Transformation and Analysis of Functional Programs

Neil Mitchell

Submitted for the degree of Doctor of Philosophy

Department of Computer Science
University of York
June 2008

Abstract

This thesis describes techniques for transforming and analysing functional programs. We operate on a core language, to which Haskell programs can be reduced. We present a range of techniques, all of which have been implemented and evaluated.

We make programs *shorter* by defining a library which abstracts over common data traversal patterns, removing *boilerplate* code. This library only supports traversals having value-specific behaviour for one type, allowing a simpler programming model. Our library allows concise expression of traversals with competitive performance.

We make programs *faster* by applying a variant of *supercompilation.* As a result of practical experiments, we have identified modifications to the standard supercompilation techniques – particularly with respect to let bindings and the generalisation technique.

We make programs *safer* by automatically checking for potential pattern-match errors. We define a transformation that takes a higher-order program and produces an equivalent program with fewer functional values, typically a first-order program. We then define an analysis on a first-order language which checks statically that, despite the possible use of partial (or non-exhaustive) pattern matching, no pattern-match failure can occur.

Contents

List of Figures

List of Tables

Acknowledgements

Throughout the PhD I have been supported by an EPSRC PhD studentship. I would like to thank Colin Runciman for his supervision throughout the last 6 years. Colin taught me Haskell, helped me with technical problems, and helped me to express myself more clearly in my writing. In addition to Colin's supervision, all the members of the PLASMA group have provided interesting discussions, lots of technical debate and answers to LaTeX problems.

Many people in the Haskell community have provided ideas, encouragement, code and answers. Included in this list are Andres Löh, Björn Bringert, Brandon Moore, Damien Sereni, Duncan Coutts, Eric Mertens, Jürgen Doser, Jules Bean, Koen Claessen, Matthew Danish, Peter Jonsson, Simon Marlow, Simon Peyton Jones, Stefan O'Rear, Tim Chevalier and the whole of the Haskell community, particularly `#haskell`. The vast number of people who have helped ensures that I have certainly forgotten many people.

While doing a PhD, I have appreciated the presence of many friends – including all the members of the York University Karate Club, and the many residents of 232 Melrosegate. Thanks to Emily for making the last month of my PhD a fantastic year. Lastly, thanks to my family, who have given me the freedom to make my own decisions, and an occasional email to check on my wellbeing.

Declaration

Chapter 2 has some overlap with material published in (Golubovsky et al. 2007). Chapter 3 is based on the paper (Mitchell and Runciman 2007c), which appeared at the Haskell Workshop 2007. Chapter 4 is based on the paper (Mitchell and Runciman 2007b) which was presented at IFL 2007, and the revised paper (Mitchell and Runciman 2008b) from the post proceedings. Chapter 6 builds on work from the papers (Mitchell and Runciman 2005, 2007a) presented at TFP 2005 and appearing in the post proceedings, and is based on the paper (Mitchell and Runciman 2008a) from the Haskell Symposium 2008.

Apart from the above cases and where stated, all of the work contained within this thesis represents the original contribution of the author.

Chapter 1

Introduction

This thesis is concerned with functional programming. Throughout the thesis all examples and implementations are presented in Haskell (Peyton Jones 2003). Much of this work takes advantage of the purity of Haskell, and some requires lazy evaluation, but many of the ideas should be applicable to other functional languages.

In this chapter, we first discuss the motivation underlying the problems we have tackled in §1.1. Next we provide details of where to obtain implementations related to this thesis in §1.2, followed by a description of each of the following chapters in §1.3.

1.1 Motivation and Objectives

This thesis has three main objectives: making functional programs shorter, faster and safer. This section explains the particular aims within each area, and how the areas are related. We present the motivation for the objectives in reverse order, being the order we tackled them, to show how each motivates the next.

1.1.1 Making Programs Safer

Haskell is a strongly typed language, ensuring that a large class of errors are caught at compile time. Despite all the guarantees that the type system provides, programs may still fail in three ways:

Wrong Behaviour Detecting incorrect behaviour requires the programmer to provide annotations describing the desired behaviour. Mandatory annotations increase the effort required to make use of a tool, and therefore reduce the potential number of users.

Non-termination The issue of non-termination has been investigated extensively – one particularly impressive tool is the AProVE framework (Giesl et al. 2006b).

Calling error The final cause of failure is calling error, often as the result of an incomplete pattern-match. This issue has not received as much attention, with suggestions that programmers only use exhaustive patterns (Turner 2004), or local analysis to decide which patterns are exhaustive (Maranget 2007). The problem of calling error is a practical one, with such failures being a common occurrence when developing a Haskell program.

In order to make programs safer, we have developed the Catch tool, which ensures a program does not call error. We decided to make our analysis *conservative* – if it reports that a program will not call error, then the program is guaranteed not to call error. We require no annotations from the programmer.

The Catch tool operates on a first-order language. We attempted to extend Catch to a higher-order language, but failed. A higher-order program has more complicated flow-control, which causes problems for Catch. In order to apply Catch to all Haskell programs, we have investigated defunctionalisation – converting a higher-order program to a first-order program. Our defunctionalisation method is called Firstify, and uses well-known transformations, particularly specialisation and inlining, applied in particular ways. The defunctionalisation method is designed to be used as a transformation before analysis, primarily for Catch, but can be used independently.

1.1.2 Making Programs Faster

After making a program first-order, it can often execute faster than before. As we explored this aspect of defunctionalisation, we were drawn towards other optimisation techniques – in particular supercompilation (Turchin 1986). Just as defunctionalisation often leads to improved performance, so supercompilation often leads to the removal of higher-order values. We attempted to construct a defunctionalisation method by restricting supercompilation, but the result was not very successful. However, we did enhance our defunctionalisation method using techniques from supercompilation, particularly the termination criteria.

We have developed a supercompiler named Supero. Our work on supercompilation aims to allow Haskell programs to be written in a high-level style, yet perform competitively. Often, to obtain high performance, Haskell programmers are forced to make use of low-level features such as unboxed types (Peyton Jones and Launchbury 1991), provide additional annotations such as rewrite rules (Peyton Jones et al. 2001) and express programs in an unnatural style, such as using foldr to obtain deforestation (Gill et al. 1993). Supero can optimise Haskell programs, providing substantial speed-ups in some cases. Like Catch, Supero requires no annotations from the programmer.

1.1.3 Making Programs Shorter

Our final contribution is the Uniplate library. The expression type of the Core language we work with has over ten constructors. Most of these constructors contain embedded subexpressions. For most operations, we wish to have value-specific behaviour for a handful of constructors, and a default operation for the others. We started developing a small library of useful functions to deal with this complexity, and gradually abstracted the ideas. After refinement, the Uniplate library emerged. The library is particularly focused on concisely expressing common patterns. Compared to other work on generic programming patterns, such as SYB (Lämmel and Peyton Jones 2003) and Compos (Bringert and Ranta 2006), the Uniplate library makes use of fewer language extensions and permits more concise operations.

The Uniplate library stands apart from the rest of the thesis in that it does not work on a core functional language, but is instead a general purpose library. However, the Uniplate techniques have been invaluable in implementing the other transformations.

1.2 Implementations

We have implemented all the ideas presented in this thesis, and include sample code in the related chapters. Most of our implementations make use of a monadic framework to deal with issues such as obtaining unique free variables and tracking termination constraints. But to simplify the presentation, we ignore these issues – they are mostly tedious engineering concerns, and do not effect the underlying algorithms.

All the code is available from the author's homepage[1]. Additionally, we have released the following packages on the Hackage website[2]:

Homeomorphic This is a library for testing for homeomorphic embedding, used to ensure termination, as described in §2.4.

Uniplate This is the library described in Chapter 3.

Derive This tool can generate Uniplate instances, and is mentioned in §3.3.4.

Yhc.Core This is a library providing the data type for Yhc's Core language. It requires Uniplate to implement some of the functions.

Supero This is the program described in Chapter 4. It requires Yhc.Core as the Core language to operate on, Homeomorphic to ensure termination and Uniplate for various transformations.

Firstify This is the library described in Chapter 5. Like Supero, this library requires Yhc.Core, Homeomorphic and Uniplate.

Proposition This is the proposition library described in 6, particularly Figure 6.3.

Catch This is the program described in Chapter 6. This library requires Proposition, and the Firstify library and all its dependencies.

[1] `http://www.cs.york.ac.uk/~ndm/`

[2] `http://hackage.haskell.org/`

1.3 Chapter Outline

The Background chapter (2) describes a common Core language which is used in the subsequent chapters. It also describes the homeomorphic embedding relation, used to ensure termination in a number of transformations.

The Boilerplate Removal chapter (3) describes the Uniplate library. In particular, it describes the interface to the library – both the traversal functions and the information a data type must provide. It also compares the Uniplate library to the Scrap Your Boilerplate (SYB) library (Lämmel and Peyton Jones 2003) and the Compos library (Bringert and Ranta 2006) – both in terms of speed and conciseness.

The Supercompilation chapter (4) describes the design and implementation of the Supero tool. The method includes techniques for dealing with let bindings, and a new method for generalisation. Results are presented comparing a combination of Supero and the Glasgow Haskell Compiler (GHC) (The GHC Team 2007) to C, and comparing Supero and GHC to GHC alone.

The Defunctionalisation chapter (5) describes how to combine several existing transformations to produce a defunctionalisation method. The main focus is how to restrict the existing methods to ensure they terminate and cooperate to obtain a program with few residual functional values.

The Pattern-Match Analysis chapter (6) describes the implementation of the Catch tool. It presents a mechanism for reasoning about programs using a constraint language, along with two alternative constraint languages. The Catch tool is tested on a number of benchmark programs, and for several larger programs.

The Conclusions chapter (7) gives directions for future work, and makes concluding remarks.

The Soundness of Pattern-Match Analysis Appendix (A) provides a soundness proof of the algorithms presented in Chapter 6.

The Function Index Appendix (B) is an index of most of the Haskell functions used in the thesis, both those defined in the thesis and those from the standard Haskell libraries.

Chapter 2

Background

In this chapter we introduce the background material and general notations used throughout the rest of this thesis. We start by introducing a Core language in §2.1, then discuss its sharing properties in §2.2 and how we generate Core in §2.3. We then cover the homeomorphic embedding relation in §2.4, particularly applied to the expression type of our Core language.

2.1 Core Language

The syntax of our Core language is given in Figure 2.1. To specify a list of items of unspecified length we write either $\mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{n}$ or $\overline{\mathsf{xs}}$. Our Core language is higher order and lazy, but lacks much of the syntactic sugar found in Haskell. The language is based upon Yhc.Core, a semantics for which is given in (Golubovsky et al. 2007).

A program is a list of functions, with a root function named main. A function definition gives a name, a list of arguments and a body expression. Variables and lambda abstractions are much as they would be in any Core language. Pattern matching occurs only in case expressions; alternatives match only the top level constructor and are exhaustive, including an error alternative if necessary.

In later chapters it will be necessary to make a distinction between higher-order and first-order programs, so our Core language has some redundancy in its representation. Our Core language permits both lambda expressions, and allows top-level definitions to take arguments. There are three forms

$\textbf{data}\ \mathsf{Prog} = [\mathsf{Func}]$	program
$\textbf{data}\ \mathsf{Func} = (\mathsf{f}\ \overline{\mathsf{vs}} = \mathsf{x})$	function
$\textbf{data}\ \mathsf{Expr} = \mathsf{v}$	local variable
$\mid\ \mathsf{c}\ \overline{\mathsf{xs}}$	constructor application
$\mid\ \mathsf{f}\ \overline{\mathsf{xs}}$	function application
$\mid\ \mathsf{x}\ \overline{\mathsf{xs}}$	general application
$\mid\ \lambda\mathsf{v} \rightarrow \mathsf{x}$	lambda abstraction
$\mid\ \textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ \mathsf{y}$	let binding, non-recursive
$\mid\ \textbf{case}\ \mathsf{x}\ \textbf{of}\ \overline{\mathsf{as}}$	case expression
$\textbf{data}\ \mathsf{Alt} = \mathsf{c}\ \overline{\mathsf{vs}} \rightarrow \mathsf{x}$	case alternative

Where v ranges over variables, c ranges over constructors, f ranges over function names, x and y range over expressions and a ranges over case alternatives.

Figure 2.1: Syntax for the Core language.

of application, all of which take two values: the first value may be either a constructor, a top-level named function, or any arbitrary expression; the second value is a list of arguments, which may be empty. These forms of application give rise to three equivalences:

$$(\mathsf{c}\ \overline{\mathsf{xs}})\ \overline{\mathsf{ys}} \equiv \mathsf{c}\ \overline{\mathsf{xs}}\ \overline{\mathsf{ys}}$$
$$(\mathsf{f}\ \overline{\mathsf{xs}})\ \overline{\mathsf{ys}} \equiv \mathsf{f}\ \overline{\mathsf{xs}}\ \overline{\mathsf{ys}}$$
$$(\mathsf{x}\ \overline{\mathsf{xs}})\ \overline{\mathsf{ys}} \equiv \mathsf{x}\ \overline{\mathsf{xs}}\ \overline{\mathsf{ys}}$$

We allow a list of variables to appear in a lambda abstraction and a list of bindings to appear in a let. This syntactic sugar can be translated away using the following rules:

$$\lambda\mathsf{v}\ \overline{\mathsf{vs}} \rightarrow \mathsf{x} \Rightarrow \lambda\mathsf{v} \rightarrow (\lambda\overline{\mathsf{vs}} \rightarrow \mathsf{x})$$
$$\textbf{let}\ \mathsf{v} = \mathsf{x}; \overline{\mathsf{binds}}\ \textbf{in}\ \mathsf{y} \Rightarrow \textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ (\textbf{let}\ \overline{\mathsf{binds}}\ \textbf{in}\ \mathsf{y})$$
$$\textbf{let}\ \mathsf{v}\ \overline{\mathsf{vs}} = \mathsf{x}\ \overline{\mathsf{xs}}\ \textbf{in}\ \mathsf{y} \Rightarrow \textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ (\textbf{let}\ \overline{\mathsf{vs}} = \overline{\mathsf{xs}}\ \textbf{in}\ \mathsf{y})$$

The arity of a top-level function is the number of arguments in its associated definition. In any application, if the function is given fewer arguments than its arity we refer to it as *partially-applied*, matching the arity is *fully-applied*, and more than the arity is *over-applied*.

Some functions are used but lack corresponding definitions in the program. These are defined to be *primitive*. They have some meaning to an underly-

```
type CtorName  = String
type VarName   = String
type FuncName = String

body :: FuncName → Expr
args  :: FuncName → [VarName]
rhs   :: Alt         → Expr
arity :: String      → Int
ctors :: CtorName  → [CtorName]
```

Figure 2.2: Operations on Core.

ing runtime system, but are not available for transformation. A primitive function may perform an action such as outputting a character to the screen, or may manipulate primitive numbers such as addition.

The largest difference between our Core language and GHC-Core (Tolmach 2001) is that our Core language is untyped. Core is generated from well-typed Haskell, and is guaranteed not to fail with a type error. All our algorithms could be implemented equally well in a typed core language, but we prefer to work in an untyped language for simplicity of implementation. For describing data types we use the same notation as Haskell 98. One of the most common data types is the list, which can be defined as:

```
data List α = Nil | Cons α (List α)
```

A list is either an empty list, or a cons cell which contains an element of the list type and the tail of the list. For example the list of 1,2,3 would be written (Cons 1 (Cons 2 (Cons 3 Nil))). We allow the syntactic sugar of representing Cons as a right-associative infix application of (:) and Nil as [] – allowing us to write (1 : 2 : 3 : []). We also permit $[1, 2, 3]$.

2.1.1 Operations on Core

There are several operations that can be defined on our Core expressions type. We present some of those used in later chapters.

General Operations

Figure 2.2 gives the signatures for helper functions over the core data types. We use the functions $\mathsf{body}\ \mathsf{f}$ and $\mathsf{args}\ \mathsf{f}$ to denote the body and arguments of the function definition for f. We use the function rhs to extract the expression on the right of a case alternative. Every function and constructor has an arity, which can be obtained with the arity function. To determine alternative constructors the ctors function can be used; for example ctors `"True"` $=$ [`"False"`, `"True"`] and ctors `"[]"` $=$ [`"[]"`, `":"`].

Substitution

We define $\mathsf{e}\ [\mathsf{v}\ /\ \mathsf{x}]$ to be the capture-free substitution of the variable v by the expression x within the expression e. We define $\mathsf{e}\ [\mathsf{v}_1, \ldots, \mathsf{v}_\mathsf{n}\ /\ \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{n}]$ to be the simultaneous substitution of each variable $\mathsf{v_i}$ for each expression $\mathsf{x_i}$ in e.

Example 1

$$\begin{array}{ll} (\mathsf{v} + 1)\ [\mathsf{v}\ /\ 2] & \Rightarrow 2 + 1 \\ (\textbf{let}\ \mathsf{v} = 3\ \textbf{in}\ \mathsf{v} + 1)\ [\mathsf{v}\ /\ 2] & \Rightarrow \textbf{let}\ \mathsf{v} = 3\ \textbf{in}\ \mathsf{v} + 1 \end{array}$$

□

Variable Classification

An occurrence of a variable v is *bound* in x if it occurs on the right-hand side of a case alternative whose pattern includes v, as the argument of an enclosing lambda abstraction or as a binding in an enclosing let expression; all other variable occurences are *free*. The set of free variables of an expression e is denoted by $\mathsf{freeVars}\ \mathsf{e}$, and can be computed using the function in Figure 2.3.

In order to avoid accidental variable name clashes while performing transformations, we demand that all variables within a program are unique. All transformations may assume and should preserve this invariant.

```
freeVars :: Expr → [VarName]
freeVars ⟦v⟧              = [v]
freeVars ⟦c xs⟧           = freeVars′ xs
freeVars ⟦f xs⟧           = freeVars′ xs
freeVars ⟦x xs⟧           = freeVars x ∪ freeVars′ xs
freeVars ⟦λv → x⟧         = freeVars x \ [v]
freeVars ⟦let v = x in y⟧ = freeVars x ∪ (freeVars y \ [v])
freeVars ⟦case x of as⟧   = freeVars x ∪ ⋃(map f as)
  where f ⟦c vs → y⟧ = freeVars y \ vs

freeVars′ xs = ⋃(map freeVars xs)
```

Figure 2.3: Free variables of an expression.

2.1.2 Simplification Rules

We present several simplification rules in Figure 2.4, which can be applied to our Core language. These rules are standard and would be applied by any optimising compiler (Peyton Jones and Santos 1994). Some of the rules duplicate code, but none duplicate work. All the rules preserve both the semantics and the sharing behaviour of an expression. We believe the rules are confluent.

The (app-app), (fun-app) and (con-app) rules normalise applications. The (case-con) and (lam-app) rules simply follow the semantics, using let expressions to preserve the sharing. The (case-app), (let-case) and (case-case) rules move outer expressions over an inner case expression, duplicating the outer expression in each alternative. The (case-lam) rule promotes a lambda from inside a case alternative outwards. The (let-app) and (let-case) rules move an expression over an inner let expression. The (let) rule substitutes let expressions where the bound variable is used only once, and therefore no loss of sharing is possible.

2.2 Sharing

This section informally discusses the relevant sharing properties of Haskell. In general, any optimisation must take account of sharing, but semantic analysis can sometimes ignore the effects of sharing. The sharing present in Haskell is not specified in the Haskell Report (Peyton Jones 2003), but a

$$(\mathsf{x}\ \overline{\mathsf{xs}})\ \overline{\mathsf{ys}} \qquad \text{(app-app)}$$
$$\quad\Rightarrow \mathsf{x}\ (\overline{\mathsf{xs}} \mathbin{+\!\!+} \overline{\mathsf{ys}})$$

$$(\mathsf{f}\ \overline{\mathsf{xs}})\ \overline{\mathsf{ys}} \qquad \text{(fun-app)}$$
$$\quad\Rightarrow \mathsf{f}\ (\overline{\mathsf{xs}} \mathbin{+\!\!+} \overline{\mathsf{ys}})$$

$$(\mathsf{c}\ \overline{\mathsf{xs}})\ \overline{\mathsf{ys}} \qquad \text{(con-app)}$$
$$\quad\Rightarrow \mathsf{c}\ (\overline{\mathsf{xs}} \mathbin{+\!\!+} \overline{\mathsf{ys}})$$

$$\textbf{case}\ \mathsf{c}\ \overline{\mathsf{xs}}\ \textbf{of}\ \{\ldots; \mathsf{c}\ \overline{\mathsf{vs}} \rightarrow \mathsf{y}; \ldots\} \qquad \text{(case-con)}$$
$$\quad\Rightarrow \textbf{let}\ \overline{\mathsf{vs}} = \overline{\mathsf{xs}}\ \textbf{in}\ \mathsf{y}$$

$$(\lambda \mathsf{v} \rightarrow \mathsf{x})\ \mathsf{y} \qquad \text{(lam-app)}$$
$$\quad\Rightarrow \textbf{let}\ \mathsf{v} = \mathsf{y}\ \textbf{in}\ \mathsf{x}$$

$$(\textbf{case}\ \mathsf{x}\ \textbf{of}\ \{\mathsf{c}_1\ \overline{\mathsf{vs}}_1 \rightarrow \mathsf{y}_1; \ldots; \mathsf{c}_n\ \overline{\mathsf{vs}}_n \rightarrow \mathsf{y}_n\})\ \mathsf{z} \qquad \text{(case-app)}$$
$$\quad\Rightarrow \textbf{case}\ \mathsf{x}\ \textbf{of}\ \{\mathsf{c}_1\ \overline{\mathsf{vs}}_1 \rightarrow \mathsf{y}_1\ \mathsf{z}; \ldots; \mathsf{c}_n\ \overline{\mathsf{vs}}_n \rightarrow \mathsf{y}_n\ \mathsf{z}\}$$

$$(\textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ \mathsf{y})\ \mathsf{z} \qquad \text{(let-app)}$$
$$\quad\Rightarrow \textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ \mathsf{y}\ \mathsf{z}$$

$$\textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ (\textbf{case}\ \mathsf{y}\ \textbf{of}\ \{\mathsf{c}_1\ \overline{\mathsf{vs}}_1 \rightarrow \mathsf{y}_1; \ldots; \mathsf{c}_n\ \overline{\mathsf{vs}}_n \rightarrow \mathsf{y}_n\}) \qquad \text{(let-case)}$$
$$\begin{aligned} \quad\Rightarrow \textbf{case}\ \mathsf{y}\ \textbf{of}\ \{ & \mathsf{c}_1\ \overline{\mathsf{vs}}_1 \rightarrow \textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ \mathsf{y}_1 \\ & ; \ldots \\ & ; \mathsf{c}_n\ \overline{\mathsf{vs}}_n \rightarrow \textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ \mathsf{y}_n\} \end{aligned}$$

where v is not used in y

$$\textbf{case}\ (\textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ \mathsf{y})\ \textbf{of}\ \overline{\mathsf{as}} \qquad \text{(case-let)}$$
$$\quad\Rightarrow \textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ (\textbf{case}\ \mathsf{y}\ \textbf{of}\ \overline{\mathsf{as}})$$

$$\textbf{case}\ (\textbf{case}\ \mathsf{x}\ \textbf{of}\ \{\mathsf{c}_1\ \overline{\mathsf{vs}}_1 \rightarrow \mathsf{y}_1; \ldots; \mathsf{c}_n\ \overline{\mathsf{vs}}_n \rightarrow \mathsf{y}_n\})\ \textbf{of}\ \overline{\mathsf{as}} \qquad \text{(case-case)}$$
$$\begin{aligned} \quad\Rightarrow \textbf{case}\ \mathsf{x}\ \textbf{of}\ \{ & \mathsf{c}_1\ \overline{\mathsf{vs}}_1 \rightarrow \textbf{case}\ \mathsf{y}_1\ \textbf{of}\ \overline{\mathsf{as}} \\ & ; \ldots \\ & ; \mathsf{c}_n\ \overline{\mathsf{vs}}_n \rightarrow \textbf{case}\ \mathsf{y}_n\ \textbf{of}\ \overline{\mathsf{as}}\} \end{aligned}$$

$$\textbf{case}\ \mathsf{x}\ \textbf{of}\ \{\ldots; \mathsf{c}\ \overline{\mathsf{vs}} \rightarrow \lambda \mathsf{v} \rightarrow \mathsf{y}; \ldots\} \qquad \text{(case-lam)}$$
$$\begin{aligned} \quad\Rightarrow \lambda \mathsf{z} \rightarrow \textbf{case}\ \mathsf{x}\ \textbf{of} & \\ & \{\ldots \mathsf{z}; \mathsf{c}\ \overline{\mathsf{vs}} \rightarrow (\lambda \mathsf{v} \rightarrow \mathsf{y})\ \mathsf{z}; \ldots \mathsf{z}\} \end{aligned}$$

$$\textbf{let}\ \mathsf{v} = \mathsf{x}\ \textbf{in}\ \mathsf{y} \qquad \text{(let)}$$
$$\quad\Rightarrow \mathsf{y}\ [\mathsf{v}\ /\ \mathsf{x}]$$

where v occurs once in y, see §2.2

Figure 2.4: Simplification rules.

```
occurs :: VarName → Expr → Int
occurs v ⟦v′⟧               = if v ≡ v′ then 1 else 0
occurs v ⟦c xs⟧             = occurss v xs
occurs v ⟦f xs⟧             = occurss v xs
occurs v ⟦x xs⟧             = occurss v (x : xs)
occurs v ⟦λv′ → x⟧          = if v ≡ v′ then 0 else 2 ∗ occurs v x
occurs v ⟦let v′ = x in y⟧  = if v ≡ v′ then 0 else occurss v [x, y]
occurs v ⟦case x of as⟧     = occurs v x + maximum (map f as)
  where f ⟦c vs → y⟧ = if v ∈ vs then 0 else occurs v y

occurss v = sum ∘ map (occurs v)

linear :: VarName → Expr → Bool
linear v x = occurs v x ⩽ 1
```

Figure 2.5: Linear variables within an expression.

possible interpretation is defined elsewhere (Bakewell and Runciman 2000).

