ \mathit{Maybe}\end{array}$$

Kind abstraction poses few problems for the supercompiler, and can be handled almost entirely analogously to type abstraction. The two minor issues that occur are:

- When λ-abstracting a new h-function, to preserve well-scopedness we would usually abstract over type variables first, followed by term variables. With kind polymorphism, we must be careful to abstract over *kind* variables first, then type variables, and then finally term variables.
- In the implementation of most specific generalisation, we might say that the MSG of $\Lambda(a :: *).\, \lambda(x :: \mathit{Int}).\, x$ and $\Lambda(a :: * \to *).\, \lambda(x :: \mathit{Int}).\, x$ is the kind-polymorphic term $\Lambda(a :: k).\, \lambda(x :: \mathit{Int}).\, x$ (with appropriate substitutions for k). Although this seems to be theoretically sound for simple Fω extended with kind polymorphism, in practice we found that it interacted badly with the "subkinding" system supporting GHC's use of unlifted types, and so our MSG implementation never abstracts over new kind variables.

E.4 Controlling code explosion

Supercompilation of some programs only completes when given unfeasibly large amounts of time and memory in which to run. Our supercompiler implementation incorporates two mechanisms that are designed to ameliorate these issues:

- We use heuristics to determine which non-normalising β-reductions should be performed (Section E.4.1)
- We incorporate an optional mechanism that limits the total number of non-normalising β-reductions which can be performed in one supercompilation run (Section E.4.2)

E.4.1 Controlling inlining

In the course of our supercompilation research, we found that the supercompiler would often inline large library functions into contexts where there was little to no available additional information, contributing to code bloat without achieving useful specialisation. This code bloat would often reach the point of inlining most of the IO library into the program that uses it, along with code to e.g. parse numbers out of the command line arguments to the program. To combat this, we decided to make use of a combination of annotation and heuristics.

Concretely, we made changes to the implementation of the small-step operational semantics of *step*. The standard *step* function as described in Section 3.3 will use the BETA and TYBETA rules to perform any non-normalising β-reductions. When asked to perform such a β-reduction of a function f, our *step* function instead makes the following checks in order:

1. If f is marked with a *SUPERINLINABLE* pragma in the module where it is defined, or is lexically within a function marked with such a pragma, the β-reduction is unconditionally allowed.

2. If f is a recursive function[2] then the β-reduction is not allowed. This is a powerful size control mechanism as only those loops which we explicitly annotate as *SUPERINLINABLE* will be inlined, which immediately eliminates a large .

3. If GHC's own inlining heuristics [Peyton Jones and Marlow, 2002] would allow the β-reduction to occur, we perform it. The inlining heuristics attempt to determine if the body of the lambda can be simplified enough, given the information available at the call site, to justify the code size increase of the inlining. Although this is necessarily an approximate process, they have been developed and refined by the GHC developers over many years, and so the heuristics give reasonably good results in practice. By deferring to GHC's heuristics, we ensure that the supercompiler β-reduces interesting invocations of non-recursive functions even if those functions are not annotated with *SUPERINLINABLE*.

4. If the normalised result of performing the β-reduction is not any larger (in terms of number of live AST nodes) than the state with the pending β-reduction, then the reduction is allowed. This is useful for permitting β-reductions of functions f which are simple wrappers around other things, or are used exactly once.

5. If inlining f means that the (normalised) result of *step* will be a value, we allow the β-reduction. This is a somewhat ad-hoc condition, but we found it to be crucial for speculation. The reason for this is that typically GHC's inlining heuristics will (correctly) report that inlining a lambda such as $\lambda x\ y.\, x$ into a partial application $f\ x$ is not an interesting thing to do. However, when speculating we absolutely need that inlining to be performed so that we can simplify an expensive binding $g = f\ x$ to the manifestly-cheap binding $g = \lambda y.\, x$.

[2] In fact, we check that f is not a "loop breaker", as defined in Peyton Jones and Marlow [2002].

6. Finally, if no other case matched then we prevent the β-reduction.

As a convenience, when supercompiling a module the supercompiler behaves as if all functions in that module are marked *SUPERINLINABLE*.

For the purposes of the benchmarks of Chapter 7, we annotated the *GHC.List*, *Data.List* and *Data.Complex* modules of GHC's Haskell library implementation with the *SUPERINLINABLE* pragma, since those modules defined those functions which the supercompiler needed to inline in order to achieve deforestation.

E.4.2 Work bounding

Our supercompiler also implements a rudimentary mechanism for limiting the total number of β-reductions performed in the course of supercompiling a module. By limiting the number of β-reduction we indirectly limit the amount of specialisations that we need to create, since we prevent the supercompiler from gaining information about function parameters. To achieve this:

- The *State* type is augmented with a natural number denoting the amount of "fuel" the state has access to. Whenever the *step* function performs a non-normalising β-reduction, it returns a *State* with one less unit of fuel than the input *State*. If no fuel remains, *step* returns *Nothing* even when there is an available β-reduction.

- When *split*ting a state, the fuel of the state input to *split* is distributed evenly across the recursively driven states in proportion to the size of those states (in AST nodes), on the basis that in general larger states will tend to require more fuel to be fully supercompiled.

- Any fuel unused by supercompilation is returned to the caller of the supercompiler—so, for example, the *sc* function gets the new type

  $$sc :: History \to State \to ScpM\ (Fuel, Term)$$

  Fuel unused by previous children of a *split* call is supplied as extra fuel to the next child of *split* to be supercompiled.

- Tying back does not use any fuel, so if *memo* manages to tie back it returns all of the fuel of the input state to the caller.

- The initial amount of fuel is set to a fixed multiple—called the "bloat factor"—of the size of the input module (in AST nodes), on the basis that supercompilation of larger programs will tend to require more β-reductions. In Chapter 7's benchmarks, we set the fuel to 10 times the size of the input: i.e. we use a bloat factor of 10.

One of the attractions of a scheme such as this is that for any given program there exists some bloat factor for which supercompilation will terminate with exactly the same result as it would have without work bounding. Therefore, the bloat factor can act as a user-tuneable "knob" where the user can choose the trade-off between compilation time and output program speed that they wish to make.

A number of variations on this scheme are possible:

- Fuel can be distributed to the children of *split* using some other heuristic. For example, all the fuel could be given to the first *State* to be recursively supercompiled, which would lead to a pure first-come-first-served scheme for fuel, equivalent to the amount of fuel available being stored in a single global mutable variable.

- With the addition of fuel, the order in which the children of *split* are traversed can become important: several variations of traversal order (largest first, smallest-first, closer-to-focus first) are possible.

- Taking traversal order a step further, it is possible to imagine changing the traversal order of the supercompiler itself from a depth-first strategy to breath-first. Such a supercompiler would burn as much fuel as possible close to the root of the process tree, with deeper levels only getting what is left after those higher levels are done.

We have not investigated these possibilities in detail, nor do we regard this class of work-bounding mechanisms as particularly promising avenues of further research. In our experience with this mechanism, we found that if programs are sufficiently slow to supercompile that they require the use β-reduction limiting, then the performance of those programs when supercompiled with limits enabled is rarely better than the performance of the non-supercompiled program.

E.5 Supercompilation versus existing optimisations

There are many existing optimisations for functional programming languages. This section considers how some of these existing optimisations relate to supercompilation. Due to the presence of termination testing and generalisation, it is almost never the case that we can say that supercompilation strictly subsumes another optimisation. However, we can say if a supercompiler would achieve the given optimisation for a program where the termination test does not force generalisation.

E.5.1 Simplification

Almost all compilers for functional languages include a simplification or "shrinking" pass [Appel and Jim, 1997; Peyton Jones and Marlow, 2002] that achieves any obvious reductions in the input program which shrink the program according to some size metric. Examples of things done by a such a pass include β-reduction, inlining of **let** bound values used at most once and **let** floating.

Relationship to supercompilation Due to the fact that a supercompiler incorporates an evaluator, it will achieve all of these optimisations. However, due to the introduction of h-functions, the output of a supercompiler will often include pending β-reductions not present in the input program. For example, consider the trivial term *Just* x which operationally just allocates a heap-bound value and returns it—in our operational semantics, evaluating this term takes 0 steps. The immediate output of supercompiling this term will be:

$$\begin{array}{l}\textbf{let } h0 = \lambda y.\, \mathit{Just}\ y \\ \textbf{in } h0\ x\end{array}$$

Operationally, the supercompiled code:

1. Allocates a value $\lambda y.\, \mathit{Just}\ y$ on the stack

2. Pushes a stack frame applying the x argument

3. Dereferences $h0$, finding a value

4. Pops the x-application stack frame and enters the body of the function with $y = x$
5. Allocates a heap-bound value *Just* x and returns it

Each additional β-reduction introduced by supercompilation amounts to some extra overhead: the sequence above amounts to 3 reduction steps in our operational semantics (LETREC, APP and finally BETA).

This overhead will be much lower than the above sequence of steps indicates in an efficient machine implementation such as the eval-apply STG machine [Peyton Jones, 1992; Marlow and Peyton Jones, 2004]. Furthermore, these new reductions can often be cleaned up by using a standard shrinking simplification pass just after supercompilation—for example, this is the case for the output above because *h0* is used linearly.