2.2.1 Let bindings

A let expression introduces *sharing* of the computational result of expressions.

Example 2

```
let x = f 1
in x + x
```

The evaluation of this expression results in:

```
(x + x) [x / f 1]
(f 1 + f 1)
```

The expression f 1 is reduced twice. However, a compiler would only evaluate f 1 once. The first time the value of x is demanded, f 1 evaluates to weak head normal form, and is bound to x. Any successive examinations of x return immediately, pointing at the same result. □

In general, the substitution of a bound variable by the associated expression may cause duplicate computation to be formed. However, in some circumstances, duplicate computation can be guaranteed not to occur. If a bound variable can be used at most once in an expression, it is said to be *linear*, and substitution can be performed. A variable is linear if it is used at most once, i.e. occurs at most once down each possible flow of control according to the definition in Figure 2.5.

2.2.2 Recursive let bindings

In the Haskell language, let bindings can be *recursive*. A recursive let binding is one where the local variable is in scope during the computation of its associated expression. The repeat function is often defined using a recursive let binding.

Example 3

```
repeat x = let xs = x : xs
           in xs
```

Here the variable xs is both defined and referenced in the binding. Given the application repeat 1, regardless of how much of the list is examined, the program will only ever create one single cons cell. This construct effectively ties a loop in the memory. □

Our Core language does not allow recursive let bindings, for reasons of simplicity. If there is a recursive binding to a function, it will be removed by lambda lifting (Johnsson 1985). To remove all recursive let bindings, we can replace value bindings with lambda expressions applied to dummy arguments, then lambda lift.

Example 3 (revisited)

Applying this algorithm to our example from before, we first add a lambda expression and a dummy argument:

```
repeat x = let xs = λdummy → x : xs dummy
           in xs dummy
```

Then we lambda lift:

$\mathsf{repeat\ x = f\ dummy\ x}$

$\mathsf{f\ dummy\ x = x : f\ dummy\ x}$

Optionally, we can remove the inserted dummy argument:

$\mathsf{repeat\ x = f\ x}$

$\mathsf{f\ x = x : f\ x}$

$\square$

In the repeat example we have lost sharing of the (:)-node. If a program consumes n elements of the list generated by the new repeat function, the space complexity will be $O(n)$, compared to $O(1)$ for the recursive let definition. The time complexity remains unchanged at $O(n)$, but the constant factor will be higher. However, in other examples, the time complexity may be worse.

Example 4

Consider the following program, where f is an expensive computation:

```
main x = let y = f x : y
         in y
```

We insert dummy arguments around recursive lets:

```
main x = let y = λdummy → f x : y dummy
         in y dummy
```

We have now changed the time complexity of the example. Originally f was performed once per call of main, in the revised code f will be performed once for each element of main demanded – an unbounded number of times, changing the complexity. $\square$

In practice, only a small number of programs make use of values bound in recursive lets, and nearly all of them are instances of repeat. However, it is possible to construct examples where the removal of recursive lets makes the computation significantly more expensive.

2.2.3 Constant Applicative Forms

A Constant Applicative Form (CAF) is a top level definition of zero arity. In Haskell, CAFs are computed at most once per program run, and retained as long as references to them remain.

Example 5

```
caf = expensive

main = caf + caf
```

A compiler will only compute expensive once. □

If a function with positive arity is inlined, this will not dramatically change the runtime behaviour of a program. If a CAF is inlined, this may have adverse effects on the performance.

2.3 Generating Core

In order to generate our Core language from the full Haskell language, we use the Yhc compiler (The Yhc Team 2007), a fork of nhc (Röjemo 1995).

The internal Core language of Yhc is PosLambda – a simple variant of lambda calculus without types, but with source position information. Yhc works by applying basic desugaring transformations, without optimisation. This simplicity ensures the generated PosLambda is close to the original Haskell in its structure. Each top-level function in a source file maps to a top-level function in the generated PosLambda, retaining the same name. However, PosLambda has constructs that have no direct representation in Haskell. For example, there is a FatBar construct (Peyton Jones 1987), used for compiling pattern matches which require fall through behaviour. We have therefore introduced a new Core language to Yhc, to which PosLambda can easily be translated (Golubovsky et al. 2007).

The Yhc compiler can generate the Core for a single source file. Yhc can also link in all definitions from all necessary libraries, producing a single Core file representing a whole program. All function and constructor names are fully qualified, so the linking process simply involves merging the list of functions from each required Core file.

In the process of generating a Core file, Yhc performs several transformations. Haskell's type classes are removed using the dictionary transformation (see §2.3.1). All local functions are lambda lifted, leaving only top-level functions – ensuring Yhc generated Core does *not* contain any lambda expressions. All constructor applications and primitive applications are fully applied.

2.3.1 The Dictionary Transformation

Most transformations in Yhc operate within a single function definition. The only phases which require information about more than one function are type checking and the transformation used to implement type classes (Wadler and Blott 1989). The dictionary transformation introduces tuples (or *dictionaries*) of methods passed as additional arguments to class-polymorphic functions. Haskell also allows subclassing. For example, Ord requires Eq for the same type. In such cases the dictionary transformation generates a nested tuple: the Eq dictionary is a component of the Ord dictionary.

Example 6

$\mathsf{f} :: \mathsf{Eq}\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \mathsf{Bool}$
$\mathsf{f\ x\ y} = \mathsf{x} \equiv \mathsf{y} \vee \mathsf{x} \not\equiv \mathsf{y}$

is translated by Yhc into

$\mathsf{f} :: (\alpha \rightarrow \alpha \rightarrow \mathsf{Bool}, \alpha \rightarrow \alpha \rightarrow \mathsf{Bool}) \rightarrow \alpha \rightarrow \alpha \rightarrow \mathsf{Bool}$
$\mathsf{f\ dict\ x\ y} = (\vee)\ (((\equiv)\ \mathsf{dict})\ \mathsf{x\ y})\ (((\not\equiv)\ \mathsf{dict})\ \mathsf{x\ y})$

$(\equiv)\ (\mathsf{a}, \mathsf{b}) = \mathsf{a}$
$(\not\equiv)\ (\mathsf{a}, \mathsf{b}) = \mathsf{b}$

The Eq class is implemented as two selector functions, $(\equiv)$ and $(\not\equiv)$, acting on a method table. For different types of α, different method tables are provided. □

The dictionary transformation is a global transformation. In Example 6 the Eq context in f not only requires a dictionary to be accepted by f; it requires all the callers of f to pass a dictionary as first argument. There are alternative approaches to implementing type classes, such as Jones (1994), which does not create a tuple of higher order functions. We use the dictionary transformation for simplicity, as it is already implemented within Yhc.

2.4 Homeomorphic Embedding

The homeomorphic embedding relation (Leuschel 2002) has been used to guarantee termination of certain program transformations (Sørensen and Glück 1995). The relation $x \trianglelefteq y$ indicates the expression x is an embedding of y. We can define $\trianglelefteq$ using the following rewrite rule:

$$emb = \{f(x_1, \ldots, x_n) \to x_i \,|\, 1 \leqslant i \leqslant n\}$$

Now $x \trianglelefteq y$ can be defined as $x \leftarrow^*_{emb} y$ (Baader and Nipkow 1998). The rule *emb* takes an expression, and replaces it with one of its immediate subexpressions. If repeated non-deterministic application of this rule to any subexpression transforms y to x, then $x \trianglelefteq y$. The intuition is that by removing some parts of y we obtain x, or that x is somehow "contained" within y.

Some examples:

$$\begin{array}{rlrl}
a \trianglelefteq & a & b(a) \ntrianglelefteq & a \\
a \trianglelefteq & b(a) & a \ntrianglelefteq & b(c) \\
c(a) \trianglelefteq & c(b(a)) & d(a,a) \ntrianglelefteq & d(b(a),c) \\
d(a,a) \trianglelefteq & d(b(a),c(c(a))) & b(a,a) \ntrianglelefteq & b(a,a,a)
\end{array}$$

Homeomorphic embedding $\trianglelefteq$ is a well-quasi order, meaning that for every infinite sequence of expressions $e_1, e_2 \ldots$ over a finite alphabet, there exist indicies $i < j$ such that $e_i \trianglelefteq e_j$. This result is known as Kruskal's Tree Theorem (Kruskal 1960). We can use this result to ensure an algorithm over expressions performs a bounded number of iterations, by stopping at iteration n once $\exists i \bullet 1 \leqslant i < n \wedge e_i \trianglelefteq e_n$.

```
data Shell α = Shell α [Shell α]

(⊴) :: Eq α ⇒ Shell α → Shell α → Bool
x ⊴ y = dive x y ∨ couple x y

dive x (Shell _ ys) = any (x⊴) ys

couple (Shell x xs) (Shell y ys) =
  x ≡ y ∧ length xs ≡ length ys ∧ and (zipWith (⊴) xs ys)
```

Figure 2.6: Homeomorphic embedding relation.

2.4.1 Homeomorphic Embedding of Core Expressions

Figure 2.6 gives an implementation of homeomorphic embedding in Haskell, making use of the auxiliary functions dive and couple (Leuschel 2002). The dive function checks if the first term is contained as a child of the second term, while the couple function checks if both terms have the same outer shell.

In order to perform homeomorphic embedding tests on expressions in our Core language, it is necessary to convert expressions to shells. To generate shells it is useful to have some sentinel value for expressions, we use the variable consisting of the empty string, which we represent as •. To convert an expression x to a Shell, we make the first field of Shell the expression x with all subexpressions replaced by •, and the second field a list of the shells of all the immediate subexpressions. Some examples:

```
shell ⟦v⟧                = Shell ⟦v⟧ [ ]
shell ⟦map f xs⟧         = Shell ⟦• • •⟧ [⟦map⟧, ⟦f⟧, ⟦xs⟧]
shell ⟦c xs⟧             = Shell ⟦• •⟧ [⟦c⟧, ⟦xs⟧]
shell ⟦λv → c xs⟧        = Shell ⟦λv → •⟧ [Shell ⟦• •⟧ [⟦c⟧, ⟦xs⟧]]
shell ⟦ let v = x in y⟧  = Shell ⟦ let v = • in •⟧ [⟦x⟧, ⟦y⟧]
```

To ensure that the first field in a Shell is drawn from a finite alphabet, we can replace any locally bound variables with the empty string. For example, shell ⟦v⟧ would become Shell ⟦•⟧ [ ].

2.4.2 Fast Homeomorphic Embedding

To compute whether $x \trianglelefteq y$, using the function in Figure 2.6, takes worse than polynomial time in the size of the expressions. Fortunately, there exists an algorithm (Stillman 1989; Narendran and Stillman 1987) which takes $O(\text{size}(x) \cdot \text{size}(y) \cdot a)$, where a is the maximum arity of any subexpression in x or y.

The faster algorithm first constructs a $\text{size}(x) \times \text{size}(y)$ table, recording whether each pair of subexpressions within x and y satisfy the homeomorphic embedding. By computing the homeomorphic embedding in a bottom-up manner, making use of the table to cache pre-computed results, much duplicate computation can be eliminated. By first assigning each subexpression a uniquely identifying number, table access and modification are both $O(1)$ operations. The result is a polynomial algorithm.

We have implemented the polynomial algorithm in Haskell. Haskell is not well-suited to the use of mutable arrays, so we have instead used tree data structures to model the table. In practical experiments, the table-based algorithm seems to perform around three times faster than the function in Figure 2.6. Comparing the complexity classes, we may have expected a greater speed-up, but it appears that the worst-case behaviour of the simple algorithm occurs infrequently.

Chapter 3

Boilerplate Removal

Generic traversals over recursive data structures are often referred to as *boilerplate* code. This chapter describes the Uniplate library, which offers a way to abstract several common forms of boilerplate code. The Uniplate library only supports traversals having value-specific behaviour for one type, and does not operate on functional values contained within a data structure. §3.1 gives an example problem, and our solution. §3.2 introduces the traversal combinators that we propose, along with short examples. §3.3 discusses how these combinators are implemented in terms of a single primitive. §3.4 extends this approach to multi-type traversals, and §3.5 covers the extended implementation. §3.6 investigates some performance optimisations. §3.7 gives comparisons with other approaches, using examples such as the "paradise" benchmark. §3.8 presents related work.

3.1 Introductory Example

Take a simple example of a recursive data type:

```
data Expr = Add Expr Expr | Val Int
          | Sub Expr Expr | Var String
          | Mul Expr Expr | Neg Expr
          | Div Expr Expr
```

The Expr type represents a small language for integer expressions, which permits free variables. Suppose we need to extract a list of all the variable occurrences in an expression:

```
variables :: Expr → [String]
variables (Var  x   ) = [x]
variables (Val  x   ) = [ ]
variables (Neg  x   ) = variables x
variables (Add x y) = variables x ++ variables y
variables (Sub  x y) = variables x ++ variables y
variables (Mul  x y) = variables x ++ variables y
variables (Div  x y) = variables x ++ variables y
```

This definition has the following undesirable characteristics: (1) adding a new constructor would require an additional equation; (2) the code is repetitive, the last four right-hand sides are identical; (3) the code cannot be shared with other similar operations. This problem is referred to as the *boilerplate* problem. Using the Uniplate library, the above example can be rewritten as:

```
variables :: Expr → [String]
variables x = [y | Var y ← universe x]
```

The type signature is optional, and would be inferred automatically if left absent. This example assumes a Uniplate instance for the Expr data type, given in §3.3.2. This example requires only Haskell 98. For more advanced examples we require multi-parameter type classes (Jones 2000) – but no functional dependencies, rank-2 types or generalised algebraic data types (GADTs).

The central idea is to exploit a common property of many traversals: they only require value-specific behaviour for a *single uniform type*. Looking at the variables example, the only type of interest is Expr. In practical applications, this pattern is common[1]. By focusing only on uniform type traversals, we are able to exploit well-developed techniques in list processing.

3.1.1 Contribution

Ours is far from the first technique for ‘scrapping boilerplate’. The area has been researched extensively. But there are a number of distinctive features in our approach:

[1]Most examples in boilerplate removal papers meet this restriction, even though the systems being discussed do not depend on it.

- We require *no language extensions* for single-type traversals, and only multi-parameter type classes for multi-type traversals.
- Our *choice of operations* is new: we shun some traditionally provided operations, and provide some uncommon ones.
- Our type classes can be defined independently *or* on top of Typeable and Data (Lämmel and Peyton Jones 2003), making *optional use of built-in compiler support.*
- We make use of *list-comprehensions* (Wadler 1987) for succinct queries.
- We *compare the conciseness* of operations using our library, by counting lexemes, showing our approach leads to less boilerplate.
- We *compare the performance* of traversal mechanisms, something that has been neglected in previous work.

3.2 Queries and Transformations

We define various traversals, using the Expr type defined in the introduction as an example throughout. We divide *traversals* into two categories: queries and transformations. A *query* is a function that takes a value, and extracts some information of a different type. A *transformation* takes a value, and returns a modified version of the original value. All the traversals rely on the class Uniplate, an instance of which is assumed for Expr. The definition of this class and its instances are covered in §3.3.

For some of the definitions we will make use of the terminology α-parent. The α-parent of a value

3.2.1 Children

The first function in the Uniplate library serves as both a function, and a definition of terminology:

children :: Uniplate $\alpha \Rightarrow \alpha \rightarrow [\alpha]$

The function children takes a value x, and returns the substructures of x with type α, that are not contained by any value of type α apart from x.

For example:

```
children (Add (Neg (Var "x")) (Val 12)) = [Neg (Var "x"), Val 12]
```

Note that Var "x" is *not* returned, as it is contained within Neg (Var "x"). The children function is occasionally useful, but is used more commonly as an auxiliary in the definition of other functions.

3.2.2 Queries

The Uniplate library provides the universe function to support queries.

```
universe :: Uniplate α ⇒ α → [α]
```

This function takes a data structure, and returns a list of *all* structures of the same type found within it, including the root. For example:

```
universe (Add (Neg (Var "x")) (Val 12)) =
  [Add (Neg (Var "x")) (Val 12)
  , Neg (Var "x")
  , Var "x"
  , Val 12]
```

One use of this mechanism for querying was given in the introduction. Using the universe function, queries can be expressed very concisely. Using a list-comprehension to process the results of universe is common.

Example 7

Consider the task of counting divisions by the literal 0.

```
countDivZero :: Expr → Int
countDivZero x = length [() | Div _ (Val 0) ← universe x]
```

Here we make essential use of a feature of list comprehensions: if a pattern does not match, then the item is skipped. In other syntactic constructs, failing to match a pattern results in a pattern-match error. □

3.2.3 Bottom-up Transformations

Another common operation provided by many boilerplate removal systems (Lämmel and Peyton Jones 2003; Visser 2004; Lämmel and Visser 2003; Ren and Erwig 2006) applies a given function to every subtree of the argument type. We define as standard a bottom-up transformation.

```
transform :: Uniplate α ⇒ (α → α) → α → α
```

The result of transform f x is f x′ where x′ is obtained by replacing each α-child x_i in x by transform f x_i.

Example 8

Suppose we wish to remove the Sub constructor assuming the equivalence: $x - y \equiv x + (-y)$. To apply this equivalence as a rewriting rule, at all possible places in an expression, we define:

```
simplify x = transform f x
  where f (Sub x y) = Add x (Neg y)
        f x         = x
```

This code can be read: apply the subtraction rule where you can, and where you cannot, do nothing. Adding more rules is easy. Take for example: $x + y = 2 * x$ **where** $x \equiv y$. Now we can add this new rule into our existing transformation:

```
simplify x = transform f x
  where f (Sub x y)          = Add x (Neg y)
        f (Add x y) | x ≡ y = Mul (Val 2) x
        f x                  = x
```

Each equation corresponds to the natural Haskell translation of the rule. The transform function manages all the required boilerplate. □

3.2.4 Top-Down Transformation

The Scrap Your Boilerplate approach (Lämmel and Peyton Jones 2003) (known as SYB) provides a top-down transformation named everywhere′. We describe this traversal, and our reasons for *not* providing it, even though it could easily be defined. We instead provide descend, based on the composOp operator (Bringert and Ranta 2006).

The everywhere′ f transformation applies f to a value, then recursively applies the transformation on all the children of the freshly generated value. Typically, the intention in a transformation is to apply f to *every node exactly once*. Unfortunately, everywhere′ f does not necessarily have this effect.

Example 9

Consider the following transformation:

```
doubleNeg (Neg (Neg x)) = x
doubleNeg x             = x
```

The intention is clear: remove all instances of double negation. When applied in a bottom-up manner, this is the result. But when applied top-down some nodes are missed. Consider the value Neg (Neg (Neg (Neg (Val 1)))); only the outermost double negation will be removed. □

Example 10

Consider the following transformation:

```
reciprocal (Div n m) = Mul n (Div (Val 1) m)
reciprocal x         = x
```

This transformation removes arbitrary division, converting it to divisions where the numerator is always 1. If applied once to each subtree, this computation would terminate successfully. Unfortunately, top-down transformation treats the generated Mul as being transformed, but cannot tell that the generated Div is the result of a transformation, not a fragment of the original input. This leads to a non-termination error. □

As these examples show, when defining top-down transformations using everywhere′ it is easy to slip up. The problem is that the program cannot tell the difference between freshly created constructors, and values that come originally from the input.

So we do support top-down transformations, but require the programmer to make the transformation more explicit. We introduce the descend function, inspired by the Compos paper (Bringert and Ranta 2006).

```
descend :: Uniplate α ⇒ (α → α) → α → α
```

The result of descend f x is obtained by replacing each outermost α-child x_i in x by f x_i. Unlike everywhere′, there is *no recursion* within descend.

Example 11

Consider the addition of a constructor Let String Expr Expr. Now let us define a function subst to replace free variables with given expressions. In order to determine which variables are free, we need to "remember" variables that are bound as we descend[2]. We can define subst using a descend transformation:

```
subst :: [(String, Expr)] → Expr → Expr
subst rep x =
  case x of
        Let name bind x → Let name (subst rep bind)
           (subst (filter ((≢ name) ∘ fst) rep) x)
        Var x → fromMaybe (Var x) (lookup x rep)
        _ → descend (subst rep) x
```

The Var alternative may return an Expr from rep, but no additional transformation is performed on this value, since all transformation is made explicit. In the Let alternative we explicitly continue the subst transformation. □

3.2.5 Transformations to a Normal Form

In addition to top-down and bottom-up transformations, we also provide transformations to a normal form. The idea is that a rule is applied exhaustively until a normal form is achieved. Consider a rewrite transformation:

```
rewrite :: Uniplate α ⇒ (α → Maybe α) → α → α
```

A rewrite-rule argument r takes an expression e of type α, and returns either Nothing to indicate that the rule is not applicable, or Just e′ indicating that

[2]For simplicity, we ignore issues of hygienic substitution that may arise if substituted expressions themselves contain free variables.

e is rewritten by r to e′. The intuition for rewrite r is that it applies r exhaustively; a postcondition for rewrite is that there must be no places where r could be applied. That is, the following property must hold:

```
propRewrite r x = all (isNothing ∘ r) (universe (rewrite r x))
```

One possible definition of the rewrite function uses transform:

```
rewrite :: Uniplate α ⇒ (α → Maybe α) → α → α
rewrite f = transform g
  where g x = maybe x (rewrite f) (f x)
```

This definition tries to apply the rule everywhere in a bottom-up manner. If at any point it makes a change, then the new value has the rewrite applied to it. The function only terminates when a normal form is reached.