An example term where simplification may not help us would be $(\mathit{Just}\ (f\ (f\ x)), \mathit{Just}\ (f\ (f\ x)))$, which supercompiles to this term:

$$\begin{array}{l} \textbf{let}\ \mathit{h1}\ f\ x = \mathit{Just}\ (f\ (f\ x)) \\ \quad\ \ \mathit{h0}\ f\ x = (\mathit{h1}\ f\ x, \mathit{h1}\ f\ x) \\ \textbf{in}\ \mathit{h0}\ f\ x \end{array}$$

Although *h0* can be inlined at its single use site, inlining *h1* would not shrink the lambda term because it would replace the two application nodes in each *h1* f x with two application nodes (f applied to x and f applied to f x) *and* a use of the *Just* data constructor.

E.5.2 Inlining

Compilers for functional languages will often perform inlinings that do not manifestly shrink the input, as long as the inlinings appear to be improvements according to some heuristic. For example, a compiler might insist that the inlining not increase the input size more than a fixed threshold, and for it to "worth it". An example that passes both this criteria might be the function $f = \lambda x.\, \textbf{if}\ x\ \textbf{then}\ \mathit{Left}\ x\ \textbf{else}\ \mathit{Right}\ x$—inlining it at an application site f *True* grows the syntax tree by only a few nodes, and is likely to be beneficial because we can see that f scrutinises its argument.

Because these inlinings do not necessarily shrink the input, there is no guarantee that applying them exhaustively will terminate. A simple example would be the following program involving a recursive definition:

$$\begin{array}{l} \textbf{let}\ f\ x = \textbf{if}\ x\ \textbf{then}\ f\ \mathit{False}\ \textbf{else}\ f\ \mathit{True} \\ \textbf{in}\ f\ \mathit{True} \end{array}$$

Noticing that f scrutinises its argument, a compiler may inline to produce the following program:

$$\begin{array}{l} \textbf{let}\ f\ x = \textbf{if}\ x\ \textbf{then}\ f\ \mathit{False}\ \textbf{else}\ f\ \mathit{True} \\ \textbf{in}\ f\ \mathit{False} \end{array}$$

After f once more, the compiler would clearly be in a loop.

However, the presence of recursive definitions is not required to expose the problems with non-shrinking inlining. Since Haskell allows data types recursive in negative positions we can encode Russel's paradox in Haskell:

$$\begin{array}{l} \textbf{data}\ \mathit{Russel}\ a = \mathit{Rus}\ (\mathit{Russel} \to a) \\ \mathit{foo} :: \mathit{Russel}\ a \to a \end{array}$$

$\mathit{foo}\ (\mathit{Rus}\ f) = f\ (\mathit{Rus}\ f)$

$\mathit{bar} :: a$

$\mathit{bar} = \mathit{foo}\ (\mathit{Rus}\ \mathit{foo})$

A compiler may note that *foo* scrutinises its argument, so inlining it at the call site *foo* (*Rus foo*) is likely to be beneficial. Unfortunately, performing that inlining leads to a program identical to the initial one. Attempting to exhaustively apply this inlining thus leads to non-termination of the compiler.

In practice, non-termination due to non-shrinking inlining is almost universally caused by inlining recursive definitions rather than Russel-like definitions [Peyton Jones and Marlow, 2002]. A pragmatic solution is therefore to avoid inlining any recursive definition, which is sufficient to avoid non-termination in almost all cases.

Relationship to supercompilation In supercompilation, inlining is done aggressively at every use of a function, regardless of how interesting (or otherwise) the context is. This results in excellent specialisation at the cost of considerable code bloat, though the memoisation will recover quite a lot of the loss of code sharing. Supercompilation explicitly addresses questions about non-termination by incorporating a termination condition that ensures that the supercompiler cannot diverge.

E.5.3 Case-of-case transformation

The case-of-case transformation is a non-shrinking reduction that can expose additional information to the optimiser. A simple example of where this is beneficial is the following code corresponding to a version of *fromEnum* (*not x*) where both functions have themselves been inlined:

$$\begin{array}{l}\textbf{case}\ (\textbf{case}\ x\ \textbf{of}\ \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True})\ \textbf{of}\ \mathit{True} \to 1 \\ \hphantom{\textbf{case}\ (\textbf{case}\ x\ \textbf{of}\ \mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True})\ \textbf{of}\ } \mathit{False} \to 0\end{array}$$

The case-of-case transformation pushes the outer case (originating from *fromEnum*) into the two branches of the inner case (originating from *not*):

$$\begin{array}{l}\textbf{case}\ x\ \textbf{of}\ \mathit{True} \to \textbf{case}\ \mathit{False}\ \textbf{of}\ \mathit{True} \to 1; \mathit{False} \to 2 \\ \hphantom{\textbf{case}\ x\ \textbf{of}\ } \mathit{False} \to \textbf{case}\ \mathit{True}\ \textbf{of}\ \mathit{True} \to 1; \mathit{False} \to 2\end{array}$$

As a result of the transformation, we can evaluate the case expression originating from *fromEnum* at compile time rather than at runtime.

In the above example, we immediately duplicated the branches of the outer case into both of the inner cases. This could potentially cause considerable code duplication, though in this case we are lucky in that almost all of the duplication can immediately be removed by simple reductions.

In general, if the outer case branches are of size m and there are n inner branches performing the case-of-case transformation will increase code size by $O(m(n-1))$. The more refined versions of the transformation that tend to be used in practice (such as that in Santos [1995]), would instead transform as follows:

$$\begin{array}{l}\textbf{let}\ j = \lambda y.\ \textbf{case}\ y\ \textbf{of}\ \mathit{True} \to 1; \mathit{False} \to 2 \\ \textbf{in}\ \textbf{case}\ x\ \textbf{of}\ \mathit{True} \to j\ \mathit{False}; \mathit{False} \to j\ \mathit{True}\end{array}$$

The new bound variable j is called a “join point”. With this version of the transformation, code size only increases by a constant factor. The standard non-shrinking inlining heuristics of Section E.5.2 can clean up the output by inlining the normal function j wherever it seems beneficial to do so, achieving most of the same benefits as the version of the transformation that greedily duplicates the branches of the outer case.

The case-of-case transformation can be seen as the dual of standard inlining in that it corresponds to inlining a *continuation* so it can take advantage of the value "returned" to it, rather than inlining a normal function so it can take advantage of the thunk applied to it. This can be readily seen if we represent the original term in a continuation-passing style, explicitly abstracting it over both the free variable x and the continuation k:

```
λx k.
  letcont k0 y = case y of True → k 1; False → k 0
  in case x of True → k0 False; False → k0 True
```

Relationship to supercompilation The case-of-case transformation is key to achieving good optimisation through supercompilation—in particular, *map-map* fusion relies on it to deforest intermediate cons-cells. In keeping with supercompilation's aggressive duplication of code through inlining, the case-of-case transformation is done very aggressively, and does not attempt to share code in the way that the refined case-of-case transformation above does.

E.5.4 Constructor specialisation

In constructor specialisation [Peyton Jones, 2007], versions of functions specialised to particular constructor arguments are generated where an analysis can detect that:

- The corresponding parameters are scrutinised within the body of the function, so the specialised code is likely to be faster than the unspecialised variant.
- There is at least one syntactic occurrence of a call to this function where that constructor is used as an argument, so the specialised code is likely to be used (by at least one caller).

Given a function like the following:

```
last [ ]      = error "last"
last [x]      = x
last (x : xs) = last xs
```

The compiler can see that in the recursive application of *last* (*last xs*), *xs* is guaranteed to be a cons, and that *last* scrutinises that part of the argument. Thus, constructor specialisation will produce a specialised function *last_spec*, such that any call of the form *last* $(y : ys)$ can be replaced with a call *last_spec y ys*. After replacing all such calls, the following optimised program can be obtained:

```
last [ ]          = error "last"
last [x]          = x
last (x : y : zs) = last_spec y ys

last_spec y [ ]      = y
last_spec y (x : xs) = last_spec x xs
```

Relationship to supercompilation Constructor specialisation is naturally achieved via supercompilation, as the whole business of a supercompiler is to perform specialisation. Supercompilation is a much more robust specialisation method than constructor specialisation. For example, you might hope that the following program would be optimised so that recursive calls to g do not need to scrutinise the free variable p:

```
f p = let g y = case p of (x, _) → g (p + y)
      in Just (g 10)
```

Unfortunately, constructor specialisation is only capable of specialising functions on their *arguments*, not their *free variables*, so we cannot obtain this optimised code:

```
f p = let g y = case p of (x, _) → g_spec x (x + y)
          g_spec x y = g_spec x (x + y)
      in Just (g 10)
```

A closely-related optimisation (specific to Haskell programs) is that of type-class specialisation, where function are specialised for the "dictionary" arguments used to implement ad-hoc overloading via type-classes [Wadler and Blott, 1989]. These dictionaries are typically tuples of *functions*, not algebraic data, so constructor specialisation has no effect. Supercompilation makes no distinction between functions and data so can achieve type-class specialisation via the same mechanisms as it obtains constructor specialisation.

E.5.5 Strictness analysis

When compiling a non-strict functional language, it can be beneficial to detect functions for which it is semantically safe to use call-by-value application rather than call-by-name or call-by-need application. By using call-by-value application you can avoid the overheads associated with heap-allocating thunks to implement suspended computations.