The rewrite function has two potential problems. The first issue is that different application strategies may given results. Consider the rule replacing Neg (Neg x) with 1, applied to the value Neg (Neg (Neg (Val 1))). Depending on the application strategy, the result will be either Neg (Val 1) or Val 1. The second issue is that the implementation of rewrite given may check unchanged sub-expressions repeatedly, causing a performance problem. Both these issues can be avoided by using an explicit transformation, and managing the rewriting manually.

Bottom-Up Transformations to a Normal Form

An alternative way of obtaining a transformation to a normal form is to use the transform function directly. What restrictions on f ensure that transform f is idempotent, and hence a normal form? It is sufficient that the constructors on the right-hand side of f do not overlap with the constructors on the left-hand side.

Example 8 (revisited)

Recall the simplify transformation:

```
simplify = transform f
```

```
f (Sub x y)         = Add x (Neg y)
f (Add x y) | x ≡ y = Mul (Val 2) x
f x                 = x
```

Here Add occurs on the right-hand side of the first line, and on the left-hand side of the second. From this we can construct a value where transform f is not idempotent:

```
let x = Sub (Neg (Var "q")) (Var "q")

transform f x               ≡ Add (Neg (Var "q")) (Neg (Var "q"))
transform f (transform f x) ≡ Mul (Val 2) (Neg (Var "q"))
```

To remedy this situation, whenever the right-hand side of a rule applies a constructor of type Expr, f can be reapplied:

```
f (Sub x y)         = f $ Add x (f $ Neg y)
f (Add x y) | x ≡ y = f $ Mul (f $ Val 2) x
f x                 = x
```

Now we can guarantee that transform f is idempotent. In this particular example, we can inline the f applications attached to the constructors Neg, Val and Mul to give the more concise:

```
f (Sub x y)         = f $ Add x (Neg y)
f (Add x y) | x ≡ y = Mul (Val 2) x
f x                 = x
```

□

3.2.6 Action Transformations

Rewrite transformations apply a set of rules *repeatedly* until a normal form is found. One alternative is an action transformation, where each node is visited and transformed *once*, and state is maintained and updated as the operation proceeds. The standard technique is to thread a monad through the operation, which we do using transformM, with a bottom-up transformation strategy.

Example 12

Suppose we wish to rename each variable occurrence to be unique:

```
uniqueVars :: Expr → Expr
uniqueVars x = evalState (transformM f x) vars
  where
    vars = ['x' : show i | i ← [1..]]

    f (Var i) = do y : ys ← get
                   put ys
                   return (Var y)
    f x       = return x
```

The function transformM is a monadic variant of transform. Here a *state monad* is used to keep track of the list of names not yet used, with evalState computing the result of the monadic action, given an initial state vars. □

3.2.7 Paramorphisms

A paramorphism is a fold in which the recursive step may refer to the recursive components of a value, not just the results of folding over them (Meertens 1992). We define a similar recursion scheme in our library.

```
para :: Uniplate α ⇒ (α → [r] → r) → α → r
```

The para function uses the functional argument to combine a value, and the results of para on its children, into a new result.

Example 13

Compiler writers might wish to compute the *depth of expressions*:

```
depth :: Expr → Int
depth = para (λ_ cs → 1 + maximum (0 : cs))
```

□

3.2.8 Holes and Contexts

The final two operations in the library seem to be a novelty – we have not seen them in any other generics library, even in those which attempt to include all variations (Ren and Erwig 2006). These operations are similar to contextual pattern matching (Mohnen 1996), and have some connection to the zipper pattern (Huet 1997).

```
holes, contexts :: Uniplate α ⇒ α → [(α, α → α)]
```

Given a value y, these functions both return lists of pairs (x, f) where x is a sub-expression of y, and f replaces the hole in y from which x was removed. In the case of holes, x will be a member of children y, and for contexts, x will be a member of universe y.

Example 14

Suppose that mutation testing requires all expressions obtained by incrementing or decrementing *any single* literal in an original expression.

```
mutants :: Expr → [Expr]
mutants x = [c (Val j) | (Val i, c) ← contexts x, j ← [i − 1, i + 1]]
```

□

In general, these functions have the following properties:

```
propChildren x = children x ≡ map fst (holes x)
propId       x = all (≡ x) [b a | (a, b) ← holes x]
```

```
propUniverse x = universe x ≡ map fst (contexts x)
propId       x = all (≡ x) [b a | (a, b) ← contexts x]
```

3.2.9 Summary

We present signatures for all our methods in Figure 3.1, including several monadic variants. In our experience, the most commonly used operations are universe and transform, followed by transformM and descend.

```
module Data.Generics.Uniplate where

children   :: Uniplate α ⇒ α → [α]
contexts   :: Uniplate α ⇒ α → [(α, α → α)]
descend    :: Uniplate α ⇒ (α → α) → α → α
descendM   :: (Uniplate α, Monad m) ⇒ (α → m α) → α → m α
holes      :: Uniplate α ⇒ α → [(α, α → α)]
para       :: Uniplate α ⇒ (α → [r] → r) → α → r
rewrite    :: Uniplate α ⇒ (α → Maybe α) → α → α
rewriteM   :: (Uniplate α, Monad m) ⇒ (α → m (Maybe α)) → α → m α
transform  :: Uniplate α ⇒ (α → α) → α → α
transformM :: (Uniplate α, Monad m) ⇒ (α → m α) → α → m α
universe   :: Uniplate α ⇒ α → [α]
```

Figure 3.1: All Uniplate methods.

```
data Str α = Zero | One α | Two (Str α) (Str α)

instance Functor Str where
  fmap f (Zero   ) = Zero
  fmap f (One x  ) = One (f x)
  fmap f (Two x y) = Two (fmap f x) (fmap f y)

strList :: Str α → [α]
strList (Zero   ) = [ ]
strList (One x  ) = [x]
strList (Two x y) = strList x ++ strList y

listStr :: [α] → Str α
listStr [ ]      = Zero
listStr [x]      = One x
listStr (x : xs) = Two (One x) (listStr xs)
```

Figure 3.2: Str data type.

3.3 Implementing the Uniplate class

Requiring each instance of the Uniplate class to implement eleven separate methods would be an undue imposition. Instead, given a type specific instance for a *single* auxiliary method with a pair as result, we can define *all eleven* operations generically, at the class level. The auxiliary method is defined as:

```
uniplate :: Uniplate α ⇒ α → (Str α, Str α → α)
uniplate x = (cs, context)
```

The original Uniplate paper (Mitchell and Runciman 2007c) used lists of items, but we instead use the Str data type defined in Figure 3.2 to collect items. The use of Str simplifies the definition of instances and improves performance. The value cs contains the same elements as children x; the context is a function to generate a new value, with a different set of children. The caller of context must ensure that the value given to context has precisely the same structure of Str constructors as cs. The result pair splits the information in the value, but by combining the context with cs the original value can be recovered:

```
propId x = x ≡ context cs
  where (cs, context) = uniplate x
```

3.3.1 Operations in terms of uniplate

All eleven operations from §3.2 can be defined in terms of uniplate. We define all eleven operations in Figure 3.3. The common pattern is to call uniplate, then operate on the current children, often calling context to create a modified value. Some of these definitions can be made more efficient – see §3.6.1.

3.3.2 Writing Uniplate instances

We define a Uniplate instance for the Expr type in Figure 3.4.

The distinguishing feature of our library is that the children are defined in terms of their type. While this feature keeps the traversals simple, it does

```
children :: Uniplate α ⇒ α → [α]
children = strList ∘ fst ∘ uniplate

universe :: Uniplate α ⇒ α → [α]
universe x = x : concatMap universe (children x)

descend :: Uniplate α ⇒ (α → α) → α → α
descend f x = context $ fmap f children
  where (children, context) = uniplate x

transform :: Uniplate α ⇒ (α → α) → α → α
transform f = f ∘ descend (transform f)

rewrite :: Uniplate α ⇒ (α → Maybe α) → α → α
rewrite f = transform g
  where g x = maybe x (rewrite f) (f x)

descendM :: (Monad m, Uniplate α) ⇒ (α → m α) → α → m α
descendM f x = liftM context $ mapM f children
  where (children, context) = uniplate x

transformM :: (Monad m, Uniplate α) ⇒ (α → m α) → α → m α
transformM f x = f =≪ descendM (transformM f) x

rewriteM :: (Monad m, Uniplate α) ⇒ (α → m (Maybe α)) → α → m α
rewriteM f = transformM g
  where g x = f x ≫= maybe (return x) (rewriteM f)

para :: Uniplate α ⇒ (α → [r] → r) → α → r
para op x = op x $ map (para op) $ children x

holes :: Uniplate α ⇒ α → [(α, α → α)]
holes = f ∘ uniplate
  where f (Zero    , g) = []
        f (One x   , g) = [(x, g ∘ One)]
        f (Two l r, g) = f (l, g ∘ flip Two r) ++ f (r, g ∘ Two l)

contexts :: Uniplate α ⇒ α → [(α, α → α)]
contexts x = (x, id) : [(x2, g1 ∘ g2)
                | (x1, g1) ← holes x, (x2, g2) ← contexts x1]
```

Figure 3.3: Implementation of all Uniplate methods.

```
class Uniplate α where
  uniplate :: α → (Str α, Str α → α)


instance Uniplate Expr where
  uniplate (Neg a  ) = (One a, λ(One a') → Neg a')
  uniplate (Add a b) = (Two (One a) (One b)
                       , λ(Two (One a') (One b')) → Add a' b')
  uniplate (Sub a b) = (Two (One a) (One b)
                       , λ(Two (One a') (One b')) → Sub a' b')
  uniplate (Mul a b) = (Two (One a) (One b)
                       , λ(Two (One a') (One b')) → Mul a' b')
  uniplate (Div a b) = (Two (One a) (One b)
                       , λ(Two (One a') (One b')) → Div a' b')
  uniplate x         = (Zero, λZero → x)
```

Figure 3.4: The Uniplate class and an instance for Expr.

mean that rules for *deriving* instance definitions are not purely syntactic, but depend on the types of the constructors. We now describe the derivation rules, followed by information on the Derive tool that performs this task automatically. (If we are willing to make use of Multi-Parameter Type Classes, simpler derivation rules can be used: see §3.5.)

3.3.3 Derivation Rules

We model the derivation of an instance by describing a derivation from a data type to a set of declarations. The derivation rules are given in Figure 3.5. The $\mathcal{I}$ rule takes a concrete type, with no type variables, and generates an instance for the Uniplate class. The $\mathcal{D}$ rule takes a data type declaration, and defines a function over that data type. The $\mathcal{C}$ rule defines a case alternative for each constructor. The $\mathcal{T}$ rule defines type specific behaviour: a type is either the target type on which an instance is being defined, or a primitive such as Char, or an algebraic data type, or a free type variable.

The result of applying these functions to Expr is given in Figure 3.6. By applying simple transformation steps we can obtain the same instance as presented in Figure 3.4.

$$\begin{array}{l}
\mathcal{I}\llbracket \mathsf{d}\ \mathsf{t}_1 \ldots \mathsf{t}_n \rrbracket = \\
\quad \textbf{instance}\ \mathsf{Uniplate}\ (\mathsf{d}\ \mathsf{t}_1 \ldots \mathsf{t}_n)\ \textbf{where} \\
\qquad \mathsf{uniplate} = \mathcal{N}\llbracket \mathsf{d} \rrbracket\ \mathcal{T}\llbracket \mathsf{t}_1 \rrbracket \ldots \mathcal{T}\llbracket \mathsf{t}_n \rrbracket
\end{array}$$

$$\begin{array}{l}
\mathcal{D}\llbracket \textbf{data}\ \mathsf{d}\ \mathsf{v}_1 \ldots \mathsf{v}_n = \mathsf{a}_1 \ldots \mathsf{a}_m \rrbracket = \\
\qquad \mathcal{N}\llbracket \mathsf{d} \rrbracket\ \mathsf{v}_1 \ldots \mathsf{v}_n\ \mathsf{x} = \textbf{case}\ \mathsf{x}\ \textbf{of}\ \mathcal{C}\llbracket \mathsf{a}_1 \rrbracket \ldots \mathcal{C}\llbracket \mathsf{a}_m \rrbracket \\
\quad \textbf{where}\ \mathsf{x}\ \text{is fresh}
\end{array}$$

$$\begin{array}{l}
\mathcal{C}\llbracket \mathsf{c}\ \mathsf{t}_1 \ldots \mathsf{t}_n \rrbracket = \\
\qquad \mathsf{c}\ \mathsf{y}_1 \ldots \mathsf{y}_n \to (\mathsf{Zero}\ \text{`}\mathsf{Two}\text{`}\ \mathsf{a}_1\ \text{`}\mathsf{Two}\text{`} \ldots\ \text{`}\mathsf{Two}\text{`}\ \mathsf{a}_n \\
\qquad\qquad\qquad\quad , \lambda(\mathsf{Zero}\ \text{`}\mathsf{Two}\text{`}\ \mathsf{z}_1\ \text{`}\mathsf{Two}\text{`} \ldots\ \text{`}\mathsf{Two}\text{`}\ \mathsf{z}_n) \to \mathsf{c}\ (\mathsf{b}_1\ \mathsf{z}_1) \ldots (\mathsf{b}_n\ \mathsf{z}_n)) \\
\quad \textbf{where}\ \mathsf{y}_1 \ldots \mathsf{y}_n\ \text{ and } \mathsf{z}_1 \ldots \mathsf{z}_n\ \text{are fresh} \\
\qquad\qquad (\mathsf{a}_i, \mathsf{b}_i) = \mathcal{T}\llbracket \mathsf{t}_i \rrbracket\ \mathsf{y}_i
\end{array}$$

$$\begin{array}{ll}
\mathcal{T}\llbracket \mathsf{TargetType} \rrbracket & = \lambda \mathsf{x} \to (\mathsf{One}\ \mathsf{x}, \lambda(\mathsf{One}\ \mathsf{x}') \to \mathsf{x}') \\
\mathcal{T}\llbracket \mathsf{PrimitiveType} \rrbracket & = \lambda \mathsf{x} \to (\mathsf{Zero}, \lambda \mathsf{Zero} \to \mathsf{x}) \\
\mathcal{T}\llbracket \mathsf{d}\ \mathsf{t}_1 \ldots \mathsf{t}_n \rrbracket & = \mathcal{N}\llbracket \mathsf{d} \rrbracket\ \mathcal{T}\llbracket \mathsf{t}_1 \rrbracket \ldots \mathcal{T}\llbracket \mathsf{t}_n \rrbracket \\
\mathcal{T}\llbracket \mathsf{v} \rrbracket & = \mathsf{v}
\end{array}$$

$\mathcal{N}$ is an injection to fresh variables

Figure 3.5: Derivation rules for Uniplate instances.

3.3.4 Automated Derivation of uniplate

Applying these derivation rules is a form of boilerplate coding! The DrIFT tool (Winstanley 1997) derives instances automatically given rules depending only on the information contained in a type definition. However DrIFT is unable to operate with certain Haskell extensions (eg. TEX style literate Haskell), and requires a separate pre-processing stage.

In collaboration with Stefan O'Rear we have developed the Derive tool (Mitchell and O'Rear 2007), which is is based on Template Haskell (Sheard and Peyton Jones 2002). The Derive tool generates Uniplate instances by applying the rules from Figure 3.5, along with some simplification steps, replacing values such as Two x Zero with x.

Example 15

```
data Term = Name String
          | Apply Term Term
            deriving ({-! Uniplate !-})
```

```
𝓘⟦Expr⟧ =
  instance Uniplate Expr where
    uniplate = 𝓝⟦Expr⟧

𝓝⟦Expr⟧ x = case x of
  Val  y1     → (Zero `Two` a1, λ(Zero `Two` z1) → Val (b1 z1)
    where (a1, b1) = (λx → (Zero, λZero → x)) y1

  Var  y1     → (Zero `Two` a1, λ(Zero `Two` z1) → Var (b1 z1)
    where (a1, b1) = 𝓝⟦List⟧ y1

  Neg y1      → (Zero `Two` a1, λ(Zero `Two` z1) → Neg (b1 z1))
    where (a1, b1) = (λx → (One x, λ(One x′) → x′)) y1

  Add y1 y2 → (Zero `Two` a1 `Two` a2
               , λ(Zero `Two` z1 `Two` z2) → Neg (b1 z1) (b2 z2))
    where (a1, b1) = (λx → (One x, λ(One x′) → x′)) y1
          (a2, b2) = (λx → (One x, λ(One x′) → x′)) y2

    -- other constructors following the same pattern as Add ...

𝓝⟦List⟧ v1 x = case x of
  []          → (Zero, λ(Zero) → [])
  (:)  y1 y2 → (Zero `Two` a1 `Two` a2
              , λ(Zero `Two` z1 `Two` z2) → (:) (b1 z1) (b2 z2))
    where (a1, b1) = v1 y1
          (a2, b2) = 𝓝⟦List⟧ v1 y2
```

Figure 3.6: The result of applying $\mathcal{D}$ to Expr.

Running the Derive tool over this file, the generated code is:

```
instance Uniplate Term where
  uniplate (Apply x₁ x₂) = (Two (One x₁) (One x₂)
                           ,λ(Two (One x₁) (One x₂)) → Apply x₁ x₂)
  uniplate x             = (Zero, λ_ → x)
```

□

3.4 Multi-type Traversals

We have introduced the Uniplate class and an instance of it for type Expr. Now let us imagine that Expr is merely the expression type in a language with statements:

```
data Stmt = Assign   String Expr
          | Sequence [Stmt]
          | If       Expr   Stmt Stmt
          | While    Expr   Stmt
```

We could define a Uniplate instance for Stmt, and so perform traversals upon statements too. However, we may run into limitations. Consider the task of finding all literals in a Stmt – this requires boilerplate to find not just inner statements of type Stmt, but inner expressions of type Expr.

The Uniplate class takes a value of type α, and operates on its substructures of type α. What we now require is something that takes a value of type β, but operates on the children of type α within it – we call this class Biplate. Typically the type β will be a container of α. We can extend our operations by specifying how to find the α's within the β's, and then perform the standard Uniplate operations upon the α type. In the above example, $\alpha =$ Expr, and $\beta =$ Stmt.

We first introduce UniplateOn, which requires an explicit function to find the occurrences of type α within type β. We then make use of Multi-parameter type classes (MPTC's) to generalise this function into a type class, named Biplate.

3.4.1 The UniplateOn Operations

We define operations, including universeOn and transformOn, which take an extra argument relative to the standard Uniplate operators. We call this extra argument biplate: it is a function from the containing type (β) to the contained type (α).

```
type BiplateType β α = β → (Str α, Str α → β)
biplate :: BiplateType β α
```

The intuition for biplate is that given a structure of type β, the function should return the largest substructures in it of type α. Unlike uniplate, if $\alpha \equiv \beta$ the original value should be returned:

```
biplateSelf :: BiplateType α α
biplateSelf x = (One x, λ(One x′) → x′)
```

Unlike Uniplate, Biplate does *not* always descend. The idea is that Biplate is used once at the beginning of a traversal to find the values of type α, then Uniplate operates recursively descending through the data structure.

We can now define a selection of the On functions – the remaining functions from Figure 3.3 can be implemented in a similar way. Each takes a biplate function as an argument:

```
childrenOn :: Uniplate α ⇒ BiplateType β α → β → [α]
childrenOn biplate x = concatMap children $ strList $ fst $ biplate x

universeOn :: Uniplate α ⇒ BiplateType β α → β → [α]
universeOn biplate x = concatMap universe $ strList $ fst $ biplate x

descendOn :: Uniplate α ⇒ BiplateType β α → (α → α) → β → β
descendOn biplate f x = context $ fmap (descend f) cs
  where (cs, context) = biplate x

transformOn :: Uniplate α ⇒ BiplateType β α → (α → α) → β → β
transformOn biplate f x = context $ fmap (transform f) cs
  where (cs, context) = biplate x
```

These operations are similar to the original functions. They unwrap β values to find the α values within them, operate using the standard Uniplate operations for type α, then rewrap if necessary. If α is constant, there is another way to abstract away the biplate argument, as the following example shows.

Example 16

The Yhc.Core library (Golubovsky et al. 2007), part of the York Haskell Compiler (Yhc), makes extensive use of Uniplate. In this library, the central types include:

```
data Core     = Core String [String] [CoreData] [CoreFunc]

data CoreFunc = CoreFunc String String CoreExpr

data CoreExpr = CoreVar   String
              | CoreApp  CoreExpr [CoreExpr]
              | CoreCase CoreExpr [(CoreExpr, CoreExpr)]
              | CoreLet  [(String, CoreExpr)] CoreExpr
                  -- other constructors
```

Most traversals are performed on the CoreExpr type. However, it is often convenient to start from one of the other types. For example, coreSimplify :: CoreExpr → CoreExpr may be applied not just to an individual expression, but to all expressions in a function definition, or a complete program. If we are willing to freeze the type of the second argument to biplate as CoreExpr we can write a class UniplateExpr β corresponding to Biplate β CoreExpr:

```
class UniplateExpr β where
      uniplateExpr :: BiplateType β CoreExpr
```

We then need to write instances for the types we are interested in, mainly those which are likely to contain a CoreExpr within them. These instances must follow the same rules as for the Biplate class.

```
instance UniplateExpr Core where
  uniplateExpr (Core a b c d) = (cs, λcs → Core a b c (gen cs))
    where (cs, gen) = uniplateExpr d

instance UniplateExpr CoreFunc where
  uniplateExpr (CoreFunc a b c) = (cs, λcs → CoreFunc a b (gen cs))
    where (cs, gen) = uniplateExpr c

instance UniplateExpr CoreExpr where
  uniplateExpr = biplateSelf

instance UniplateExpr a ⇒ UniplateExpr [a] where
  uniplateExpr [] = (Zero, λZero → [])
  uniplateExpr (x : xs) = (Two a as, λ(Two n ns) → b n : bs ns)
    where (a, b) = uniplateExpr x
          (as, bs) = uniplateExpr xs
```

We can then implement traversal functions specific to CoreExpr in terms of the On functions:

```
childrenExpr  x = childrenOn  uniplateExpr x
universeExpr  x = universeOn  uniplateExpr x
descendExpr   x = descendOn   uniplateExpr x
transformExpr x = transformOn uniplateExpr x
  -- Similarly for all 11 functions in Figure 3.1
```

□

This technique has been used in the Yhc compiler. The Yhc compiler is written in Haskell 98 to allow for bootstrapping, so only the standard single-parameter type classes are available.

3.4.2 The Biplate class

If we are willing to make use of *multi-parameter type classes* (Jones 2000) we can define a class Biplate with biplate as its sole method. We do not require functional dependencies.

```
class Uniplate α ⇒ Biplate β α where
      biplate :: BiplateType β α
```

We can now implement universeBi and transformBi in terms of their On counterparts:

```
universeBi :: Biplate β α ⇒ β → [α]
universeBi = universeOn biplate

transformBi :: Biplate β α ⇒ (α → α) → β → β
transformBi = transformOn biplate
```

In general the move to Biplate requires few code changes, merely the use of the new set of Bi functions. To illustrate we give generalisations of two examples from previous sections, implemented using Biplate. We extend the variables and simplify functions to work on Expr, Stmt or many other types.