Strictness analysis [Mycroft, 1980] detects lambdas (and corresponding application sites) for which it is safe to use call-by-value application. For example, most strictness analysers (e.g. [Wadler and Hughes, 1987]) will identify *fact* as being strict in the n and *acc* parameters:

```
fact acc n = if n ≡ 0 then acc else fact (acc * n) (n − 1)
```

Since *fact* is strict in *acc* and n, it is semantically safe to use call-by-value application for the corresponding arguments. In fact, GHC will not only use the results of strictness analysis as an opportunity to perform call-by-value application, but will also use the worker-wrapper transformation [Gill and Hutton, 2009] to unpack strict arguments. For example, in the following program:

```
f p x = if x ≡ 0 then fst p else f (if odd x then (snd p, fst p) else p) (x − 1)
```

GHC will not only identify f as strict in p and x, but will also unpack the pair p, resulting in the following code:

```
f (p1, p2) x = f_worker p1 p2 x
f_worker p1 p2 x = if x ≡ 0 then fst p else f (if odd x then (snd p, fst p) else p)
                                                (x − 1)
  where p = (p1, p2)
```

After optimisation, the "wrapper" f is inlined into *f_worker* and we obtain an efficient loop that avoids allocating new pairs in the heap:

```
f_worker p1 p2 x = if x ≡ 0 then p1 else if odd x then f_worker p2 p1 (x − 1)
                                                   else f_worker p1 p2 (x − 1)
```

Relationship to supercompilation Supercompilation is not always able to unpack those arguments that would get unpacked by the combination of strictness analysis and worker-wrapper transformation. We consider two situations.

First, the case of a series of non-recursive functions (these typically do not trigger the supercompilation termination condition). For these functions, supercompilation tends to cause an allocated thunk to reach the context in which it is used and consequently the allocation of the thunk can be prevented. For example, in the following program:

```
let fst p = case p of (a, b) → a
    snd p = case p of (a, b) → b
    x = if b then (1, 2) else (3, 4)
    g p = fst p + snd p
in g x
```

Supercompilation can straightforwardly reduce this to the following code:

```
if b then 3 else 7
```

Interestingly, this process is not impeded by the check (in call-by-need supercompilation) that prevents duplication of thunks that are used non-linearly. The reason for this is that non-linear uses of a variable x arise from expressions like (x, x) or $\lambda y.\, x$ that as well as causing non-linearity, cause x to be non-strict as well.

The second situation to consider is where we have recursive functions which make recursive calls with newly-constructed thunks as arguments. The first two examples in this section (*fact* and f) are both of this sort. Recall that the definition of f is as follows:

```
f p x = if x ≡ 0 then fst p else f (if odd x then (snd p, fst p) else p) (x − 1)
```

We consider the sequence of specialisations created when supercompiling the call *f p0 x0*:

```
f p0 x0

let p1 = if odd x0 then (snd p0, fst p0) else p0
    x1 = x0 − 1
in f p1 x1

let p1 = if odd x0 then (snd p0, fst p0) else p0
    p2 = if odd x1 then (snd p1, fst p1) else p1
    x2 = x1 − 1
in f p2 x2
```

(Note the asymmetric treatment of the p and x parameters: this arises from the fact that x is strictly evaluated by the enclosing **if** before the recursive call to f.)

At this point, the supercompiler will generalise, preventing specialisation on the bindings for *p0* and *p1*. As a result, supercompilation is not able to remove the thunks allocated to implement call-by-need evaluation of the pair arguments to f.

This is a general pattern: applications of non-strict arguments are by almost always *accumulating* arguments (these applications build a new thunk that refer to those older thunks bound by unevaluated parameters). Supercompilation of non-strict functions shares all the same problems as supercompilation of functions with accumulating arguments, and fails to optimise well.

Although supercompilation is unable to *replace* strictness analysis, it can be complementary to it. For example, a compiler that uses a first-order strictness analyser can increase the effectiveness of strictness analysis by supercompiling first, as supercompilation will tend to specialise away higher-order arguments.

Bibliography

GCC, the GNU Compiler Collection. URL `http://gcc.gnu.org`.

javac - Java programming language compiler. URL `http://download.oracle.com/javase/`.

A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *J. Funct. Program.*, 7(5):515–540, Sept. 1997. ISSN 0956-7968. doi: 10.1017/S0956796897002839. URL `http://dx.doi.org/10.1017/S0956796897002839`.

A. P. Black, M. Carlsson, M. P. Jones, R. Kieburtz, and J. Nordlander. Timber: a programming language for real-time embedded systems. Technical Report CSE-02-002, Oregon Health & Science University, 2002.

M. Bolingbroke and S. Peyton Jones. Supercompilation by evaluation. In *Proceedings of the 2010 ACM SIGPLAN Haskell Symposium*, September 2010.