Example from §1 (revisited)

```
variables :: Biplate β Expr ⇒ β → [String]
variables x = [y | Var y ← universeBi x]
```

The equation requires only one change: the addition of the Bi suffix to universe. In the type signature we replace Expr with Biplate β Expr $\Rightarrow \beta$. Instead of requiring the input to be an Expr, we merely require that from the input we know how to reach an Expr. □

Example 8 (revisited)

```
simplify :: Biplate β Expr ⇒ β → β
simplify x = transformBi f x
  where f (Sub x y) = Add x (Neg y)
        f x         = x
```

In this redefinition we have again made a single change to the equation: the addition of Bi at the end of transform. □

3.5 Implementing Biplate

The complicating feature of biplate is that when defining Biplate where $\alpha \equiv \beta$ the function does not descend to the children, but simply returns its argument – requiring a check for type equality between values. This check can be captured either using the type system, or using the Typeable class (Lämmel and Peyton Jones 2003). We present three methods for defining a Biplate instance – offering a trade-off between performance, compatibility and volume of code.

1. Direct definition requires $O(n^2)$ instances, but offers the highest performance with the fewest extensions. See §3.5.1.

2. The Typeable class can be used, requiring $O(n)$ instances and no further Haskell extensions, but giving worse performance. See §3.5.2.

3. The Data class can be used, providing fully automatic instances with GHC, but requiring the use of rank-2 types, and giving the worst performance. See §3.5.3.

All three methods can be fully automated using the Derive tool, and all provide a simplified method for writing Uniplate instances. The Biplate class definition itself is independent of the method used to implement its

```
module Data.Generics.PlateDirect where

type Type β α = (Str α, Str α → β)

plate :: β → Type β α
plate f = (Zero, λ_ → f)

(|*) :: Type (α → β) α → α → Type β α
(|*) (xs, x') y = (Two xs (One y), λ(Two xs (One y)) → x' xs y)

(|+) :: Biplate τ α ⇒ Type (τ → β) α → τ → Type β α
(|+) (xs, x') y = (Two xs ys, λ(Two xs ys) → x' xs (y' ys))
  where (ys, y') = biplate y

(|-) :: Type (τ → β) α → τ → Type β α
(|-) (xs, x') y = (xs, λxs → x' xs y)

(||*) :: Type ([α] → β) α → [α] → Type β α
(||*) (xs, x') y = (Two xs (listStr y), λ(Two xs ys) → x' xs (strList ys))

(||+) :: Biplate τ α ⇒ Type ([τ] → β) α → [τ] → Type β α
(||+) (xs, x') y = (Two xs ys, λ(Two xs ys) → x' xs (y' ys))
  where (ys, y') = plateListDiff y
        plateListDiff [] = plate []
        plateListDiff (x : xs) = plate (:) |+ x ||+ xs
```

Figure 3.7: Implementation of PlateDirect.

instances. This abstraction allows the user to start with the simplest instance scheme available to them, then move to alternative schemes to gain increased performance or compatibility.

3.5.1 Direct instances

Writing direct instances requires the Data.Generics.PlateDirect module to be imported, whose implementation is given in Figure 3.7. This style requires a maximum of n^2 instance definitions, where n is the number of concrete types which contain each other, but gives the highest performance and most type-safety. The instances still depend on the type of each field, but are easier to define than the Uniplate instance discussed in §3.3.2. Here is a possible instance for the Expr type:

```
instance Uniplate Expr where
  uniplate (Neg a   ) = plate Neg |* a
  uniplate (Add a b) = plate Add |* a |* b
  uniplate (Sub  a b) = plate Sub |* a |* b
  uniplate (Mul  a b) = plate Mul |* a |* b
  uniplate (Div  a b) = plate Div  |* a |* b
  uniplate x          = plate x
```

Five infix combinators (|*, |+, |-, ||* and ||+) indicate the structure of the field to the right. The |* combinator says that the value on the right is of the target type, |+ says that a value of the target type *may occur* in the right operand, |- says that values of the target type *cannot occur* in the right operand. ||* and ||+ are versions of |* and |+ used when the value to the right is a *list* either of the target type, or of a type that may contain target values. The law plate f |- x ≡ plate (f x) justifies the definition presented above.

This style of definition naturally expands to the multi-type traversal. For example:

```
instance Biplate Stmt Expr where
  biplate (Assign    a b  ) = plate Assign     |- a |* b
  biplate (Sequence a     ) = plate Sequence ||+ a
  biplate (If          a b c) = plate If          |* a |+ b |+ c
  biplate (While      a b  ) = plate While       |* a |+ b
```

The information provided by uses of |- and |+ avoids redundant exploration down branches that do not have the target type. The use of ||* and ||+ avoid the definition of additional instances. The combinators are implemented in Figure 3.7, and work by building up the instance from left to right, adding successive fields.

Instances are given only on concrete types, containing no type variables. In the worst case, this approach requires a Biplate instance for each container/contained pair. In reality few traversal pairs are actually needed. The restricted pairing of types in Biplate instances also gives increased type safety; instances such as Biplate Expr Stmt do not exist.

In our experience definitions using these combinators offer similar performance to hand-tuned instances; see §3.7.2 for measurements.

```
module Data.Generics.PlateTypeable where

class PlateAll β α where
  plateAll :: β → Type β α

type Type β α = (Str α, Str α → β)

plate :: β → Type β α
plate x = (Zero, λ_ → x)

(|+) :: (Typeable τ, Typeable α, PlateAll τ α)
     ⇒ Type (τ → β) α → τ → Type β α
(|+) (xs, x′) y = (Two xs ys, λ(Two xs ys) → x′ xs (y′ ys))
  where (ys, y′) = plateSome y

instance (Typeable α, Typeable β, Uniplate α, PlateAll β α) ⇒
         Biplate β α where
  biplate = plateSome

plateSome :: (Typeable β, Typeable α, PlateAll β α) ⇒ β → Type β α
plateSome x = res
  where
    res = case asTypeOf (cast x) (Just $ head $ strList $ fst res) of
      Nothing → plateAll x
      Just y → (One y, λ(One y) → fromJust $ cast y)
```

Figure 3.8: Implementation of PlateTypeable.

3.5.2 Typeable based instances

Instead of writing $O(n^2)$ class instances to locate values of the target type, we can use the Typeable class to *test at runtime* whether we have reached the target type. This strategy is implemented in the Data.Generics.PlateTypeable module, whose implementation is given in Figure 3.8. The user is required to define an instance of the auxiliary class PlateAll, separating the fields of a constructor with |+:

```
instance (Typeable α, Uniplate α) ⇒ PlateAll Expr α where
  plateAll (Neg a   ) = plate Neg |+ a
  plateAll (Add a b) = plate Add |+ a |+ b
  plateAll (Sub a b) = plate Sub |+ a |+ b
  plateAll (Mul a b) = plate Mul |+ a |+ b
  plateAll (Div a b ) = plate Div  |+ a |+ b
  plateAll x          = plate x

instance (Typeable α, Uniplate α) ⇒ PlateAll Stmt α where
  plateAll (Assign    a b  ) = plate Assign    |+ a |+ b
  plateAll (Sequence a     ) = plate Sequence |+ a
  plateAll (If          a b c) = plate If          |+ a |+ b |+ c
  plateAll (While      a b  ) = plate While      |+ a |+ b
```

To give an instance of PlateAll β α, for a concrete type β, we require the context Typeable α and Uniplate α. To give an instance where β has type variables of kind $*$, we require the context Typeable τ and PlateAll τ α for all the type variables τ in β. We do not permit β to have any higher-kinded type variables. To give an example instance with type variables, the instance for lists is:

```
instance (PlateAll τ α, Typeable τ, Typeable α, Uniplate α) ⇒
            PlateAll [τ] α where
  plateAll [] = plate []
  plateAll (x : xs) = plate (:) |+ x |+ xs
```

From a PlateAll instance, a Biplate is inferred, using the code from Figure 3.8. The function plateAll always descends at least one level, then looks for values of the target type, corresponding to the functionality of uniplate. The function plateSome looks for the first values of the target type, corresponding to biplate. Unfortunately, we cannot infer an instance for Uniplate automatically, and must explicitly declare:

```
instance Uniplate Expr where
  uniplate = plateAll

instance Uniplate Stmt where
  uniplate = plateAll
```

The reader may wonder why we cannot define:

```
instance PlateAll α α ⇒ Uniplate α where
  uniplate = plateAll
```

Consider the Expr type. To infer Uniplate Expr we require an instance for PlateAll Expr Expr. But to infer this instance we require Uniplate Expr – which we are in the process of inferring! [3]

3.5.3 Using the Data class

The existing Data and Typeable instances provided by the SYB approach can also be used to define Uniplate instances:

```
import Data.Generics
import Data.Generics.PlateData

data Expr = ... deriving (Typeable, Data)
data Stmt = ... deriving (Typeable, Data)
```

The disadvantages of this approach are (1) *lack of type safety* – there are now Biplate instances for many pairs of types where one is not a container of the other; (2) *compiler dependence* – it will only work where Data.Generics is supported, namely GHC at the time of writing.[4] The clear advantage is that there is almost no work required to create instances.

How do we implement the Uniplate class instances? The implementation is given in Figure 3.9, making use of the class PlateAll from PlateTypeable in Figure 3.8. The operation to get the children can be done using gmapQ. The operation to replace the children is more complex, requiring a state monad to keep track of the items to insert.

[3]GHC has co-inductive or recursive dictionaries, but Hugs does not. To allow continuing compatibility with Hugs, and the use of fewer extensions, we require the user to write these explicitly for each type.

[4]Hugs supports the required rank-2 types for Data.Generics, but the work to port the library has not been done yet.

```
module Data.Generics.PlateData where

import Data.Generics.PlateTypeable

instance (Typeable α, Data α, Typeable β, Data β) ⇒ PlateAll β α where
  plateAll x = (children, context)
    where
      children = listStr $ concat $ gmapQ (strList ∘ fst ∘ plateSome) x

      context xs = evalState (gmapM f x) $ strList xs
      f y = do let (cs, con) = plateSome y
               (as, bs) ← liftM (splitAt $ length $ strList cs) get
               put bs
               return $ con $ listStr as

instance (Typeable α, Data α) ⇒ Uniplate α where
  uniplate = plateAll
```

Figure 3.9: Implementation of PlateData.

The code in Figure 3.9 is not optimised for speed. Uses of splitAt and length require the list of children to be traversed multiple times. Transforming between Str α and $[\alpha]$ is also inefficient. We discuss improvements in §3.6.2.

3.6 Performance Improvements

This section describes some of the performance improvements we have been able to make. First we focus on our optimisation of universe, using foldr/build fusion properties (Peyton Jones et al. 2001). Next we turn to our Data class based instances, improving them enough to outperform SYB itself.

3.6.1 Optimising the universe function

Our initial universe implementation was presented in §3.3.1 as:

```
universe :: Uniplate on ⇒ on → [on]
universe x = x : concatMap universe (children x)
```

A disadvantage is that concatMap produces and consumes a list at every level in the data structure. We can fix this by calling the uniplate method directly, and building the list with a tail:

```
universe :: Uniplate on ⇒ on → [on]
universe x = f (One x) [ ]
  where f (Zero     ) res = res
        f (One x    ) res = x : f (fst $ uniplate x) res
        f (Two x y) res = f x (f y res)
```

Now we only perform one reconstruction. We can do even better using GHC's list fusion (Peyton Jones et al. 2001). The user of universe is often a list comprehension, which is a *good consumer*. We can make f a *good producer*:

```
universe :: Uniplate on ⇒ on → [on]
universe x = build f
  where f cons nil = g cons nil (One x) nil
        g cons nil (Zero     ) res = res
        g cons nil (One x    ) res = x `cons` g cons nil (fst $ uniplate x) res
        g cons nil (Two x y) res = g cons nil x (g cons nil y res)
```

3.6.2 Optimising PlateData

Surprisingly, it is possible to layer Uniplate over the Data instances of SYB, with better performance than SYB itself. The first optimisation is to generate the two members of the uniplate pair with only one pass over the data value. We cannot use SYB's gmapM or gmapQ – we must instead use gfoldl directly. With this first improvement in place we perform much the same operations as SYB. But the overhead of structure creation in uniplate makes traversals about 10% slower than SYB.

The next optimisation relies on the extra information present in the Uniplate operations – namely the target type. A boilerplate operation walks over a data structure, looking for target values to process. In SYB, the target values may be of *any* type. For Uniplate the target is a *single uniform* type. If a value is reached which is not a container for the target type, no further exploration is required of the values children. Computing which types are containers for the target type can be done relatively easily in the SYB framework (Lämmel and Peyton Jones 2004):

```
data TypeBox = ∀ α ∘ (Typeable α, Data α) ⇒ TypeBox α

contains :: TypeBox → [TypeBox]
contains (TypeBox x) = if isAlgType dtyp then concatMap f ctrs else [ ]
  where
    f c = gmapQ TypeBox (asTypeOf (fromConstr c) x)
    ctrs = dataTypeConstrs dtyp
    dtyp = dataTypeOf x
```

A TypeBox can be thought of as storing a *type*, rather than a *value*. The TypeBox constructor stores the Data and Typeable instance information for a particular type, along with the value undefined of that type. The contains function takes a TypeBox and returns a TypeBox representing the type of each field for each possible constructor. For example, the type [Int] has two constructors, [ ] which has no fields, and (:) which has fields of type Int and [Int]:

```
contains (undefined :: [Int]) =
  [TypeBox (undefined :: Int), TypeBox (undefined :: [Int])]
```

The contains function determines the types directly contained by a data type. By taking the transitive closure we can determine all the types reachable from a particular type. Hence all types can be divided into three sets:

1. The singleton set containing the type of the target.
2. The set of other types which *may* contain the target type.
3. The set of other types which *do not* contain the target type.

We compute these sets for each type only once, and the cost of computing them is small. When examining a value, if its type is a member of set 3 we can prune the search. This trick is surprisingly effective. Take for example an operation over Bool on the value (True, "Haskell"). The SYB approach finds 16 subcomponents, Uniplate touches only 3 subcomponents.

With all these optimisations we can usually perform both queries and transformations faster than SYB. In the benchmarks we improve on SYB by between 30% and 225%, with an average of 145% faster. Full details are presented in §3.7.2.

3.7 Results and Evaluation

We evaluate our boilerplate reduction scheme in two ways: firstly by the *conciseness of traversals* using it (i.e. the amount of boilerplate it removes), and secondly by its *runtime performance*. We measure conciseness by counting lexemes – although we concede that some aspects of concise expression may still be down to personal preference. We give a set of nine example programs, written using Uniplate, SYB and Compos operations. We then compare both the conciseness and the performance of these programs. Other aspects, such as the clarity of expression, are not so easily measured. Readers can make their own assessment based on the full sources we give.

3.7.1 Boilerplate Reduction

As test operations we have taken the first three examples from this chapter, three from the Compos paper (Bringert and Ranta 2006), and the three given in the SYB paper (Lämmel and Peyton Jones 2003) termed the "Paradise Benchmark". In all cases the Compos, SYB and Uniplate functions are given an appropriately prefixed name. In some cases, a helper function can be defined in the same way in both SYB and Uniplate; where this is possible we have done so. Type signatures are omitted where the compiler is capable of inferring them. For SYB and Compos we have used definitions from the original authors where available, otherwise we have followed the guidelines and style presented in the corresponding paper.

Examples from this Chapter

Example from §3.1 (revisited)

```
uni_variables x = [y | Var y ← universe x]

syb_variables = everything (++) ([] `mkQ` f)
  where f (Var y) = [y]
        f _       = []

com_variables :: Expr a → [String]
com_variables x = case x of
  Var y → [y]
  _ → composOpFold [] (++) com_variables x
```

Only Compos needs a type signature, due to the use of GADTs. List comprehensions allow for succinct queries in Uniplate. □

Example 7 (revisited)

```
uni_zeroCount x = length [() | Div _ (Val 0) ← universe x]
```

```
syb_zeroCount = everything (+) (0 `mkQ` f)
  where f (Div _ (Val 0)) = 1
        f _               = 0

com_zeroCount :: Expr a → Int
com_zeroCount x = case x of
  Div y (Val 0) → 1 + com_zeroCount y
  _ → composOpFold 0 (+) com_zeroCount x
```

In the Uniplate solution the list of () is perhaps inelegant. However, Uniplate is the only scheme that is able to use the standard length function: the other two express the operation as a fold. Compos requires additional boilerplate to continue the operation on Div y. □

Example 8 (revisited)

```
simp (Sub x y)          = simp $ Add x (Neg y)
simp (Add x y) | x ≡ y  = Mul (Val 2) x
simp x                  = x

uni_simplify = transform simp

syb_simplify = everywhere (mkT simp)

com_simplify :: Expr a → Expr a
com_simplify x = case x of
  Sub a b → com_simplify $ Add (com_simplify a) (Neg (com_simplify b))
  Add a b → case (com_simplify a, com_simplify b) of
                 (a′, b′) | a′ ≡ b′    → Mul (Val 2) a′
                          | otherwise → Add a′ b′
  _ → composOp com_simplify x
```

This is a modified version of simplify discussed in §3.2.5. The two rules are applied everywhere possible. Compos does not provide a bottom-up transformation, so needs extra boilerplate. □

```
data Stm = SDecl   Typ Var | SAss     Var Exp
         | SBlock [Stm]    | SReturn Exp
data Exp = EStm Stm        | EAdd Exp Exp
         | EVar  Var       | EInt  Int
data Var = V String
data Typ = T_int           | T_float
```

Figure 3.10: Data type from Compos.

Multi-type examples from the Compos paper

The statement type manipulated by the Compos paper is given in Figure 3.10. The Compos paper translates this type into a GADT, while Uniplate and SYB both accept the definition as supplied.

As the warnAssign function from the Compos paper could be implemented much more neatly as a query, rather than a monadic fold, we choose to ignore it. We cover the remaining three functions.

Example 17 (rename)

```
ren (V x) = V ("_" ++ x)

uni_rename = transformBi ren

syb_rename = everywhere (mkT ren)

com_rename :: Tree c → Tree c
com_rename t = case t of
  V x → V ("_" ++ x)
  _ → composOp com_rename t
```

The Uniplate definition is the shortest, as there is only one constructor in type Var. As Compos redefines all constructors in one GADT, it cannot benefit from this knowledge. □

Example 18 (symbols)

```
uni_symbols x = [(v, t) | SDecl t v ← universeBi x]
```

```
syb_symbols = everything (++) ([] `mkQ` f)
  where f (SDecl t v) = [(v, t)]
        f _           = []
```

```
com_symbols :: Tree c → [(Tree Var, Tree Typ)]
com_symbols x = case x of
  SDecl t v → [(v, t)]
  _ → composOpMonoid com_symbols x
```

Whereas the Compos solution explicitly manages the traversal, the Uniplate solution is able to use the built-in universeBi function. The use of lists again benefits Uniplate over SYB. □

Example 19 (constFold)

```
optimise (EAdd (EInt n) (EInt m)) = EInt (n + m)
optimise x = x

uni_constFold = transformBi optimise

syb_constFold = everywhere (mkT optimise)

com_constFold :: Tree c → Tree c
com_constFold e = case e of
  EAdd x y → case (com_constFold x, com_constFold y) of
                  (EInt n, EInt m) → EInt (n + m)
                  (x', y') → EAdd x' y'
  _ → composOp com_constFold e
```

The constant-folding operation is a bottom-up transformation, requiring all subexpressions to have been transformed before an enclosing expression is examined. Compos only supports top-down transformations, requiring a small explicit traversal in the middle. Uniplate and SYB both support bottom-up transformations. □

The Paradise Benchmark from SYB

The Paradise benchmark was introduced in the SYB paper (Lämmel and Peyton Jones 2003). The data type is shown in Figure 3.11. The idea is that this data type represents an XML file, and a Haskell program is being

```
type Manager  = Employee
type Name     = String
type Address  = String
data Company  = C [Dept]
data Dept     = D Name Manager [Unit]
data Unit     = PU Employee | DU Dept
data Employee = E Person Salary
data Person   = P Name Address
data Salary   = S Integer
```

Figure 3.11: Paradise Benchmark data structure.

written to perform various operations over it. The Compos paper includes an encoding into a GADT, with tag types for each of the different types.

We have made one alteration to the data type: Salary is no longer of type Float but of type Integer. In various experiments we found that the rounding errors for floating point numbers made different definitions return different results.[5] This change is of no consequence to the boilerplate code.

Example 20 (increase)

The first function discussed in the SYB paper is increase. This function increases every item of type Salary by a given percentage. In order to fit with our modified Salary data type, we have chosen to increase all salaries by k.

```
incS k (S s) = S (s + k)

uni_increase k = transformBi (incS k)

syb_increase k = everywhere (mkT (incS k))

com_increase :: Integer → Tree c → Tree c
com_increase k c = case c of
  S s → S (s + k)
  _ → composOp (com_increase k) c
```

In the Compos solution all constructors belong to the same GADT, so instead of just matching on S, all constructors must be examined. □

[5]Storing your salary in a non-exact manner is probably not a great idea!

Example 21 (incrOne)

The incrOne function performs the same operation as increase, but only within a named department. The one subtlety is that if the named department has a sub-department with the same name, then the salaries of the sub-department should only be increased once. We are able to reuse the increase function from the previous section in all cases.

```
uni_incrOne d k = descendBi f
   where f x@(D n _ _) | n ≡ d       = uni_increase k x
                       | otherwise = descend f x
```

```
syb_incrOne :: Data a ⇒ Name → Integer → a → a
syb_incrOne d k x | isDept d x = syb_increase k x
                  | otherwise  = gmapT (syb_incrOne d k) x
   where isDept   d = False `mkQ` isDeptD d
         isDeptD d (D n _ _) = n ≡ d
```

```
com_incrOne :: Name → Integer → Tree c → Tree c
com_incrOne d k x = case x of
   D n _ _ | n ≡ d → com_increase k x
   _ → composOp (com_incrOne d k) x
```

The SYB solution has grown substantially more complex, requiring two different utility functions. In addition syb_incrOne now *requires* a type signature. Compos retains the same structure as before, requiring a case to distinguish between the types of constructor. For Uniplate we use descend rather than transform, to ensure no salaries are incremented twice. □

Example 22 (salaryBill)

The final function is one which sums all the salaries.

```
uni_salaryBill x = sum [s | S s ← universeBi x]
```

```
syb_salaryBill = everything (+) (0 `mkQ` billS)
   where billS (S s) = s
```

```
com_salaryBill :: Tree c → Integer
com_salaryBill x = case x of
   S s → s
   _ → composOpFold 0 (+) com_salaryBill x
```

Here the Uniplate solution wins by being able to use a list comprehension to select the salary value out of a Salary object. The Uniplate class is the only one that is able to use the standard Haskell sum function, not requiring an explicit fold. □

Uniplate compared to SYB and Compos

In order to measure conciseness of expression, we have taken the code for all solutions and counted the number of lexemes – using the lex function provided by Haskell. A table of results is given in Table 3.1. The definitions of functions shared between SYB and Uniplate are included in both measurements. For the incrOne function we have not included the code for increase as well.

The Compos approach requires much more residual boilerplate than Uniplate, particularly for queries, bottom-up transformations and in type signatures. The Compos approach also requires a GADT representation.

Compared with SYB, Uniplate seems much more similar. For queries, Uniplate is able to make use of list comprehensions, which produces shorter code and does not require encoding a manual fold over the items of interest. For transformations, typically both are able to use the same underlying operation, and the difference often boils down to the mkT wrappers in SYB.

All the Uniplate functions could be implemented in the SYB framework, using the Data and Typeable classes instead of Uniplate and Biplate. If this was done, then the SYB examples would be identical to the Uniplate examples, and consequently have identical lexeme counts.

3.7.2 Runtime Overhead

This section compares the speed of solutions for the nine examples given in the previous section, along with hand-optimised versions, using no boilerplate removal library. We use four Uniplate instances, provided by:

Manual: These are Uniplate and Biplate instances written by hand.