M. Bolingbroke, S. Peyton Jones, and D. Vytiniotis. Termination combinators forever. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 23–34. ACM, 2011. ISBN 978-1-4503-0860-1.

A. Bondorf. Improving binding times without explicit CPS-conversion. In *Proceedings of the 1992 ACM conference on LISP and functional programming*, LFP '92, pages 1–10, New York, NY, USA, 1992. ACM. ISBN 0-89791-481-3. doi: 10.1145/141471.141483. URL `http://doi.acm.org/10.1145/141471.141483`.

R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1):44–67, 1977.

M. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 241–253, New York, NY, USA, 2005. ACM.

W. N. Chin. *Automatic Methods for Program Transformation.* PhD thesis, 1990.

O. Chitil. Common subexpression elimination in a lazy functional language. In *Draft Proceedings of the 9th International Workshop on Implementation of Functional Languages*, pages 501–516, 1997.

C. Consel and O. Danvy. For a better support of static data flow. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 496–519. Springer Berlin Heidelberg, 1991. ISBN 978-3-540-54396-1. doi: 10.1007/3540543961_24. URL `http://dx.doi.org/10.1007/3540543961_24`.

C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 493–501, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. doi: 10.1145/158511.158707. URL http://doi.acm.org/10.1145/158511.158707.

C. Consel and S. C. Khoo. Parameterized partial evaluation. *ACM Trans. Program. Lang. Syst.*, 15(3):463–493, July 1993. ISSN 0164-0925. doi: 10.1145/169683.174155. URL http://doi.acm.org/10.1145/169683.174155.

D. Coutts. *Stream Fusion: practical shortcut fusion for coinductive sequence types.* PhD thesis, Worcester College, University of Oxford, 2010.

D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, April 2007.

L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

R. Ennals and S. P. Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*, pages 287–298, New York, NY, USA, 2003. ACM. ISBN 1-58113-756-7. doi: http://doi.acm.org/10.1145/944705.944731.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. *ACM SIGPLAN Notices*, 28(6):237–247, 1993. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/173262.155113.

J. Gibbons and B. C. d. S. Oliveira. The essence of the iterator pattern. *Journal of Functional Programming*, 19, 2009. doi: 10.1017/S0956796809007291.

A. Gill and G. Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251, March 2009.

A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X. doi: 10.1145/165180.165214. URL http://doi.acm.org/10.1145/165180.165214.

R. Glück and J. Jørgensen. Generating transformers for deforestation and supercompilation. In *Static Analysis, volume 864 of Lecture Notes in Computer Science*, pages 432–448. Springer-Verlag, 1994.

G. W. Hamilton. *Compile-time optimisation of store usage in lazy functional programs.* PhD thesis, 1993.

G. W. Hamilton. Higher order deforestation. *Fundam. Inf.*, 69(1-2):39–61, July 2005. ISSN 0169-2968. URL http://dl.acm.org/citation.cfm?id=1227247.1227250.

G. W. Hamilton. Distillation: extracting the essence of programs. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '07, pages 61–70, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-620-2. doi: http://doi.acm.org/10.1145/1244381.1244391. URL http://doi.acm.org/10.1145/1244381.1244391.

G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 3(1):326, 1952.

R. Hinze et al. Generalizing generalized tries. *Journal of Functional Programming*, 10(4): 327–351, 2000.

G. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, Sept. 1997. ISSN 0956-7968. doi: 10.1017/S0956796897002864. URL `http://dx.doi.org/10.1017/S0956796897002864`.

J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

R. Hughes. *The design and implementation of programming languages.* PhD thesis, University of Oxford, 1983.

N. Jones, P. Sestoft, and H. Sndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer Berlin / Heidelberg, 1985. ISBN 978-3-540-15976-6. URL `http://dx.doi.org/10.1007/3-540-15976-2_6`.

N. D. Jones, P. Sestoft, and H. Sndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2:9–50, 1989. ISSN 0892-4635. URL `http://dx.doi.org/10.1007/BF01806312`. 10.1007/BF01806312.

N. D. Jones, C. K. Gomard, and P. Sestoft. Partial evaluation and automatic program generation. *Prentice-Hall International Series In Computer Science*, page 415, 1993.

S. Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture*, pages 636–666. Springer, 1991.

P. A. Jonsson. *Time- and Size-Efficient Supercompilation.* PhD thesis, Luleå University of Technology, 2011.

P. A. Jonsson and J. Nordlander. Positive supercompilation for a higher order call-by-value language. In *POPL '09: Proceedings of the 36th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2009.

P. A. Jonsson and J. Nordlander. Strengthening supercompilation for call-by-value languages. In *Proceedings of Second International Workshop on Metacomputation*, pages 64–81, 2010.

J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 258–268, New York, NY, USA, 1992. ACM. ISBN 0-89791-453-8. doi: http://doi.acm.org/10.1145/143165.143220.