Direct: These instances use the direct combinators from §3.5.1.

Typeable: These instances use the Typeable combinators from §3.5.2.

	simp	var	zero	const	ren	syms	bill	incr	incr1
Lexemes									
Uniplate	40	12	18	27	16	17	13	21	30
SYB	43	29	29	30	19	34	21	24	56
Compos	71	30	32	54	27	36	25	33	40
Performance									
Uniplate Manual	1.26	1.31	1.89	1.25	1.25	1.33	2.18	1.28	1.15
Uniplate Direct	1.30	1.37	2.17	1.34	1.36	1.28	2.89	1.40	1.24
Compos	1.50	1.17	1.65	1.50	1.38	1.46	3.70	1.65	1.60
Uniplate Typeable	1.50	1.72	2.86	2.09	2.00	3.10	9.49	1.74	1.81
Uniplate Data	2.35	3.76	7.52	2.31	2.50	4.10	16.72	2.08	2.03
SYB	3.24	7.28	16.33	3.69	3.33	9.75	54.70	4.09	3.70

	Query	**Transform**	**All**
Lexemes			
Uniplate	60	134	194
SYB	113	172	285
Compos	123	225	348
Performance			
Uniplate Manual	1.68	1.24	1.43
Uniplate Direct	1.93	1.33	1.59
Compos	2.00	1.53	1.73
Uniplate Typeable	4.29	1.83	2.92
Uniplate Data	8.03	2.25	4.81
SYB	22.02	3.61	11.79

Lexemes are the number of lexemes for each of the solutions to the test problems using each of Uniplate, SYB and Compos. **Performance** is expressed as multiples of the run-time for a hand-optimised version not using any traversal library, with lower being better.

Table 3.1: Table of lexeme counts and runtime performance.

Data: These instances use the SYB Data instances directly, as described in §3.5.3.

For all data types we generate 100 values at random using QuickCheck (Claessen and Hughes 2000). In order to ensure a fair comparison, we define one data type which is the same as the original, and one which is a GADT encoding. All operations take these original data types, transform them into the appropriate structure, apply the operation and then unwrap them. We measure all results as multiples of the time taken for a hand-optimised version. We compiled all programs with GHC 6.8.2 and -O2 on Windows XP.

The results are presented in Table 3.1. Using Manual or Direct instances, Uniplate is slightly faster than Compos – but about 50% slower than hand-optimised versions. Using the Data instances provided by SYB, we are able to substantially outperform SYB itself! See §3.6 for details of some of the optimisations used.

3.8 Related Work

The Uniplate library is intended to be a way to remove the boilerplate of traversals from Haskell programs. It is far from the first library to attempt boilerplate removal.

3.8.1 The SYB library

The SYB library (Lämmel and Peyton Jones 2003) is perhaps the most popular boilerplate removal system in Haskell. One of the reasons for its success is tight integration with the GHC compiler, lowering the barrier to use. We have compared directly against traversals written in SYB in §3.7.1, and have also covered how to implement Uniplate in terms of SYB in §3.5.3. In our experience most operations are shorter and simpler than the equivalents in SYB, and we are able to operate without the extension of rank-2 types. Most of these benefits stem directly from our definition of children as being the children of the same uniform type, contrasting with the SYB approach of all direct children.

The SYB library is, however, more powerful than Uniplate. If you wish to visit values of different type in a single traversal, Uniplate is unsuitable. The Data and Typeable classes have also been pushed further in successive papers (Lämmel and Peyton Jones 2004, 2005), allowing operations such as runtime reflection on values, which Uniplate cannot provide.

3.8.2 The Compos library

The Compos library (Bringert and Ranta 2006) is another approach to the removal of boilerplate, requiring GADTs (Peyton Jones et al. 2006) along with rank-2 types. The Compos library requires an existing data type to be rewritten as a GADT. The conversion from standard Haskell data structures to GADTs currently presents several problems: they are GHC specific, deriving is not supported on GADTs, and GADTs require explicit type signatures. The Compos approach is also harder to write instances for, having no simple instance generation framework, and no automatic derivation tool (although one could be written). The inner composOp operator is very powerful, and indeed we have chosen to replicate it in our library as descend. But the Compos library is unable to replicate either universe or transform from our library.

3.8.3 The Stratego tool

The Stratego tool (Visser 2004) provides support for generic operations, focusing on both the operations and the strategies for applying them. This approach is performed in an *untyped* language, although a typed representation can be modelled (Lämmel 2003). Rather than being a Haskell library, Stratego implements a domain specific language that can be integrated with Haskell.

3.8.4 The Strafunski library

The Strafunski library (Lämmel and Visser 2003; Lämmel 2002) has two aspects: generic transformations and queries for trees of any type; and features to integrate components into a larger programming system. Generic operations are performed using strategy combinators which can define special case behaviour for particular types, along with a default to perform in other situations. The Strafunski library is integrated with Haskell, primarily providing support for generic programming in application areas that involve traversals over large abstract syntax trees.

3.8.5 The Applicative library

The Applicative library (McBride and Paterson 2007) works by threading an Applicative operation through a data structure, in a similar way to threading a Monad through the structure. There is additionally a notion of Traversable functor, which can be used to provide generic programming. While the Applicative library can be used for generic programming, this task was not its original purpose, and the authors note they have "barely begun to explore" its power as a generic toolkit.

3.8.6 Generic Programming

There are a number of other libraries which deal with generic programming, aimed more at writing *type generic* (or *polytypic*) functions, but which can be used for boilerplate removal. The *Haskell generics suite*[6] showcases several approaches (Weirich 2006; Hinze 2004; Hinze and Jeuring 2003).

[6]`http://darcs.haskell.org/generics/`

Chapter 4

Supercompilation

This chapter deals with developing a *supercompiler* for Haskell, which we have called Supero. We start with an introductory example in §4.1, then describe our supercompilation method in §4.2. We then give a number of benchmarks, comparing both against C (compiled with GCC) in §4.3 and Haskell (compiled with GHC) in §4.4. Finally, we review related work in §4.5.

4.1 Introductory Example

Haskell (Peyton Jones 2003) can be used in a highly declarative manner, to express specifications which are themselves executable. Take for example the task of counting the number of words in a file read from the standard input. In Haskell, one could write:

```
main = print ∘ length ∘ words =≪ getContents
```

From right to left, the getContents function reads the input as a list of characters, words splits this list into a list of words, length counts the number of words, and finally print writes the value to the screen.

An equivalent C program is given in Figure 4.1. Compared to the C program, the Haskell version is more concise and more easily seen to be correct. Unfortunately, the Haskell program (compiled with GHC (The GHC Team 2007)) is also three times slower than the C version (compiled with GCC). This slowdown is caused by several factors:

```
int main()
{
        int i = 0;
        int c, last_space = 1, this_space;
        while ((c = getchar()) != EOF) {
                this_space = isspace(c);
                if (last_space && !this_space)
                        i++;
                last_space = this_space;
        }
        printf("%i\n", i);
        return 0;
}
```

Figure 4.1: Word counting in C.

Intermediate Lists The Haskell program produces and consumes many intermediate lists as it computes the result. The getContents function produces a list of characters, words consumes this list and produces a list of lists of characters, length then consumes the outermost list. The C version uses no intermediate data structures.

Functional Arguments The words function is defined using the dropWhile function, which takes a predicate and discards elements from the input list until the predicate becomes true. The predicate is passed as an invariant function argument in all applications of dropWhile.

Laziness and Thunks The Haskell program proceeds in a lazy manner, first demanding one character from getContents, then processing it with each of the functions in the pipeline. At each stage, a lazy thunk for the remainder of each function is created.

Using Supero, we can eliminate all these overheads. We obtain a program that performs *faster* than the C version. The optimiser is based around the techniques of supercompilation (Turchin 1986), where some of the program is evaluated at compile time, leaving an optimised residual program.

Our goal is an automatic optimisation that makes high-level Haskell programs run as fast as low-level equivalents, eliminating the current need for hand-tuning and low-level techniques to obtain competitive performance. We require no annotations on any part of the program, including the library functions.

4.1.1 Contributions

- To our knowledge, this is the first time supercompilation has been applied to Haskell.
- We make careful study of the let expression, something absent from the core language of many other papers on supercompilation.
- We present an alternative generalisation step, based on a homeomorphic embedding (Leuschel 2002).

4.2 Supercompilation

Our supercompiler takes a Core program as input, in the format described in §2.1, and produces an equivalent Core program as output. To improve the program we do not make small local changes to the original, but instead *evaluate it at compile time* so far as possible, leaving a *residual program* to be run.

The general method of supercompilation is shown in Figure 4.2. Each function in the output program is an optimised version of some associated expression in the input program. Supercompilation starts at the `main` function, and supercompiles the expression associated with `main`. Once the expression has been supercompiled, the outermost shell of the expression becomes part of the residual program – making use of a Uniplate instance for our Core language (see Chapter 3). All the subexpressions are assigned names, and will be given definitions in the residual program. If any expression (up to α-renaming) already has a name in the residual program, then the same name is used. Each of these named inner expressions is then supercompiled as before.

The supercompilation of an expression proceeds by repeatedly inlining a function application until some termination criterion is met. Once the termination criterion holds, the expression is generalised before the outer shell of the expression becomes part of the residual program and all subexpressions are assigned names. After each inlining step, the expression is simplified using the standard simplification rules from §2.1.2, along with additional simplification rules from Figure 4.3. The additional simplification rules all reduce the sharing in an expression, but by small constant amounts, and

```
supercompile ()
  seen := { }
  bind := { }
  tie ({ }, main)

tie ($\rho$, x)
  if x $\notin$ seen then
    seen := seen $\cup$ {x}
    bind := bind $\cup$ {$\psi$(x) = $\lambda$freeVars(x) $\rightarrow$ drive($\rho$, x)}
  endif
  return ($\psi$(x) freeVars(x))

drive ($\rho$, x)
  if terminate($\rho$, x) then
    (cs, gen) = uniplate(generalise($\rho$, x))
    return gen(fmap (tie $\rho$) cs)
  else
    return drive($\rho \cup$ {x}, simplify(unfold(x)))
```

Where ψ is a mapping from expressions to function names, ρ is the termination context and freeVars(x) returns the free variables in x. This code is parameterised by: terminate which decides whether to stop supercompilation of this expression; generalise which generalises an expression before residuation; unfold which chooses a function application and unfolds it. The simplify function is the application of the simplification rules given in Figures 2.4 and 4.3.

Figure 4.2: The supercompile function.

```
case v of {...; c $\overline{vs}$ $\rightarrow$ x; ...}
  $\Rightarrow$ case v of {...; c $\overline{vs}$ $\rightarrow$ x [v / c $\overline{vs}$]; ...}

let v = x in y
  $\Rightarrow$ y [v / x]
  where x is a lambda or a variable

let v = c $x_1 \ldots x_n$ in y
  $\Rightarrow$ let $v_1 = x_1$ in
      ...
      let $v_n = x_n$ in
      y [v / c $v_1 \ldots v_n$]
  where $v_1 \ldots v_n$ are fresh
```

Figure 4.3: Additional simplification rules.

permit additional transformations. For example, the first rule will cause a fresh constructor application to be created inside a case alternative, instead of sharing the case scrutinee. There are three key decisions in the supercompilation of an expression:

1. Which function to inline.
2. What termination criterion to use.
3. What generalisation to use.

The original Supero work (Mitchell and Runciman 2007b) inlined following evaluation order (with the exception of let expressions), used a bound on the size of the expression to ensure termination, and performed no generalisation. First we give examples of our supercompiler in use, then we return to examine each of the three choices we have made.

4.2.1 Examples of Supercompilation

Example 23 (Supercompiling and Specialisation)

```
main as = map (λb → b + 1) as

map f cs = case cs of
               []     → []
               d : ds → f d : map f ds
```

There are two primary inefficiencies in this example: (1) the map function passes the f argument invariantly in every call; (2) the application of f is more expensive than if the function was known in advance.

In order to simplify the example, we begin supercompilation from the expression map (λb → b + 1) as, rather than main. Supercompilation proceeds by first applying ψ(map (λb → b + 1) as), to generate a fresh name, which we choose to be h_0. This new function h_0 then has the free variables of the original expression as arguments, namely as. We then apply drive to the RHS, first inlining the map application, then applying the simplification rules:

```
h₀ as = map (λb → b + 1) as
      = case as of
```

```
                [ ]     → [ ]
                d : ds → d + 1 : map (λb → b + 1) ds
```

We now have a **case** with a variable as the scrutinee at the root of the expression, which terminate determines cannot be reduced further, so we residuate the outer shell of the expression. When processing the expression map (λb → b + 1) ds the tie function spots this to be an α-renaming of the body of an existing generated function already in the seen set, namely h_0, so h_0 is used without applying drive:

```
h0 as = case as of
                [ ]     → [ ]
                d : ds → d + 1 : h0 ds
```

We have now specialised the higher-order argument, passing less data at runtime. □

Example 24 (Supercompiling and Deforestation)

The deforestation transformation (Wadler 1988) removes intermediate lists from a traversal. A similar result is obtained by applying supercompilation, as shown here. Consider the operation of mapping (∗2) over a list and then mapping (+1) over the result. The first map deconstructs one list, and constructs another. The second does the same.

```
main as = map (λb → b + 1) (map (λc → c ∗ 2) as)
```

We first assign a new name for the body of main, then choose to expand the outer call to map:

```
h0 as = case map (λc → c ∗ 2) as of
                [ ]     → [ ]
                d : ds → d + 1 : map (λb → b + 1) ds
```

Next the unfold function chooses to inline the map scrutinised by the case, then perform the **case**/**case** simplification, and finally residuate:

```
h0 as = case (case as of
                        [ ]     → [ ]
                        e : es → e ∗ 2 : map (λc → c ∗ 2) es) of
                [ ]     → [ ]
```

$$
\begin{array}{l}
\qquad \mathsf{d : ds} \rightarrow \mathsf{d} + 1 : \mathsf{map}\ (\lambda \mathsf{b} \rightarrow \mathsf{b} + 1)\ \mathsf{ds} \\[1em]
= \mathbf{case}\ \mathsf{as}\ \mathbf{of} \\
\qquad [\,] \qquad \rightarrow [\,] \\
\qquad \mathsf{d : ds} \rightarrow (\mathsf{d} * 2) + 1 : \mathsf{map}\ (\lambda \mathsf{b} \rightarrow \mathsf{b} + 1)\ (\mathsf{map}\ (\lambda \mathsf{c} \rightarrow \mathsf{c} * 2)\ \mathsf{ds}) \\[1em]
= \mathbf{case}\ \mathsf{as}\ \mathbf{of} \\
\qquad [\,] \qquad \rightarrow [\,] \\
\qquad \mathsf{d : ds} \rightarrow (\mathsf{d} * 2) + 1 : \mathsf{h}_0\ \mathsf{ds}
\end{array}
$$

Both intermediate lists have been removed, and the functional arguments to map have both been specialised. □

4.2.2 Which function to inline

During the supercompilation of an expression, at each step some function needs to be inlined. Which to choose? In most supercompilation work the choice is made following the runtime semantics of the program. But in a language with let expressions this may be inappropriate. If a function applied *in a let binding* is inlined, its application when reduced may be a constructor or lambda, which would then be substituted in the let body. However, if a function applied *in a let body* is inlined, the let body may now only refer to the let binding once, allowing the binding to be substituted. Let us take two expressions, based on intermediate steps obtained from real programs (word counting and prime number calculation respectively):

$$
\begin{array}{ll}
\mathbf{let}\ \mathsf{x} = (\equiv)\ \$\ 1 & \qquad \mathbf{let}\ \mathsf{x} = \mathsf{repeat}\ 1 \\
\mathbf{in}\ \mathsf{x}\ 1 : \mathsf{map\ x\ ys} & \qquad \mathbf{in}\ \mathsf{const}\ 0\ \mathsf{x} : \mathsf{map\ f\ x}
\end{array}
$$

In the first example, inlining ($) in the let binding gives $(\lambda \mathsf{x} \rightarrow 1 \equiv \mathsf{x})$, which is now a lambda and can be substituted for x, resulting in $((1 \equiv 1) : \mathsf{map}\ (\lambda \mathsf{x} \rightarrow 1 \equiv \mathsf{x})\ \mathsf{ys})$ after simplification. Now map can be specialised appropriately. Alternatively, expanding the map repeatedly would keep increasing the size of expression until the termination criterion was met, aborting the supercompilation of this expression without achieving specialisation.

Taking the second example, repeat can be inlined indefinitely. However, by unfolding the const we produce $\mathbf{let}\ \mathsf{x} = \mathsf{repeat}\ 1\ \mathbf{in}\ 0 : \mathsf{map\ f\ x}$. Since x is only used once we substitute it to produce $(0 : \mathsf{map}\ \mathsf{f}\ (\mathsf{repeat}\ 1))$, which can have an intermediate list removed.

Unfortunately these two examples seem to suggest different strategies for unfolding – unfold in the let binding or unfold in the let body. However, they do have a common theme – unfold the function that cannot be unfolded infinitely often. Our strategy can be defined by the unfold function:

```
unfold :: Expr → Expr
unfold x = head (filter (not ∘ terminate) xs ++ xs ++ [x])
  where xs = unfolds x

unfolds :: Expr → [Expr]
unfolds ⟦f $\overline{xs}$⟧ = [⟦(inline f) $\overline{xs}$⟧]
unfolds x = [gen y | (c, gen) ← holes x, y ← unfolds c]
```

The unfolds function computes all possible one-step inlinings, using an in-order traversal of the abstract syntax tree, making use of the holes function defined by Uniplate in §3.2.8. The unfold function chooses the first unfolding which does not cause the supercompilation to terminate. If no such expression exists, the first unfolding is chosen.

4.2.3 The Termination Criterion

The original Supero program used a size bound on the expression to determine when to stop. The problem with a size bound is that different programs require different bounds to ensure both timely completion at compile-time and efficient residual programs. Indeed, within a single program, there may be different elements requiring different size bounds – a problem exacerbated as the size and complexity of a program increases.

Our solution is to use the homeomorphic embedding relation (described in §2.4). We terminate the supercompilation of an expression y if on the chain of reductions from main to y (represented by ρ) we have encountered an expression x such that $\mathsf{x} \trianglelefteq \mathsf{y}$.

In addition to using the homeomorphic embedding, we also terminate if further unfolding cannot yield any improvement to the root of the expression, as calculated by simpleTerminate in Figure 4.4. For example, if the root of an expression is a constructor application, no further unfolding will change

```
simpleTerminate ⟦c _⟧              = True
simpleTerminate ⟦case ⟦v⟧ of _⟧ = True
simpleTerminate ⟦λ_ → _⟧          = True
simpleTerminate ⟦v⟧                = True
simpleTerminate _ = False
```

Figure 4.4: Simple Termination function.

the root constructor. When terminating for this reason, we always residuate the outer shell of the expression, without applying any generalisation.

4.2.4 Generalisation

When the termination criterion has been met, it is necessary to reduce the size of the current expression, so that the supercompilation terminates. We always residuate the outer shell of the expression, but first we attempt to generalise the expression to improve subsequent optimisation. The simplest generalisation strategy is to do nothing, and always residuate the outer shell. We now introduce and compare two additional strategies, the first based on the most specific generalisation, and a second of our own creation.

The additional strategies work by using both the current expression x, and the expression y which caused the termination criteria to apply, where $\mathsf{y} \in \rho$ and $\mathsf{y} \trianglelefteq \mathsf{x}$. The generalisation must return an expression equivalent to x, but aims to have similarities with y after residuation of the outer shell. By being similar to y, the parts of the expression that occur repeatedly are not split apart by residuation and can be optimised well.

Most Specific Generalisation

The paper by Sørensen and Glück (1995) provides a method for generalisation, which works by taking the most specific generalisation of the current expression and an expression which is a homeomorphic embedding of it.

The most specific generalisation of two expressions s and t, msg(s, t), is produced by applying the following rewrite rule to the initial triple $(v, \{v = s\}, \{v = t\})$, resulting in a common expression and two sets of bindings.

$$\begin{pmatrix} t_g & & \\ \{v = \sigma(s_1, \ldots, s_n)\} & \cup & \theta_1 \\ \{v = \sigma(t_1, \ldots, t_n)\} & \cup & \theta_2 \end{pmatrix} \rightarrow \begin{pmatrix} t_g[v/\sigma(y_1, \ldots, y_n)] & & \\ \{y_1 = s_1, \ldots, y_n = s_n\} & \cup & \theta_1 \\ \{y_1 = t_1, \ldots, y_n = t_n\} & \cup & \theta_2 \end{pmatrix}$$

We can now write generalise as follows:

```
generalise $\rho$ x = $[\![$let bind in x'$]\!]$
    where (x', bind, _) = msg x (head [y | y $\leftarrow \rho$, y $\trianglelefteq$ x])
```

The generalise function applies msg and creates a let expression containing the binding produced. For example, given Just $(1 : [\,])$ as the current expression, and Just $[\,]$ as the expression from ρ which is a homeomorphic embedding of it, we would obtain **let** v $= 1 : [\,]$ **in** Just v. This method factors out similar parts of the two expressions, starting from the root.

Our Generalisation

Our generalisation is characterised by $x \bowtie y$, which produces an expression equivalent to y, but similar in structure to x.

```
$x \bowtie \sigma^*(y)$, if dive$(x, \sigma^*(y)) \wedge$ couple$(x, y)$
    let f = $\lambda\overline{\text{vs}} \rightarrow$ x in $\sigma^*$(f $\overline{\text{vs}}$)
    where $\overline{\text{vs}}$ = freeVars$(y)\backslash$freeVars$(\sigma^*(y))$

$x \bowtie y$, if couple$(x, y)$
    let $\theta_2$ in $t_g$
    where $(t_g, \theta_1, \theta_2)$ = msg$(x, y)$
```

We use $\sigma^*(y)$ to denote a subexpression y within a containing context σ^*. The first rule applies if the homeomorphic embedding first applied the dive rule. The idea is to descend to the element which matched, and then promote this to the top-level using a lambda. The second rule applies the most specific generalisation operation if the coupling rule was applied first.

We can now write generalise as:

```
generalise $\rho$ x = head [y | y $\leftarrow \rho$, y $\trianglelefteq$ x] $\bowtie$ x
```

Compared the to generalisation using msg alone, our method is able to factor out similar expressions which are not at the root of the expression.

For example, given Just (1 : [ ]) as the current expression, and [ ] as the expression from ρ which is a homeomorphic embedding of it, we would obtain **let** v = [ ] **in** Just (1 : v).

Comparison of Generalisations

Some examples of the msg function, and our $\bowtie$ operator, are:

Embedding	msg	$\bowtie$
$a \trianglelefteq b(a)$	$(x\ \ , \{x = a\}\ \ , \{x = b(a)\}\ \ )$	**let** $f = b(a)$ **in** f
$c(b) \trianglelefteq c(a(b))$	$(c(x), \{x = b\}\ \ , \{x = a(b)\}\ \ )$	**let** $x = a(b)$ **in** $c(x)$
$b(a) \trianglelefteq c(b(a))$	$(x\ \ , \{x = b(a)\}, \{x = c(b(a))\})$	**let** $f = b(a)$ **in** $c(f)$

We now show an example where most specific generalisation fails to produce the ideal generalised version.