A. Klimov. A Java supercompiler and its application to verification of cache-coherence protocols. *Perspectives of Systems Informatics*, pages 185–192, 2010.

I. Klyuchnikov. Supercompiler HOSC 1.0: under the hood. Preprint 63, Keldysh Institute of Applied Mathematics, Moscow, 2009. URL `http://library.keldysh.ru/preprint.asp?lg=e&id=2009-63`.

I. Klyuchnikov. Supercompiler HOSC 1.5: homeomorphic embedding and generalization in a higher-order setting. Preprint 62, Keldysh Institute of Applied Mathematics, Moscow, 2010a. URL http://pat.keldysh.ru/~ilya/preprints/HOSC15_en.pdf.

I. Klyuchnikov. Supercompiler HOSC 1.1: proof of termination. Preprint 21, Keldysh Institute of Applied Mathematics, Moscow, 2010b. URL http://library.keldysh.ru/preprint.asp?lg=e&id=2010-21.

I. Klyuchnikov. Towards effective two-level supercompilation. Preprint, Keldysh Institute of Applied Mathematics, Moscow, 2010c.

D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

J. Kruskal. Well-quasi-ordering, the Tree Theorem, and Vazsonyi's conjecture. *Trans. Amer. Math. Soc*, 95:210–225, 1960.

J. Launchbury. A natural semantics for lazy evaluation. In *Principles of Programming Languages*, pages 144–154. ACM, January 1993.

J. Launchbury and R. Paterson. Parametricity and unboxing with unpointed types. *Programming Languages and SystemsESOP'96*, pages 204–218, 1996.

M. Leuschel. On the power of homeomorphic embedding for online termination. In *Static Analysis*, volume 1503 of *Lecture Notes in Computer Science*, pages 230–245. Springer Berlin / Heidelberg, 1998. ISBN 978-3-540-65014-0. doi: 10.1007/3-540-49727-7_14. URL http://www.springerlink.com/content/g93wxkkwpfvvmnmg/.

M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. *Partial Evaluation*, pages 263–283, 1996.

A. Lisitsa and A. Nemytykh. Towards verification via supercompilation. In *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, volume 2, pages 9–10. IEEE, 2005.

A. P. Lisitsa and A. P. Nemytykh. Verification as specialization of interpreters with respect to data. In *Proceedings of First International Workshop on Metacomputation*, pages 94–112, 2008.

S. Marlow. *Deforestation for Higher-Order Functional Programs.* PhD thesis, University of Glasgow, 1996.

S. Marlow and S. Peyton Jones. How to make a fast curry: push/enter vs eval/apply. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2004*, pages 4–15, September 2004.

S. Marlow and P. Wadler. Deforestation for higher-order functions. In *Workshop on Functional Programming, Workshops in Computing*, pages 154–165. Springer-Verlag, 1992.

S. Marlow et al. Haskell 2010 language report. URL http://www.haskell.org/onlinereport/haskell2010/.

E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

N. Mitchell. *Transformation and Analysis of Functional Programs*. PhD thesis, University of York, June 2008.

N. Mitchell. Rethinking supercompilation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2010*. ACM, 2010.

N. Mitchell and C. Runciman. A supercompiler for core Haskell. In *Implementation and Application of Functional Languages*, volume 5083 of *Lecture Notes in Computer Science*, pages 147–164. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-85372-5. doi: 10.1007/978-3-540-85373-2_9. URL `http://www.springerlink.com/content/f4306653245u2767/`.

A. Moran and D. Sands. Improvement in a lazy context: an operational theory for call-by-need. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. ACM, 1999a.

A. Moran and D. Sands. Improvement in a lazy context: an operational theory for call-by-need (extended version). 1999b.

A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the Fourth 'Colloque International sur la Programmation' on International Symposium on Programming*, pages 269–281, London, UK, 1980. Springer-Verlag. ISBN 3-540-09981-6.

C. S. Nash-Williams. On well-quasi-ordering finite trees. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 59, pages 833–835. Cambridge Univ Press, 1963.

C. S. Nash-Williams. On well-quasi-ordering infinite trees. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 61, pages 697–720. Cambridge Univ Press, 1965.

K. Nielsen and M. H. Sørensen. Call-by-name cps-translation as a binding-time improvement. In *Proceedings of the Second International Symposium on Static Analysis*, SAS '95, pages 296–313, London, UK, UK, 1995. Springer-Verlag. ISBN 3-540-60360-3. URL `http://dl.acm.org/citation.cfm?id=647163.717677`.

W. Partain. The nofib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer-Verlag. ISBN 3-540-19820-2.

S. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2:127–202, April 1992. URL `http://research.microsoft.com/en-us/um/people/simonpj/papers/spineless-tagless-gmachine.ps.gz`.

S. Peyton Jones. Constructor specialisation for Haskell programs. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, pages 327–337, 2007.

S. Peyton-Jones. Add a transformation limit to the simplifier (Trac #5448), 2012. URL `http://hackage.haskell.org/trac/ghc/changeset/24a2353a77111e9f236325521edd233f35954328`.

S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4-5):393–434, 2002.

S. Peyton Jones and P. Wadler. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

S. Peyton Jones, C. Hall, K. Hammond, J. Cordy, K. Hall, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview, 1992.

S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. September 2001.

G. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163, 1969.

J. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In *Machine Intelligence*, volume 5, pages 135–151, 1969.

N. Robertson and P. Seymour. Graph minors. iv. tree-width and well-quasi-ordering. *Journal of Combinatorial Theory, Series B*, 48(2):227–254, 1990. ISSN 0095-8956.

J. Robinson. A machine-oriented logic based on the resolution principle. In *Journal of the Association for Computing Machinery*, number 12, pages 23–41, January 1965.

E. Ruf. *Topics in Online Partial Evaluation*. PhD thesis, March 1993.

D. Sands. Total correctness by local improvement in program transformation. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. ACM, 1995.

D. Sands. Computing with contexts: A simple approach. *Electronic Notes in Theoretical Computer Science*, 10:134–149, 1998.

A. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, 1995. TR-1995-17.

W. L. Scherlis. Program improvement by internal specialization. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 41–49. ACM, 1981.

J. P. Secher and M. H. Sørensen. On perfect supercompilation. In *Proceedings of the Third International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, PSI '99, pages 113–127, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-67102-1. URL `http://dl.acm.org/citation.cfm?id=646801.705649`.

P. Sestoft. The structure of a self-applicable partial evaluator. In *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 236–256. Springer Berlin / Heidelberg, 1986. ISBN 978-3-540-16446-3. doi: 10.1007/3-540-16446-4_14. URL `http://www.springerlink.com/content/fq0220jv466hq750/`.

P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(03):231–264, 1997. doi: 10.1017/S0956796897002712. URL `http://journals.cambridge.org/action/displayAbstract?fromPage=online&aid=44087&fulltextType=RA&fileId=S0956796897002712`.

O. Shivers. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 164–174. ACM, 1988.

M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479. MIT Press, 1995.

M. H. Sørensen and R. Glück. Introduction to supercompilation. In *Partial Evaluation*, volume 1706 of *Lecture Notes in Computer Science*, pages 246–270. Springer Berlin / Heidelberg, 1999. ISBN 978-3-540-66710-0. doi: 10.1007/3-540-47018-2_10. URL `http://www.springerlink.com/content/91351253606g6823/`.

M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6:465–479, 1993.

M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and gpc. In *ESOP '94: Proceedings of the 5th European Symposium on Programming*, pages 485–500, London, UK, 1994. Springer-Verlag. ISBN 3-540-57880-3.

G. Steele Jr and L. Common. *Common Lisp: the language*, volume 20. 1984.

M. Sulzmann, M. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*. ACM, 2007.

J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, pages 124–132, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8. doi: 10.1145/581478.581491. URL `http://doi.acm.org/10.1145/581478.581491`.

D. Syme and J. Margetson. The F# programming language. 2008. URL `http://research.microsoft.com/projects/fsharp`.

M. Tullsen, P. Hudak, et al. Shifting expression procedures into reverse. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Technical report BRICS-NS-99-1, University of Aarhus*, pages 95–104, 1999.

V. F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3): 292–325, 1986. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/5956.5957.

V. F. Turchin. The algorithm of generalization in the supercompiler. *Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, Partial Evaluation and Mixed Computation*, pages 531–549, 1988.

D. Vytiniotis, S. Jones, and J. Magalhães. Equality proofs and deferred type errors a compiler pearl. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP*, 2012.

P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer Berlin / Heidelberg, 1988. ISBN 978-3-540-19027-1. doi: 10.1007/3-540-19027-9_23. URL `http://www.springerlink.com/content/7217v376n7388582/`.

P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM. ISBN 0-89791-453-8. doi: 10.1145/143165.143169. URL `http://doi.acm.org/10.1145/143165.143169`.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75283. URL `http://doi.acm.org/10.1145/75277.75283`.

P. Wadler and J. Hughes. Projections for Strictness Analysis. In *Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, September 1987.

P. Walder. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0.

D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 165–191. Springer Berlin / Heidelberg, 1991. ISBN 978-3-540-54396-1. URL `http://dx.doi.org/10.1007/3540543961_9`.

H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. *SIGPLAN Not.*, 38(1):224–235, Jan. 2003. ISSN 0362-1340.

B. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66. ACM, 2012.