Example 25

```
case putStr (repeat '1') r of
        (r, _) → (r, ())
```

This expression (which we name x) prints an infinite stream of 1's. The pairs and r's correspond to the implementation of GHC's IO Monad (Peyton Jones 2002). After several unrollings, we obtain the expression (named x'):

```
case putChar '1' r of
        (r, _) → case putStr (repeat '1') r of
                        (r, _) → (r, ())
```

The homeomorphic embedding $x \trianglelefteq x'$ matches, detecting an occurrence of the **case** putStr ... expression, and the supercompilation of x' is stopped. The most specific generalisation rule is applied as msg(x, x') and produces:

```
let a = putChar
    b = '1'
    c = λr → case putStr (repeat '1') r of
                        (r, _) → (r, ())
in case a b r of
            (r, _) → c r
```

The problem is that msg works from the top, looking for a common root of both expression trees. However, if the first rule applied by $\trianglelefteq$ was dive, the roots may be unrelated. Using our generalisation, $x \bowtie x'$:

```
let x = λr → case putStr (repeat '1') r of
                     (r, _) → (r, ())
in case putChar '1' r of
        (r, _) → x r
```

Our generalisation is superior because it has split out the putStr application *without* lifting the putChar application or the constant '1'. The putChar application can now be supercompiled further in the context of the case expression. □

4.3 Performance Compared With C Programs

The benchmarks we have used as motivating examples are inspired by the Unix wc command – namely character, word and line counting. We require the program to read from the standard input, and write out the number of elements in the file. To ensure that we test computation speed, not IO speed (which is usually determined by the buffering strategy, rather than optimisation) we demand that all input is read using the standard C getchar function only. Any buffering improvements, such as reading in blocks or memory mapping of files, could be performed equally in all compilers.

All the C versions are implemented following a similar pattern to Figure 4.1. Characters are read in a loop, with an accumulator recording the current value. Depending on the program, the body of the loop decides when to increment the accumulator. The Haskell versions all follow the same pattern as in the Introduction, merely replacing words with lines, or removing the words function for character counting.

We performed all benchmarks on a machine running Windows XP, with a 3GHz processor and 1Gb RAM. All benchmarks were run over a 50Mb log file, repeated 10 times, and the lowest value was taken. The C versions used GCC[1] version 3.4.2 with -O3. The Haskell version used GHC 6.8.1 with -O2. The Supero version was compiled using our optimiser, then written back as a Haskell file, and compiled once more with GHC 6.8.1 and -O2.

[1]http://gcc.gnu.org/

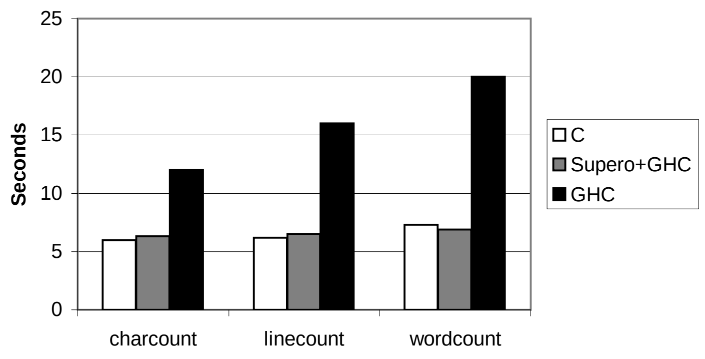


Figure 4.5: Benchmarks with C, Supero+GHC and GHC alone.

The results are given in Figure 4.5. In all the benchmarks C and Supero+GHC are within 10% of each other, while GHC trails further behind.

4.3.1 Identified Haskell Speedups

During initial trials using these benchmarks, we identified two unnecessary bottlenecks in the Haskell version of word counting. Both were remedied before the presented results were obtained.

Slow isSpace function The first issue is that isSpace in Haskell is much more expensive than `isspace` in C. The simplest solution is to use a FFI (Foreign Function Interface) (Peyton Jones 2002) call to the C `isspace` function in all cases, removing this factor from the benchmark. A GHC bug (number 1473) has been filed about the slow performance of isSpace.

Inefficient words function The second issue is that the standard definition of the words function (given in Figure 4.6) performs two additional isSpace tests per word. By appealing to the definitions of dropWhile and break it is possible to show that in words the first character of x is not a space, and that if y is non-empty then the first character is a space. The revised words′ function uses these facts to avoid the redundant isSpace tests.

```
words :: String → [String]
words s = case dropWhile isSpace s of
                [] → []
                x  → w : words y
                     where (w, y) = break isSpace x

words′ s = case dropWhile isSpace s of
                []     → []
                x : xs → (x : w) : words′ (drop1 z)
                         where (w, z) = break isSpace xs

drop1 []       = []
drop1 (x : xs) = xs
```

Figure 4.6: The words function from the Haskell standard libraries, and an improved words′.

4.3.2 Potential GHC Speedups

We have identified three factors limiting the performance of residual programs when compiled by GHC. These problems cannot be solved at the level of Core transformations. We suspect that by fixing these problems, the Supero execution time would improve by between 5% and 15%.

Strictness inference The GHC compiler is overly conservative when determining strictness for functions which use the FFI (GHC bug 1592). The `getchar` function is treated as though it may raise an exception, and terminate the program, so strict arguments are not determined to be strict. If GHC provided some way to mark an FFI function as not generating exceptions, this problem could be solved. The lack of strictness information means that in the line and word counting programs, every time the accumulator is incremented, the number is first unboxed and then reboxed (Peyton Jones and Launchbury 1991).

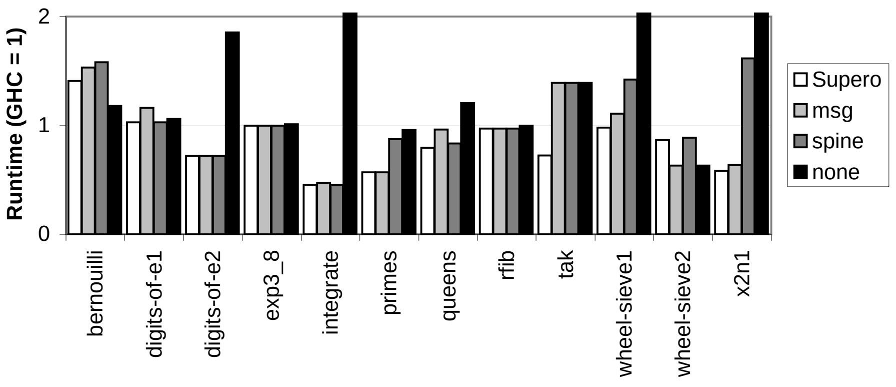
Heap checks The GHC compiler follows the standard STG machine (Peyton Jones 1992) design, and inserts heap checks before allocating memory. The purpose of a heap check is to ensure that there is sufficient memory on the heap, so that allocation of memory is a cheap operation guaranteed to succeed. GHC also attempts to lift heap checks: if two branches of a case expression both have heap checks, they are replaced with one shared heap check before the case expression. Unfortunately, with lifted heap checks, a tail-recursive function that allocates memory only upon exit can have the heap test executed on every iteration (GHC bug 1498). This problem affects the character counting example, but if the strictness problems were solved, it would apply equally to all the benchmarks.

Stack checks The final source of extra computation relative to the C version are stack checks. Before using the stack to store arguments to a function call, a test is performed to check that there is sufficient space on the stack. Unlike the heap checks, it is necessary to analyse a large part of the flow of control to determine when these checks are unnecessary. It is not clear how to reduce stack checks in GHC.

4.3.3 The Wordcount Benchmark

The most curious result is that Supero outperforms C on wordcounting, by about 6% – even with the problems discussed! The C program presented in Figure 4.1 is not optimal. The variable `last_space` is a boolean, indicating whether the previous character was a space, or not. Each time round the loop a test is performed on `last_space`, even though its value was determined and tested on the previous iteration. The way to optimise this code is to have two specialised variants of the loop, one for when `last_space` is true, and one for when it is false. When the value of `last_space` changes, the program would transition to the other loop. This pattern effectively encodes the boolean variable in the program counter, and is what the Haskell program has managed to generate from the high-level code.

However, in C it is quite challenging to capture the required control flow. The program needs two loops, where both loops can transition to the other. Using `goto` turns off many critical optimisations in the C compiler. Tail recursion is neither required by the C standard, nor supported by most compilers. The only way to express the necessary pattern is using nested while loops, but unlike newer imperative languages such as Java, C does not have named loops – so the inner loop cannot break from the outer loop if it reaches the end of the file. The only solution is to place the nested while loops in a function, and use `return` to break from the inner



Supero uses the $\bowtie$ generalisation method; **msg** uses the msg function for generalisation; **spine** applies no generalisation operation; **none** never performs any inlining.

Figure 4.7: Runtime, relative to GHC being 1.

loop. This solution would not scale to a three-valued control structure, and substantially increases the complexity of the code.

4.4 Performance Compared With GHC Alone

The standard set of Haskell benchmarks is the nofib suite (Partain et al. 2008). It is divided into three categories of increasing size: imaginary, spectral and real. Even small Haskell programs increase in size substantially once libraries are included, so we have limited our attention to the benchmarks in the imaginary section. All benchmarks were run with parameters that require runtimes of between 3 and 5 seconds for GHC.

We exclude two benchmarks, paraffins and gen_regexps. The paraffins benchmark makes substantial use of arrays, and we have not yet mapped the array primitives of Yhc onto those of GHC, which is necessary to run the transformed result. The gen_regexps benchmark tests character processing: for some reason (as yet unknown) the supercompiled executable fails.

The results of these benchmarks are given in Figure 4.7, along with detailed breakdowns in Table 4.1. All results are relative to the runtime of a program compiled with GHC -O2, lower numbers being better. The first three variants (Supero, msg, spine) all use homeomorphic embedding as the ter-

Program	**Supero**	**msg**	**spine**	**none**	**Size**	**Memory**
bernouilli	1.41	1.53	1.58	1.18	1.10	0.97
digits-of-e1	1.03	1.16	1.03	1.06	1.01	1.11
digits-of-e2	0.72	0.72	0.72	1.86	1.00	0.84
exp3_8	1.00	1.00	1.00	1.01	0.99	1.00
integrate	0.46	0.47	0.46	4.01	1.02	0.08
primes	0.57	0.57	0.88	0.96	1.00	0.98
queens	0.79	0.96	0.83	1.21	1.01	0.85
rfib	0.97	0.97	0.97	1.00	1.00	1.08
tak	0.72	1.39	1.39	1.39	1.00	1.00
wheel-sieve1	0.98	1.11	1.42	5.23	1.19	2.79
wheel-sieve2	0.87	0.63	0.89	0.63	1.49	2.30
x2n1	0.58	0.64	1.61	3.04	1.09	0.33

Program is the name of the program; **Supero** uses the $\bowtie$ generalisation method; **msg** uses the msg function for generalisation; **spine** applies no generalisation operation; **none** never performs any inlining; **Size** is the size of the Supero generated executable; **Memory** is the amount of memory allocated on the heap by the Supero executable.

Table 4.1: Runtime, relative to GHC being 1.

mination criterion, and $\bowtie$, msg or nothing respectively as the generalisation function. The final variant, none, uses a termination test that always causes a residuation. The 'none' variant is useful as a control to determine which improvements are due to bringing all definitions into one module scope, and which are a result of supercompilation. Compilation times ranged from a few seconds to twelve minutes.

The Bernouilli benchmark is the only one where Supero is slower than GHC by more than 3%. The reason for this anomaly is that a dictionary is referred to in an inner loop which is specialised away by GHC, but not by Supero.

With the exception of the wheel-sieve2 benchmark, our $\bowtie$ generalisation strategy performs as well as, or better than, the alternatives. While the msg generalisation performs better than the empty generalisation on average, the difference is not as dramatic.

4.4.1 GHC's optimisations

For these benchmarks it is important to clarify which optimisations are performed by GHC, and which are performed by Supero. The 'none' results show that, on average, taking the Core output from Yhc and compiling with GHC does *not* perform as well as the original program compiled using GHC. GHC has two special optimisations that work in a restricted number of cases, but which Supero-produced Core is unable to take advantage of.

Dictionary Removal Functions which make use of type classes are given an additional dictionary argument. In practice, GHC specialises many such functions by creating code with a particular dictionary frozen in. This optimisation is specific to type classes – a tuple of higher order functions is not similarly specialised. After compilation with Yhc, the type classes have already been converted to tuples, so Supero must be able to remove the dictionaries itself. One benchmark where dictionary removal is critical is digits-of-e2.

List Fusion GHC relies on names of functions, particularly foldr/build (Peyton Jones et al. 2001), to apply special optimisation rules such as list fusion. Many of GHC's library functions, for example iterate, are defined in terms of foldr to take advantage of these special properties. After transformation with Yhc, these names are destroyed, so no rule based optimisation can be performed. One example where list fusion is critical is primes, although it occurs in most of the benchmarks to some extent.

4.4.2 Compile Time

The compile times for some of the benchmarks presented in Table 4.1 were as high as twelve minutes. These compile times are unsuitable for general development. Profiling shows that 25% of the time is spent applying simplification rules, and 65% is spent testing for a homeomorphic embedding. We suspect both these costs can be reduced, although we have not yet tried to do so.

Simplification Time The rules from §2.1.2 are applied using the Uniplate library, in particular using the bottom-up rewrite strategy described in §3.2.5. After each function inlining, the rules are applied everywhere within the expression – despite much of the expression remaining unchanged. By targeting the application of rules more precisely, compile times would decrease.

Homeomorphic Embedding We use homeomorphic embedding to test a single element against a set of elements. The cost of homeomorphic embedding is related to the number of tests performed, and the size of the set at the time of each test. We perform many tests because of the unfolding strategy described in §4.2.2. The set is large because we maintain one set from the root function, including every inlining to the current point.

We have a solution – split the homeomorphic embedding set in two. One set can be global and used for residuation, the other set can be local and used for inlining. Each expression is optimised within the context of a fresh local set, then for residuation the global set is used. The local set will be bounded by the number of inlinings since the last residuation, while the global set will be the number of residuations from the root function. These restrictions still ensure termination, and will decrease the size of the sets substantially. This scheme would permit more inlining steps to be performed, so would change the runtime performance, but we expect the effect to be positive.

4.5 Related Work

4.5.1 Supercompilation

Supercompilation (Turchin 1986; Turchin et al. 1982) was introduced by Turchin for the Refal language (Turchin 1989). Since this original work, there have been various suggestions of both termination strategies and generalisation strategies (Turchin 1988; Sørensen and Glück 1995; Leuschel 2002). The original supercompiler maintained both positive and negative knowledge, but our implementation is a simplified version maintaining only positive information (Secher and Sørensen 2000).

The issue of let expressions in supercompilation has not previously been a primary focus. If lets are mentioned, the usual strategy is to substitute all linear lets and residuate all others. Lets have been considered in a strict setting (Jonsson and Nordlander 2007), where they are used to preserve termination semantics, but in this work all strict lets are inlined without regard to loss of sharing. Movement of lets can have a dramatic impact on performance: carefully designed let-shifting transformations give an average speedup of 15% in GHC (Peyton Jones et al. 1996), suggesting let expressions are critical to the performance of real programs.

4.5.2 Partial evaluation

There has been a lot of work on partial evaluation (Jones et al. 1993), where a program is specialised with respect to some static data. Partial evaluation works by marking all variable bindings within a program as either static or dynamic, using binding time analysis, then specialises the program with respect to the static bindings. Partial evaluation is particularly appropriate for optimising an interpreter to the expression tree of a particular program, automatically generating a compiler, and removing *interpretation overhead*. The translation of an interpreter into a compiler is known as the First Futamura Projection (Futamura 1999), and can often give an order of magnitude speedup.

Supercompilation and partial evaluation both remove overhead within a program. Partial evaluation is more suited to completely removing static data, such as an expression tree which is interpreted. Supercompilation is able to remove intermediate data structures, similar to deforestation, which partial evaluation cannot.

4.5.3 Deforestation

The deforestation technique (Wadler 1988) removes intermediate lists in computations. This technique has been extended in many ways to encompass higher order deforestation (Marlow 1996) and work on other data types (Coutts et al. 2007b). Probably the most practically motivated work has come from those attempting to restrict deforestation, in particular shortcut deforestation (Gill et al. 1993), and newer approaches such as stream fusion (Coutts et al. 2007a). In this work certain named functions are automatically fused together. By rewriting library functions in terms of these special functions, fusion occurs.

4.5.4 Whole Program Compilation

The GRIN approach (Boquist and Johnsson 1996) uses whole program compilation for Haskell. It is currently being implemented in the jhc compiler (Meacham 2008), with promising initial results. GRIN works by first removing all functional values, turning them into case expressions, allowing subsequent optimisations. The intermediate language for jhc is at a much lower level than our Core language, so jhc is able to perform detailed optimisations that we are unable to express.

4.5.5 Lower Level Optimisations

Our optimisation works at the Core level, but even once efficient Core has been generated there is still some work before efficient machine code can be produced. Key optimisations include strictness analysis and unboxing (Peyton Jones and Launchbury 1991). In GHC both of these optimisations are done at the core level, using a core language extended with unboxed types. After this lower level core has been generated, it is then compiled to STG machine instructions (Peyton Jones 1992), from which assembly code is generated. There is still work being done to modify the lowest levels to take advantage of the current generation of microprocessors (Marlow et al. 2007). We rely on GHC to perform all these optimisations after Supero generates a residual program.

4.5.6 Other Transformations

One of the central operations within our optimisation is inlining, a technique that has been used extensively within GHC (Peyton Jones and Marlow 2002). We generalise the constructor specialisation technique (Peyton Jones 2007), by allowing specialisation on any arbitrary expression, including constructors.

One optimisation we do not currently support is the use of user provided transformation rules (Peyton Jones et al. 2001), which can be used to automatically replace certain expressions with others – for example $\mathsf{sort} \circ \mathsf{nub}$ removes duplicates then sorts a list, but can be done asymptotically faster in a single operation.

Chapter 5

Defunctionalisation

This chapter details a method to reduce the number of functional values in a higher-order program, typically resulting in a first-order program. Unlike Reynolds style defunctionalisation, it does not introduce any new data types, and the results are more amenable to subsequent analysis operations. Our motivation is that the Catch analysis tool (see Chapter 6) is designed to work only upon a first-order language, but our method may have wider applicability such as termination checking (Sereni 2007).

The sections begin with an introductory example (§5.1), followed by a definition of what we consider to be a first-order program (§5.2). Next we present an overview of our method (§5.3), followed by a more detailed account (§5.4), along with a number of examples (§5.5). We classify where functional values may remain in a resultant program (§5.6) and show how to modify our method to guarantee termination (§5.7). Finally we give results (§5.8) and review related work (§5.9).

5.1 Introductory Example

Higher-order functions are widely used in functional programming languages. Having functions as first-class values leads to more concise code, but it often complicates analysis methods.

Example 26

Consider this definition of incList:

```
incList :: [Int] → [Int]
incList = map (+1)

map :: (α → β) → [α] → [β]
map f []       = []
map f (x : xs) = f x : map f xs
```

The definition of incList has higher-order features. The function (+1) is passed as a functional argument to map. The incList definition contains a partial application of map. The use of first-class functions has led to short code, but we could equally have written:

```
incList :: [Int] → [Int]
incList []       = []
incList (x : xs) = x + 1 : incList xs
```

Although this first-order variant of incList is longer (excluding the library function map), it is also more amenable to certain types of analysis. The method presented in this chapter transforms the higher-order definition into the first-order one automatically. □

Our defunctionalisation method processes the whole program to remove functional values, without changing the semantics of the program. This idea is not new. As far back as 1972 Reynolds gave a solution, now known as *Reynolds style defunctionalisation* (Reynolds 1972). Unfortunately, this method effectively introduces a mini-interpreter, which causes problems for analysis tools. Our method produces a program closer to what a human might have written, if denied the use of functional values.

5.1.1 Contributions

This chapter makes the following contributions:

- We define a defunctionalisation method which, unlike some previous work, does not introduce new data types.
- Our method can deal with the complexities of a language like Haskell, including type classes, continuations and monads.

- Our method makes use of standard transformation steps, but combined in a novel way.
- We identify restrictions which guarantee termination, but are not overly limiting.
- We have implemented our method, and present measured results for much of the nofib benchmark suite.

There are a number of limitations to our approach, most importantly:

- Our algorithm is not complete – it *does not* always succeed in removing all functional values. However, in practice, it is remarkably successful.
- The transformation can reduce sharing, causing the resulting program to be less efficient and duplicate an arbitrary amount of work. For certain types of analysis the duplication of work is not a problem, for other uses it is a severe problem.

5.2 First-Order Programs

Informally, if a program creates functional values at runtime it is higher-order, otherwise it is first-order. Functional values can only be created in two ways: (1) a lambda expression; or (2) a partially-applied function application. We therefore make the following definition:

A program which contains no lambda expressions and no partially-applied top-level functions is first-order.

Example 26 (revisited)

The original definition of `incList` is higher-order because of the partial applications of both `map` and $(+)$. The original definition of `map` is first-order. In the defunctionalised version, the program is first-order. □

We may expect the `map` definition to be higher-order, as `map` has the `f x` subexpression, where `f` is a variable, and therefore an instance of general application. We do not consider instances of general application to be higher-order, but expect that usually they will be accompanied by the creation of a functional value elsewhere within the program.

5.3 Our First-Order Reduction Method

Our method works by combining three separate and well-known transformations. Each transformation on its own is correct, and none introduces any additional data types. Our method also applies simplification rules before each transformation, most of which may be found in any optimising compiler (Peyton Jones and Santos 1994).

Arity Raising: A function can be arity raised if the body of the function is a lambda expression. In this situation, the variables bound by the lambda can be added instead as arguments of the function definition.

Inlining: Inlining is a standard technique in optimising compilers (Peyton Jones and Marlow 2002), and has been studied in depth.

Specialisation: Specialisation is another standard technique, used to remove type classes (Jones 1994) and more recently to specialise functions to a given constructor (Peyton Jones 2007).

Each transformation has the possibility of removing some functional values, but the key contribution of this chapter is *how they can be used together*. Using the fixed point operator (‡) introduced in §5.4, their combination is:

```
firstify = simplify ‡ arity ‡ inline ‡ specialise
```

We proceed by first giving a brief flavour of how these transformations may be used in isolation to remove functional values. We then discuss the transformations in detail in §5.4, including how they can be combined.

5.3.1 Simplification

Simplification serves to group several simple transformations that most optimising compilers apply. Some of these steps have the ability to remove functional values; others simply ensure a normal form for future transformations.

Example 27

```
one = (λx → x) 1
```

The simplification rule (lam-app) from §2.1.2 transforms this function to:

```
one = let x = 1 in x
```

□

Other rules do not eliminate lambda expressions, but put them into a form that other stages can remove.

Example 28

```
even = let one = 1
       in  λx → not (odd x)
```

The simplification rule (let-lam) from §5.4.1 lifts the lambda outside of the let expression.

```
even = λx → let one = 1
            in  not (odd x)
```

In general this transformation may cause duplicate computation to be performed, an issue we return to in §5.4.1. □

5.3.2 Arity Raising

The arity raising transformation increases the definition arity of functions with lambdas as bodies.

Example 29

```
even = λx → not (odd x)
```

Here the arity raising transformation lifts the argument to the lambda into a definition-level argument, increasing the arity.

```
even x = not (odd x)
```

□

5.3.3 Inlining

We use inlining to remove functions which return data constructors containing functional values. A frequent source of data constructors containing functional values is the dictionary implementation of type classes (Wadler and Blott 1989).

Example 30

```
main = case eqInt of
            (a, b) → a 1 2

eqInt = (primEqInt, primNeqInt)
```

Both components of the eqInt tuple, primEqInt and primNeqInt, are functional values. We can start to remove these functional values by inlining eqInt:

```
main = case (primEqInt, primNeqInt) of
            (a, b) → a 1 2
```

The simplification stage can now turn the program into a first-order variant, using rule (case-con) from §2.1.2.

```
main = primEqInt 1 2
```

□

5.3.4 Specialisation

Specialisation works by replacing a function application with a specialised variant. In effect, at least one argument is passed at transformation time.

Example 31

```
notList xs = map not xs
```

Here the map function takes the functional value not as its first argument. We can create a variant of map specialised to this argument:

```
map_not x = case x of
                 []     → []
                 y : ys → not y : map_not ys

notList xs = map_not xs
```

The recursive call in map is replaced by a recursive call to the specialised variant. We have eliminated all functional values. □

5.3.5 Goals

We define a number of goals: some are *essential*, and others are *desirable*. If essential goals make desirable goals unachievable in full, we still aim to do the best we can.

Essential

Preserve the result computed by the program. By making use of three established transformations, total correctness is relatively easy to show.

Ensure the transformation terminates. The issue of termination is much harder. Both inlining and specialisation could be applied in ways that diverge. In §5.7 we develop a set of criteria to ensure termination.

Recover the original program. Our transformation is designed to be performed before analysis. It is important that the results of the analysis can be presented in terms of the original program. We need a method for transforming expressions in the resultant program into equivalent expressions in the original program.

Introduce no data types. Reynolds method introduces a new data type that serves as a representation of functions, then embeds an interpreter for this data type into the program. We aim to eliminate the higher-order aspects of a program *without* introducing any new data types. By not introducing any data types we avoid introducing an interpreter, which is often a bottleneck for subsequent analysis. By composing our transformation out of existing transformations, none of which introduces data types, we can easily ensure that our resultant transformation does not introduce data types.

Desirable

Remove all functional values. We aim to remove as many functional values as possible. In §5.6 we make precise where functional values may appear in the resultant programs. If a totally first-order program is required, Reynolds' method can always be applied after our transformation. Applying our method first will cause Reynolds' method to introduce fewer additional data types and generate a smaller interpreter.

Preserve the space/sharing behaviour of the program. In the expression **let** y = f x **in** y + y, according to the rules of lazy evaluation, f x will be evaluated at most once. It is possible to inline the let binding to give f x+f x, but this expression evaluates f x twice. This transformation is valid in Haskell due to referential transparency, and will preserve both semantics and termination, but may increase the amount of work performed. In an impure or strict language, such as ML (Milner et al. 1997), this transformation may change the semantics of the program.

Our goals are primarily for analysis of the resultant code, not to compile and execute the result. Because we are not interested in performance, we permit the loss of sharing in computations if to do so will remove functional values. However, we will avoid the loss of sharing where possible, so the program remains closer to the original.

Minimize the size of the program. Previous defunctionalisation methods have reflected a concern to avoid undue code-size increase (Chin and Darlington 1996). A smaller resultant program would be desirable, but not at the cost of clarity.

Make the transformation fast. The implementation must be sufficiently fast to permit proper evaluation. Ideally, when combined with a subsequent analysis phase, the defunctionalisation should not take an excessive proportion of the runtime.

```
infixl ‡

(‡) :: Eq α ⇒ (α → α) → (α → α) → α → α
(‡) f g = fix (g ∘ fix f)

fix :: Eq α ⇒ (α → α) → α → α
fix f x = if x ≡ x′ then x else fix f x′
  where x′ = f x
```

Figure 5.1: The (‡) fixed point operator.

5.4 Method in Detail

Our method proceeds in four iteratively nested steps, simplification (simplify), arity raising (arity), inlining (inline) and specialisation (specialise). Our goal is to combine these steps to remove as many functional values as possible. For example, the initial incList example requires simplification, arity raising and specialisation.

We have implemented our steps in a monadic framework to deal with issues such as obtaining unique free variables and tracking termination constraints. But to simplify the presentation here, we ignore these issues – they are mostly tedious engineering concerns, and do not effect the underlying algorithm.

Our method is written as:

```
firstify = simplify ‡ arity ‡ inline ‡ specialise
```

Each stage will be described separately. The overall control of the algorithm is given by the (‡) operator, defined in Figure 5.1. The expression f‡g applies f to an input until it reaches a fixed point, then applies g. If g changes the value, then the whole process is repeated until a fixed point of both f and g is achieved. This formulation has several important properties:

Joint fixpoint If the operation completes, applying either f or g does not change the value.

```
propFix f g x = let r = (‡) f g x in (f r ≡ r) ∧ (g r ≡ r)
```

Idempotence The operation as a whole is idempotent.

```
propIdempotent f g x = let op = f ‡ g in op (op x) ≡ op x
```

Function ordering The function f reaches a fixed point before the function g is applied. If a postcondition of f implies a precondition of g, then we can guarantee g's precondition is always met.

These properties allow us to separate the individual transformations from the overall application strategy. The first two properties ensure that the method terminates only when no transformation is applicable. The function ordering allows us to overlap the application sites of two stages, but prefer one stage over another.

The (‡) operator is left associative, meaning that the code can be rewritten with explicit bracketing as:

```
firstify = ((simplify ‡ arity) ‡ inline) ‡ specialise
```

Within this chain we can guarantee that the end result will be a fixed point of every component transformation. Additionally, before each transformation is applied, those to the left will have reached fixed points.

The definition of operator (‡) in Figure 5.1 is written for clarity, not for speed. If the first argument is idempotent, then additional unnecessary work is performed. In the case of chaining operators, the left function is guaranteed to be idempotent if it is the result of (‡), so much computation is duplicated. We describe further optimisations in §5.8.6.

We describe each of the stages in the algorithm separately. In all subsequent stages, we assume that all the simplification rules have been applied.

5.4.1 Simplification

The simplification stage has the goal of moving lambda expressions upwards, and introducing lambdas for partially applied functions. This stage makes use of standard simplification rules from §2.1.2 plus additional rules which deal specifically with lambda expressions, given in Figure 5.2. All of the simplification rules are correct individually. The rules are applied to any subexpression, as long as any rule matches. We believe that the combination of rules from §2.1.2 and Figure 5.2 are confluent.

$\mathsf{f}\ \overline{\mathsf{xs}}$ (eta)
  $\Rightarrow \lambda\mathsf{v} \rightarrow \mathsf{f}\ \overline{\mathsf{xs}}\ \mathsf{v}$
  where arity f > length $\overline{\mathsf{xs}}$

let v = (λw → x) **in** y (bind-lam)
  ⇒ y [v / λw → x]

let v = x **in** y (bind-box)
  ⇒ y [v / x]
  where x is a boxed lambda (see §5.4.3)

let v = x **in**λw → y (let-lam)
  ⇒ λw → **let** v = x **in** y

Figure 5.2: Additional Simplification rules.

Lambda Introduction

The (eta) rule inserts lambdas in preference to partial applications, using η-expansion. For each partially applied function, a lambda expression is inserted to ensure that the function is given at least as many arguments as its associated arity.

Example 32

(∘) f g x = f (g x)

even = (∘) not odd

Here the functions (∘), not and odd are all unsaturated. Lambda expressions can be inserted to saturate these applications.

even = λx → (∘) (λy → not y) (λz → odd z) x

Here the even function, which previously had three instances of partial application, has three lambda expressions inserted. Now each function is fully-applied. This transformation enables the arity raising transformation, resulting in:

even x = (∘) (λy → not y) (λz → odd z) x

□

This replaces partial application with lambda expressions, and has the advantage of making functional values more explicit, permitting arity raising.

Lambda Movement

The (bind-lam) rule inlines a lambda bound in a let expression. The (bind-box) rule will be discussed as part of the inlining stage, see §5.4.3. The (let-lam) rule can be responsible for a reduction in sharing:

Example 33

```
f x = let i = expensive x
      in  λj → i + j

main xs = map (f 1) xs
```

Here (expensive 1) is computed once and saved. Every application of the functional argument within map performs a single (+) operation. After applying the (let-lam) rule we get:

```
f x = λj → let i = expensive x
           in  i + j
```

Now expensive is recomputed for every element in xs. We include this rule in our simplifier, focusing on functional value removal at the expense of sharing. □

5.4.2 Arity Raising

The arity raising step is:

```
function $\overline{vs}$ = λv → x
   ⇒ function $\overline{vs}$ v = x
```

Given a body which is a lambda expression, the arguments to the lambda expression can be lifted into the definition-level arguments for the function. If a function has its arity increased, fully-applied uses become partially-applied, causing the (eta) simplification rule to fire.

$$\begin{aligned}
&\mathsf{isBox}\ [\![\mathsf{c}\ \overline{\mathsf{xs}}]\!] &&= \mathsf{any\ isLambda}\ \overline{\mathsf{xs}} \lor \mathsf{any\ isBox}\ \overline{\mathsf{xs}}\\
&\mathsf{isBox}\ [\![\mathbf{let}\ \mathsf{v} = \mathsf{x}\ \mathbf{in}\ \mathsf{y}]\!] &&= \mathsf{isBox\ y}\\
&\mathsf{isBox}\ [\![\mathbf{case}\ \mathsf{x}\ \mathbf{of}\ \overline{\mathsf{alts}}]\!] &&= \mathsf{any\ (isBox} \circ \mathsf{rhs)}\ \overline{\mathsf{alts}}\\
&\mathsf{isBox}\ [\![\mathsf{f}\ \overline{\mathsf{xs}}]\!] &&= \mathsf{isBox\ (body\ f)}\\
&\mathsf{isBox}\ \_ &&= \mathsf{False}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{isLambda}\ [\![\lambda\mathsf{v} \to \mathsf{x}]\!] &&= \mathsf{True}\\
&\mathsf{isLambda}\ \_ &&= \mathsf{False}
\end{aligned}$$

The isBox function as presented may not terminate, but by simply keeping a list of followed functions, we can assume the result is False in any duplicate call. This modification does not change the result of any previously terminating evaluations.

Figure 5.3: The isBox function, to test if an expression is a boxed lambda.

5.4.3 Inlining

We use inlining of top-level functions as the first stage in the removal of functional values stored within a data value – for example Just $(\lambda x \to x)$. We refer to expressions that evaluate to functional values inside data values as *boxed lambdas*. If a boxed lambda is bound in a let expression, we substitute the let binding, using the (bind-box) rule from Figure 5.2. We only inline a function if two conditions both hold: (1) the function's body is a boxed lambda; (2) the function application occurs within a case scrutinee.

An expression e is a boxed lambda if isBox e $\equiv$ True, where isBox is defined as in Figure 5.3.

Example 34

Recalling that [e] is shorthand for (:) e [ ], where (:) is the cons constructor, the following expressions are boxed lambdas:

$[\lambda x \to x]$
$(\mathsf{Just}\ [\lambda x \to x])$
$(\mathbf{let}\ y = 1\ \mathbf{in}\ [\lambda x \to x])$
$[\mathsf{Nothing}, \mathsf{Just}\ (\lambda x \to x)]$

The following are *not* boxed lambdas:

$\lambda x \to \mathsf{id}\ x$

```
[id (λx → x)]
id [λx → x]
```

The final expression *evaluates to* a boxed lambda, but this information is hidden by the id function. We rely on specialisation to remove any boxed lambdas passed to functions. □

The inlining transformation is specified by:

```
case (f $\overline{xs}$) of $\overline{alts}$
   ⇒ case (let $\overline{vs}$ = $\overline{xs}$ in y) $\overline{alts}$
  where
     $\overline{vs}$ = args f
     y = body f
     If isBox y  evaluates to True
```

As with the simplification stage, there may be some loss of sharing if the definition being inlined has arity 0 – a constant applicative form (CAF). A Haskell implementation computes these expressions only once, and reuses their value as necessary. If they are inlined, this sharing will be lost.

5.4.4 Specialisation

For each application of a top-level function in which at least on argument has a subexpression which is a lambda, a specialised variant is created, and used where applicable. The process follows the same pattern as constructor specialisation (Peyton Jones 2007), but applies where function arguments are lambda expressions, rather than known constructors. Examples of common functions whose applications can usually be made first-order by specialisation include map, filter, foldr and foldl.

The specialisation transformation makes use of *templates*. A template is an expression where some sub-expressions are omitted, denoted by an underscore. The process of specialisation proceeds as follows:

1. Find all function applications in which at least one argument contains a lambda, and generate templates, omitting first-order components (see Generating Templates).
2. For each template, generate a function specialised to that template (see Generating Functions).

3. For each subexpression matching a template, replace it with the generated function (see Using Templates).

Example 35

```
main xs = map (λx → x) xs

map f xs = case xs of
               []      → []
               y : ys → f y : map f ys
```

Specialisation first finds the application of map in main, and generates the template map (λx → x) _. It then generates a unique name for the template (we choose map_id), and generates an appropriate function body. Next all calls matching the template are replaced with calls to map_id, including the call to map within the freshly generated map_id.

```
main xs = map_id xs

map_id xs = case xs of
               []      → []
               y : ys → y : map_id ys
```

The resulting code is first-order. □

Generating Templates

A template is generated if an expression is a function application, whose arguments include a sub-expression which is either a lambda or a boxed lambda – as calculated by the shouldTemplate function in Figure 5.4. Before generating a template, the abstractTemplate function is applied to make the template more widely applicable, while ensuring that the revised template requires no additional functional arguments. For example, given the expression f (λv → v) True, shouldTemplate would return True as one of the arguments is a lambda expression. The abstractTemplate function would abstract True, and generate the template f (λv → v) _. If the expression f (λv → x) False was encountered, the abstraction would ensure that no new template was required.

```
shouldTemplate :: Expr → Bool
shouldTemplate ⟦f $\overline{xs}$⟧ = any (λx → isLambda x ∨ isBox x) (universe ⟦f $\overline{xs}$⟧)
shouldTemplate _ = False

abstractTemplate :: Expr → Expr
abstractTemplate x =
  transform (λx → if abstract x then _ else x) $
  abstractVars (freeVars x) x

abstract :: Expr → Bool
abstract ⟦c $\overline{xs}$⟧ = all abstract $\overline{xs}$
abstract ⟦f $\overline{xs}$⟧ = all abstract $\overline{xs}$
abstract ⟦x $\overline{xs}$⟧ = all abstract $\overline{xs}$
abstract ⟦let v = x in y⟧ = abstract x ∧ abstract (abstractVars [v] y)
abstract ⟦case x of $\overline{alts}$⟧ = abstract x ∧ all alt $\overline{alts}$
  where alt ⟦c $\overline{vs}$ → x⟧ = abstract (abstractVars $\overline{vs}$ x)
abstract x = (x ≡ _)

abstractVars :: [VarName] → Expr → Expr
abstractVars vs x = ...
    -- replace the variables vs in x with _
    -- respecting variables rebound locally
```

Figure 5.4: Template generation function.

Example 36

Expression	Template after abstraction
id (λx → x)	id (λx → x)
id (Just (λx → x))	id (Just (λx → x))
id (λx → **let** y = 12 **in** 4)	id (λx → _)
id (λx → **let** y = 12 **in** x)	id (λx → **let** y = _ **in** x)

In all these examples, the id function has an argument which has a lambda expression as a subexpression. In the final two cases, there are subexpressions which do not depend on variables bound within the lambda – these have been removed and replaced with underscores. The Just constructor is also not dependent on the bound variables, but its removal would require a functional argument as a parameter, so it is left as part of the template. □

Generating Functions

Given a template, to generate an associated function, a unique function name is allocated to the template. For each occurrence of _ in a template a fresh argument variable is assigned. The body is produced by unfolding the outer function symbol in the template once.

Example 35 (revisited)

Consider the template map (λx → x) _. Let v_1 be the fresh argument variable for the single _ placeholder, and map_id be the function name:

```
map_id v₁ = map (λx → x) v₁
```

We unfold the definition of map once:

```
map_id v₁ = let f  = λx → x
                xs = v₁
            in  case xs of
                    []     → []
                    y : ys → f y : map f ys
```

After the simplification rules from Figure 5.2, we obtain:

```
map_id v₁ = let xs = v₁
            in  case xs of
                    []     → []
                    y : ys → y : map (λx → x) ys
```

□

Using Templates

After a function has been generated for each template, every expression matching a template can be replaced by a call to the new function. Every subexpression corresponding to an _ is passed as an argument.

Example 35 (continued)

```
map_id v₁ = let xs = v₁
            in  case xs of
                    []     → []
                    y : ys → y : map_id ys
```

We now have a first-order definition. □

5.4.5 Primitive Functions

Primitive functions do not have an associated body, and therefore cannot be examined or inlined. We make just two simple changes to support primitives.

1. We define that a primitive application is *not* a boxed lambda.
2. We restrict specialisation so that if a function to be specialised is actually a primitive, no template is generated. The reason for this restriction is that the generation of code associated with a template requires a one-step unfolding of the function, something which cannot be done for a primitive.

Example 37

```
main = (λx → x) `seq` 42
```

Here a functional value is passed as the first argument to the primitive seq. As we are not able to peer inside the primitive, and must preserve its interface, we cannot remove this functional value. For most primitives, such as arithmetic operations, the types ensure that no functional values are passed as arguments. However, the seq primitive is of type $\alpha \rightarrow \beta \rightarrow \beta$, allowing any type to be passed as either of the arguments, including functional values.

Some primitives not only *permit* functional values, but actually *require* them. For example, the primCatch function within the Yhc standard libraries implements the Haskell exception handling function catch. The type of primCatch is $\alpha \rightarrow (\mathsf{IOError} \rightarrow \alpha) \rightarrow \alpha$, taking an exception handler as one of the arguments. □

5.4.6 Recovering Input Expressions

Specialisation is the only stage which introduces new function names. In order to translate an expression in the result program to an equivalent expression in the input program, it is sufficient to replace all generated function names with their associated template, supplying all the necessary variables.

5.5 Examples

We now give two examples. Our method can convert the first example to a first-order equivalent, but not the second.

Example 38 (Inlining Boxed Lambdas)

An earlier version of our defunctionaliser inlined boxed lambdas everywhere they occurred. Inlining boxed lambdas means the isBox function does not have to examine the body of applied functions, and is therefore simpler. However, it was unable to cope with programs like this one:

```
main = map ($1) gen
gen = (λx → x) : gen
```

The gen function is both a boxed lambda and recursive. If we inlined gen initially the method would not be able to remove all lambda expressions.

By first specialising map with respect to gen, and waiting until gen is the subject of a case, we are able to remove the functional values. This operation is effectively deforestation (Wadler 1988), which also only performs inlining within the subject of a case. □

Example 39 (Functional Lists)

Sometimes lambda expressions are used to build up lists which can have elements concatenated onto the end. Using Hughes lists (Hughes 1986), we can define:

```
nil = id
snoc x xs = λys → xs (x : ys)
list xs = xs [ ]
```

This list representation provides nil as the empty list, but instead of providing a (:) or "cons" operation, it provides snoc which adds a single element on to the end of the list. The function list is provided to create a standard list. We are unable to defunctionalise such a construction, as it stores unbounded information within closures. We have seen such constructions in both the lines function of the HsColour program, and the sort function of Yhc. However, there is an alternative implementation of these functions:

```
nil = [ ]
snoc = (:)
list = reverse
```

We have benchmarked these operations in a variety of settings and the list based version appears to use approximately 75% of the memory, and 65% of the time required by the function-based solution. We suggest that people using continuations for snoc-lists move instead to a list type! □

5.6 Restricted Completeness

Our method would be *complete* if it removed all lambda expressions and partially-applied functions from a program. All partially-applied functions are translated to lambda expressions using the (eta) rule. We therefore need to determine where a lambda expression may occur in a program after the application of our defunctionalisation method.

5.6.1 Notation

To examine where lambda expressions may occur, we model expressions in our Core language as a set of *syntax trees*. We define the following rules, which generate sets of expressions:

$$\begin{array}{ll}
\mathsf{lam}\ \mathsf{x} & = \{\lambda \mathsf{v}' \to \mathsf{x}' \mid \mathsf{v}' \in \mathsf{v}, \mathsf{x}' \in \mathsf{x}\} \\
\mathsf{fun}\ \mathsf{x}\ \mathsf{y} & = \{\mathsf{f}'\ \overline{\mathsf{ys}}' \mid \mathsf{f}' \in \mathsf{f}, \mathsf{x}' \in \mathsf{x}, \overline{\mathsf{ys}}'\overline{\in}\mathsf{y}, \mathsf{body}\ \mathsf{f}' \equiv \mathsf{x}'\} \\
\mathsf{con}\ \mathsf{x} & = \{\mathsf{c}'\ \overline{\mathsf{xs}}' \mid \overline{\mathsf{xs}}'\overline{\in}\mathsf{x}\} \\
\mathsf{app}\ \mathsf{x}\ \mathsf{y} & = \{\mathsf{x}'\ \overline{\mathsf{ys}}' \mid \mathsf{x}' \in \mathsf{x}, \overline{\mathsf{ys}}'\overline{\in}\mathsf{y}\} \\
\mathsf{var} & = \{\mathsf{v}' \mid \mathsf{v}' \in \mathsf{v}\} \\
\mathsf{let}\ \mathsf{x}\ \mathsf{y} & = \{\mathbf{let}\ \mathsf{v}' = \mathsf{x}'\ \mathbf{in}\ \mathsf{y}' \mid \mathsf{x}' \in \mathsf{x}, \mathsf{y}' \in \mathsf{y}\} \\
\mathsf{case}\ \mathsf{x}\ \mathsf{y} & = \{\mathbf{case}\ \mathsf{x}'\ \mathbf{of}\ \overline{\mathsf{alts}}' \mid \mathsf{x}' \in \mathsf{x}, \\
& \qquad\qquad \overline{\mathsf{alts}}'\overline{\in}\{\mathsf{c}'\ \overline{\mathsf{vs}}' \to \mathsf{y}' \mid \mathsf{c}' \in \mathsf{c}, \overline{\mathsf{vs}}'\overline{\in}\mathsf{v}, \mathsf{y}' \in \mathsf{y}\}\}
\end{array}$$

Here v is the set of all variables, f the set of function names, and c the set of constructors. We use $\overline{\mathsf{xs}}\overline{\in}\mathsf{e}$ to denote that $\overline{\mathsf{xs}}$ is a sequence of any length, whose elements are drawn from e. In the definition of $\mathsf{fun}\ \mathsf{x}\ \mathsf{y}$, the expression set x represents the possible bodies of the function, while y represents the arguments. We can now define an upper bound on the set of unrestricted expressions in our Core language as the smallest solution to the equation s_0:

$$\begin{array}{ll}
\mathsf{s}_0 = & \mathsf{lam}\ \mathsf{s}_0 \cup \mathsf{fun}\ \mathsf{s}_0\ \mathsf{s}_0 \cup \mathsf{con}\ \mathsf{s}_0 \cup \mathsf{app}\ \mathsf{s}_0\ \mathsf{s}_0 \cup \mathsf{var} \cup \\
& \mathsf{case}\ \mathsf{s}_0\ \mathsf{s}_0 \cup \mathsf{let}\ \mathsf{s}_0\ \mathsf{s}_0
\end{array}$$

5.6.2 A Proposition about Residual Lambdas

We classify the location of lambdas within the residual program, assuming the following two conditions are satisfied:

1. The termination criteria do not curtail defunctionalisation (see §5.7).

2. No primitive function receives a functional argument, or returns a functional result.

Given these assumptions, a lambda or boxed lambda may only occur in the following places: (1) the body of the main function; (2) passed as an argument to a variable of functional type; (3) the body of a lambda expression. In §5.6.4 we give examples of these residual forms.

5.6.3 Proof of the Proposition

First we show that residual definition bodies belong to a proper subset of s_0, by defining successively smaller subsets, where $\mathsf{s}_\mathsf{n} \supset \mathsf{s}_{\mathsf{n}+1}$. We use the joint fixpoint property of the (‡) operator to calculate the restrictions imposed by each stage of defunctionalisation. Secondly we describe which expressions may be the *parents* of residual lambda expressions, using our refined set of possible expressions.

Restriction 1: Type Safety We know our original program is type safe. Each of our stages preserves semantics, and therefore type safety. So the scrutinee of a case cannot be a functional value. Also, all constructor expressions are saturated, so they must evaluate to a data value, and cannot be applied to arguments. Refining our bounding set to take account of these observations, we have:

$$\begin{aligned}\mathsf{s}_1 = {} & \mathsf{lam}\ \mathsf{s}_1 \cup \mathsf{fun}\ \mathsf{s}_1\ \mathsf{s}_1 \cup \mathsf{con}\ \mathsf{s}_1 \cup \mathsf{app}\ (\mathsf{s}_1 - \mathsf{con}\ \mathsf{s}_1)\ \mathsf{s}_1 \cup \mathsf{var} \cup \\ & \mathsf{case}\ (\mathsf{s}_1 - \mathsf{lam}\ \mathsf{s}_1)\ \mathsf{s}_1 \cup \mathsf{let}\ \mathsf{s}_1\ \mathsf{s}_1\end{aligned}$$

Restriction 2: Standard Simplification Rules Our simplification rules from §2.1.2 are applied until a fixed point is found, meaning that no expression matching the left-hand side of a rule can occur in the output. For example, the left-hand side of the (case-con) rule is $\mathsf{case}\ (\mathsf{con}\ \mathsf{s})\ \mathsf{s}$, so this pattern cannot remain in a residual program. By similarly examining left-hand sides of all the standard simplification rules we can further reduce the bounding set of residual expressions:

$$\begin{aligned}\mathsf{s}_2 = {} & \mathsf{lam}\ \mathsf{s}_2 \cup \mathsf{fun}\ \mathsf{s}_2\ \mathsf{s}_2 \cup \mathsf{con}\ \mathsf{s}_2 \cup \mathsf{app}\ \mathsf{var}\ \mathsf{s}_2 \cup \mathsf{var} \cup \\ & \mathsf{case}\ (\mathsf{fun}\ \mathsf{s}_2\ \mathsf{s}_2 \cup \mathsf{app}\ \mathsf{var}\ \mathsf{s}_2 \cup \mathsf{var})\ \mathsf{s}_2 \cup \mathsf{let}\ \mathsf{s}_2\ \mathsf{s}_2\end{aligned}$$

Restriction 3: Lambda Simplification Rules We apply the lambda rules from Figure 5.2. As $(\mathsf{e} - \mathsf{lam}\ \mathsf{e})$ occurs repeatedly we have factored it out as l'. To allow reuse of l' in future definitions, we parameterise by n to obtain l'_n.

$$\begin{aligned}\mathsf{s}_3 &= \mathsf{e}_3 \\ \mathsf{e}_3 &= \mathsf{lam}\ \mathsf{e}_3 \cup \mathsf{fun}\ \mathsf{e}_3\ \mathsf{e}_3 \cup \mathsf{con}\ \mathsf{e}_3 \cup \mathsf{app}\ \mathsf{var}\ \mathsf{e}_3 \cup \mathsf{var} \cup \\ &\quad\ \mathsf{case}\ (\mathsf{fun}\ \mathsf{e}_3\ \mathsf{e}_3 \cup \mathsf{app}\ \mathsf{var}\ \mathsf{e}_3 \cup \mathsf{var})\ \mathsf{l}'_3 \cup \mathsf{let}\ \mathsf{l}'_3\ \mathsf{l}'_3 \\ \mathsf{l}'_\mathsf{n} &= \mathsf{e}_\mathsf{n} - \mathsf{lam}\ \mathsf{e}_\mathsf{n}\end{aligned}$$

Restriction 4: Arity Raising Arity raising guarantees that no function body is a lambda expression.

$$\begin{aligned}
\mathsf{s_4} &= \mathsf{l'_4} \\
\mathsf{e_4} &= \mathsf{lam\ e_4} \cup \mathsf{fun\ l'_4\ e_4} \cup \mathsf{con\ e_4} \cup \mathsf{app\ var\ e_4} \cup \mathsf{var} \cup \\
&\quad \mathsf{case\ (fun\ l'_4\ e_4} \cup \mathsf{app\ var\ e_4} \cup \mathsf{var)\ l'_4} \cup \mathsf{let\ l'_4\ l'_4}
\end{aligned}$$

Restriction 5: Inlining and (bind-box) To deal with inlining, we need to work with lambda boxes, as defined by the function isBox, from Figure 5.3. We define $\mathsf{b'_n}$ to be the expressions with children drawn from $\mathsf{e_n}$ which are *not* lambda boxes:

$$\begin{aligned}
\mathsf{b'_n} &= \mathsf{lam\ e_n} \cup \mathsf{fun\ b'_n\ e_n} \cup \mathsf{con\ (b'_n} - \mathsf{lam\ e_n)} \cup \mathsf{app\ e_n\ e_n} \cup \mathsf{var} \cup \\
&\quad \mathsf{case\ e_n\ b'_n} \cup \mathsf{let\ e_n\ b'_n}
\end{aligned}$$

As an example of how the component subsets of $\mathsf{b'_n}$ are obtained, take $\mathsf{fun\ b'_n\ e_n}$. A function application is a lambda box if the function's body is a lambda box. Therefore, provided the body of the function is not a lambda box, the function application will not be. The arguments to a function application do not affect whether the application is a lambda box, and are left unrestricted as $\mathsf{e_n}$.

The inlining stage and the (bind-box) simplification rule match expressions which are lambda boxes, therefore these expressions can be eliminated from the residual program:

$$\begin{aligned}
\mathsf{s_5} &= \mathsf{l'_5} \\
\mathsf{e_5} &= \mathsf{lam\ e_5} \cup \mathsf{fun\ l'_5\ e_5} \cup \mathsf{con\ e_5} \cup \mathsf{app\ var\ e_5} \cup \mathsf{var} \cup \\
&\quad \mathsf{case\ (fun\ (l'_5} \cap \mathsf{b'_5)\ e_5} \cup \mathsf{app\ var\ e_5} \cup \mathsf{var)\ l'_5} \cup \\
&\quad \mathsf{let\ (l'_5} \cap \mathsf{b'_5)\ l'_5}
\end{aligned}$$

Restriction 6: Specialisation As specialisation removes all lambdas and boxed lambdas from the arguments of function applications we define:

$$\begin{aligned}
\mathsf{s_6} &= \mathsf{l'_6} \\
\mathsf{e_6} &= \mathsf{lam\ e_6} \cup \mathsf{fun\ l'_6\ (l'_6} \cap \mathsf{b'_6)} \cup \mathsf{con\ e_6} \cup \mathsf{app\ var\ e_6} \cup \mathsf{var} \cup \\
&\quad \mathsf{case\ (fun\ (l'_6} \cap \mathsf{b'_6)\ (l'_6} \cap \mathsf{b'_6)} \cup \mathsf{app\ var\ e_6} \cup \mathsf{var)\ l'_6} \cup \\
&\quad \mathsf{let\ (l'_6} \cap \mathsf{b'_6)\ l'_6}
\end{aligned}$$

Residual Forms Having applied all the rules, we now classify what the parent expressions of a lambda may be. Since l'_n by definition excludes lambda expressions, no residual function body can be a lambda. We can define lp, the lambda parents, consisting of the expressions drawn from e_6 which permit a lambda expression as a direct child. We have denoted the possible presence of a lambda with l, and their absence with an underscore:

$\mathsf{lp} = \mathsf{lam\ l} \cup \mathsf{con\ l} \cup \mathsf{app\ \_\ l}$

That is, a lambda may occur as the child of a lambda expression, as an argument to a constructor, or as an argument to an application. However, a constructor containing a lambda is a boxed lambda, and therefore is not permitted anywhere b' is intersected with the expression. Similarly to lp, we can define bp, the boxed parents, consisting of expressions drawn from e_6 which permit a boxed lambda as a direct child:

$\mathsf{bp} = \mathsf{lam\ b} \cup \mathsf{fun\ b\ \_} \cup \mathsf{con\ b} \cup \mathsf{app\ \_\ b} \cup \mathsf{case\ \_\ b} \cup \mathsf{let\ \_\ b}$

Either the body of the main function is a boxed lambda, or a boxed lambda must have a parent expression which is not a boxed lambda. We can restate bp to exclude expressions which are themselves boxed lambdas, and determine the ultimate parent of a boxed lambda:

$\mathsf{bp} = \mathsf{lam\ b} \cup \mathsf{app\ \_\ b}$

The $\mathsf{con\ l}$ expression is itself a boxed lambda, and is only permitted where bp permits. Therefore, the ultimate parent of either a lambda or a boxed lambda can be expressed as:

$\mathsf{p} = \mathsf{lam}\ (\mathsf{b} \cup \mathsf{l}) \cup \mathsf{app\ \_}\ (\mathsf{b} \cup \mathsf{l})$

In view of the restrictions imposed on e_6, we also know that the first argument of any general application must be a variable. So we have shown, as required, that a lambda or boxed lambda may only occur as the body of a lambda, passed as an argument to a variable in a general application, or as the body of the main function.

5.6.4 Example Residual Lambdas

The most interesting residual lambdas occur as arguments in an application of a variable – for example $\mathsf{v}\ (\lambda \mathsf{x} \to \mathsf{x})$. In this example, the lambda $(\lambda \mathsf{x} \to \mathsf{x})$

```
[x,y,z]
app(lam(x),y)     -> let(y,x)
app(case(x,y),z)  -> case(x,app(y,z))
app(let(x,y),z)   -> let(x,app(y,z))
case(let(x,y),z)  -> let(x,case(y,z))
case(con(x),y)    -> let(x,y)
case(x,lam(y))    -> lam(case(x,app(lam(y),var)))
let(lam(x),y)     -> lam(let(x,y))
```

Figure 5.5: Encoding of termination simplification.

cannot be bound to the variable v. This leaves three possibilities: (1) either v is bound to $\perp$; or (2) v is never bound to anything; or (3) v is bound outside the program. For example:

```
bottom = bottom
main₁ = bottom (λx → x)

nothing = Nothing
main₂ = case nothing of
            Nothing → 1
            Just f   → f (λx → x)

main₃ f = f (λx → x)
```

The residual lambda in main_1 is a result of the non-termination of the bottom function, and the lambda in main_2 is part of dead code. In both cases the lambda expression is never evaluated and no functional value is created at runtime. The final main_3 example could be eliminated by requiring a first-order main function.

5.7 Proof of Termination

Our algorithm, as it stands, may not terminate. In order to ensure termination, it is necessary to bound both the inlining and specialisation stages. In this section we develop a mechanism to ensure termination, by first looking at how non-termination may arise.

5.7.1 Termination of Simplification

In order to check the termination of the simplifier we have used the AProVE system (Giesl et al. 2006a) to model our rules as a *term rewriting system*, and check its termination. An encoding of a simplified version of the rules from Figures 2.4 and 5.2 is given in Figure 5.5. We have encoded rules by considering what type of expression is transformed by a rule. For example, the rule replacing $(\lambda \mathsf{v} \rightarrow \mathsf{x})\ \mathsf{y}$ with **let** $\mathsf{v} = \mathsf{y}$ **in** x is expressed as a rewrite replacing $\mathsf{app}\ (\mathsf{lam}\ (\mathsf{x}), \mathsf{y})$ with **let** (y, x). The names of binding variables with expressions have been ignored. To simplify the encoding, we have only considered applications with one argument. The rules are applied non-deterministically at any suitable location, so faithfully model the behaviour of the original rules.

The encoding of the (bind-box) and (bind-lam) rules is excluded. Given these rules, there are non terminating sequences. For example:

$(\lambda \mathsf{x} \rightarrow \mathsf{x}\ \mathsf{x})\ (\lambda \mathsf{x} \rightarrow \mathsf{x}\ \mathsf{x})$
    $\Rightarrow$ -- (lam-app) rule
let $\mathsf{x} = \lambda \mathsf{x} \rightarrow \mathsf{x}\ \mathsf{x}$ **in** $\mathsf{x}\ \mathsf{x}$
    $\Rightarrow$ -- (bind-lam) rule
$(\lambda \mathsf{x} \rightarrow \mathsf{x}\ \mathsf{x})\ (\lambda \mathsf{x} \rightarrow \mathsf{x}\ \mathsf{x})$

Such expressions are a problem for GHC, and can cause the compiler to non-terminate if encoded as data structures (Peyton Jones and Marlow 2002). Other transformation systems (Chin and Darlington 1996) are able to make use of type annotations to ensure these reductions terminate. To guarantee termination, we apply (bind-lam) or (bind-box) at most n times in any definition body. If the body is altered by either inlining or specialisation, we reset the count. Currently we have set n to 1000, and have never had this limit reached. This limited is intended to give a strong guarantee of termination, and will only be necessary rarely – hence the high bound.

5.7.2 Termination of Arity Raising

Functions may only ever increase in arity, and in a well-typed program, provided the function bodies do not grow without bound, the increase in arity may only occur a finite number of times. The untyped program $\mathsf{f}\ \mathsf{x} = \mathsf{f}$ causes arity-raising to non-terminate, and can be mitigated by a bound in a similar manner to §5.7.1.

5.7.3 Termination of Inlining

A standard technique to ensure termination of inlining is to refuse to inline recursive functions (Peyton Jones and Marlow 2002). For our purposes, this non-recursive restriction is too cautious as it would leave residual lambda expressions in cases such as Example 38. We first present a program which causes our method to fail to terminate, then our means of ensuring termination.

Example 40

```
data B x = B x
f = case f of
          B _ → B (λx → x)
```

The f inside the case is a candidate for inlining:

```
case f of B _ → B (λx → x)
   ⇒   -- inlining rule
case (case f of B _ → B (λx → x)) of B _ → B (λx → x)
   ⇒   -- (case-case) rule
case f of B _ → case B (λx → x) of B _ → B (λx → x)
   ⇒   -- (case-con) rule
case f of B _ → B (λx → x)
```

So this expression would cause non-termination. □

To avoid such problems, we permit inlining a function f, at all use sites within the definition of a function g, but only once per pair (f, g). In the previous example we would inline f within its own body, but only once. Any future attempts to inline f within this function would be disallowed, although f could still be inlined within other function bodies. This restriction is sufficient to ensure termination of inlining. Given n functions, there can only be n^2 possible inlining steps, each for possibly many application sites.

5.7.4 Termination of Specialisation

The specialisation method, left unrestricted, also may not terminate.

Example 41

```
data Wrap a = Wrap (Wrap a)
            | Value a

f x = f (Wrap x)
main = f (Value head)
```

In the first iteration, the specialiser generates a version of f specialised for the argument Value head. In the second iteration it would specialise for Wrap (Value head), then in the third with Wrap (Wrap (Value head)). Specialisation would generate an infinite number of specialisations of f. □

To ensure we only specialise a finite number of times we use a homeomorphic embedding, from §2.4. We associate a set S with each function. After specialising with a template we add that template to the set S of the function associated with that expression. When we create a new function based on a template, we copy the S associated with the function in which the specialisation is performed. If a function wants to specialise using a template that is a homeomorphic embedding of the S associated with that function, the specialisation is not permformed.

One of the conditions for termination of homeomorphic embedding is that there is only a finite alphabet. During the process of specialisation we create new functions, and these new functions are new symbols in our language. So we only use function names from the original input program. Every template has a correspondence with an expression in the original program. We perform the homeomorphic embedding test only after transforming all templates into their original equivalent expression.

Example 41 (revisited)

Using homeomorphic embedding, we again generate the specialised variant of f (Value head). Next we generate the template f (Wrap (Value head)). However, f (Value head) $\trianglelefteq$ f (Wrap (Value head)), so the new template would not be used. □

Forbidding homeomorphic embeddings in specialisation still allows full defunctionalisation in most simple examples, but there are examples where it terminates prematurely.

Example 42

$\mathsf{main\ y = f\ (\lambda x \rightarrow x)\ y}$
$\mathsf{f\ x\ y = fst\ (x, f\ x\ y)\ y}$

Here we first generate a specialised variant of $\mathsf{f\ (\lambda x \rightarrow x)\ y}$. If we call the specialised variant $\mathsf{f'}$, we have:

$\mathsf{f'\ y = fst\ (\lambda x \rightarrow x, f'\ y)\ y}$

Note that the recursive call to f has also been specialised. We now attempt to generate a specialised variant of fst, using the template $\mathsf{fst\ (\lambda x \rightarrow x, f'\ y)\ y}$. Unfortunately, this template is an embedding of the template we used for $\mathsf{f'}$, so we do not specialise and the program remains higher-order. But if we did permit a further specialisation, we would obtain the first-order equivalent:

$\mathsf{f'\ y = fst'\ y\ y}$
$\mathsf{fst'\ y_1\ y_2 = y_2}$

□

This example may look slightly obscure, but similar situations occur commonly with the standard translation of dictionaries. Often, classes have default methods, which call other methods in the same class. These recursive class calls often pass dictionaries, embedding the original caller even though no recursion actually happens.

To alleviate this problem, instead of storing one set S, we store a sequence of sets, $S_1 \ldots S_n$ – where n is a small positive number, constant for the duration of the program. Instead of adding to the set S, we now add to the lowest set S_i where adding the element will not violate the admissible sequence. Each of the sets S_i is still finite, and there are a finite number (n) of them, so termination is maintained.

By default our defunctionalisation program uses 8 sets. In the results table given in §5.8, we have given the minimum possible value of n to remove all lambda expressions within each program.

5.7.5 Termination as a Whole

Given an initial program, the arity raising, inlining and specialisation stages will each apply a finite number of times. The simplification stage is termi-

Name	**Bound**	**HO Create**		**HO Use**		**Time**	**Size**
Programs curtailed by a termination bound:							
cacheprof	8	611	44	686	40	1.8	2%
grep	8	129	9	108	22	0.8	40%
lift	8	187	123	175	125	1.2	-6%
prolog	8	308	301	203	137	1.1	-5%
All other programs:							
ansi	4	239	0	187	2	0.5	-29%
bernouilli	4	240	0	190	2	0.3	-32%
bspt	4	262	0	264	1	0.7	-22%
. . . plus 56 additional programs . . .							
sphere	4	343	0	366	2	0.7	-45%
symalg	5	402	0	453	64	1.0	-32%
x2n1	4	345	0	385	2	0.8	-57%
Summary of all other programs:							
Minimum	2	60	0	46	0	0.1	-78%
Maximum	14	580	1	581	100	1.2	27%
Average	5	260	0	232	5	0.5	-30%

Name is the name of the program; **Bound** is the numeric bound used for termination (see §5.7.4); **HO Create** the static number of lambda expressions and under-applied functions, first in the input program and then in the output program; **HO Use** the number of application expressions and over-applied functions; **Time** the execution time of our method in seconds; **Size** the change in the program size measured as the number of lines of Core.

Table 5.1: Results of defunctionalisation on the nofib suite.

nating on its own, and will be invoked a finite number of times, so will also terminate. Therefore, when combined, the stages will terminate.

5.8 Results

5.8.1 Benchmark Tests

We have tested our method with programs drawn from the nofib benchmark suite (Partain et al. 2008), and the results are given in Table 5.1. Looking at the input Core programs, we see many sources of functional values.

- Type classes create dictionaries which are implemented as tuples of functions.
- The monadic bind operation is higher-order.
- The IO data type is implemented as a function.
- The Haskell Show type class uses continuation-passing style extensively.
- List comprehensions in Yhc are desugared to continuations. There are other translations which require less functional value manipulations (Coutts et al. 2007a).

We have tested all 14 programs from the imaginary section of the nofib suite, 35 of the 47 spectral programs, and 17 of the 30 real programs. The remaining 25 programs do not compile using the Yhc compiler, mainly due to missing or incomplete libraries. Upon applying our defunctionalisation method, 4 programs are curtailed by the termination bound, and 5 additional programs remain higher-order. We first discuss the residual higher-order programs, then make some observations about each of the columns in the table.

5.8.2 Higher-Order Residues

All four programs curtailed by the termination bound are listed in Table 5.1. The lift program uses pretty-printing combinators, while the other three programs use parser combinators. In all programs, the combinators are used to build up a functional value representing the action to perform, storing an unbounded amount of information inside the functional value, which therefore cannot be removed.

The five programs that are not curtailed by the termination bound, but still contain residual higher-order expressions, are as follows:

Example 43

The integer and maillist programs pass functional values to primitive functions. The maillist program calls the catch function (see §5.4.5). The integer program passes functional values to the seq primitive, using the following function:

```
seqlist []       = return ()
seqlist (x : xs) = x `seq` seqlist xs
```

This function is invoked with the IO monad, so the return () expression is a functional value. It is impossible to remove this functional value without having access to the implementation of the seq primitive. □

Example 44

The pretty, constraints and mkhprog programs pass functional values to expressions that evaluate to ⊥. The case in pretty comes from the fragment:

```
type Pretty = Int → Bool → PrettyRep

ppBesides :: [Pretty] → Pretty
ppBesides = foldr1 ppBeside
```

Here ppBesides xs evaluates to ⊥ if xs ≡ []. The ⊥ value will be of type Pretty, and will be given further arguments, which can be functional arguments. In reality, the code ensures that the input list is never [], so the program will never fail with this error. □

5.8.3 Termination Bound

The termination bound used varies from 2 to 11 for the sample programs (see Bound in Table 5.1). If we exclude the integer program, which is complicated by the primitive operations on functional values, the highest bound is 8. Most programs have a termination bound of 4. There is no apparent relation between the size of a program and the termination bound.

5.8.4 Creating of Functional Values

We use Yhc generated programs as input, which have been lambda lifted (Johnsson 1985), so contain no lambda expressions. The residual program has no partial application, only lambda expressions. Most programs in our test suite start with hundreds of partial applications, but only 5 residual programs contain lambda expressions (see HO Create in Table 5.1).

For the purposes of testing defunctionalisation, we have worked on unmodified Yhc libraries, including all the low-level detail. For example, readFile in Yhc is implemented in terms of file handles and pointer operations. Most analysis operations work on an abstracted view of the program, which reduces the number and complexity of functional values.

5.8.5 Uses of Functional Values

While very few programs have residual functional values, a substantial number make use of general application, and use over-application of functions (see HO Use in Table 5.1). In most cases these result from supplying error calls with additional arguments, typically related to the desugaring of **do** notation and pattern matching within Yhc.

5.8.6 Execution Time

The timing results were all measured on a 1.2GHz laptop, running GHC 6.8.2 (The GHC Team 2007). The longest execution time was just over one second, with the average time being half a second (see Time in Table 5.1). The programs requiring most time made use of floating point numbers, suggesting that library code requires most effort to defunctionalise. If abstractions were given for library methods, the execution time would drop substantially.

In order to gain acceptable speed, we perform a number of optimisations over the algorithm presented in §5.4. (1) We transform functions in an order determined by a topological sort with respect to the call-graph. (2) We delay the transformation of dictionary components, as these will often be eliminated. (3) We fuse the inlining, arity raising and simplification stages. (4) We track the arity and boxed lambda status of each function.

5.8.7 Program Size

We measure program size by counting the number of lines of Core code. On average the size of the resultant program is smaller by 30% (see Size in

Table 5.1). The decrease in program size is mainly due to the elimination of dictionaries holding references to unnecessary code. An optimising compiler will perform dictionary specialisation, and therefore is likely to also reduce program size. We do not claim that defunctionalisation reduces code size, merely hope to alleviate concerns raised by previous papers (Chin and Darlington 1996) that doing so might cause an explosion in code size.

5.9 Related Work

5.9.1 Reynolds style defunctionalisation

Reynolds style defunctionalisation (Reynolds 1972) is the seminal method for generating a first-order equivalent of a higher-order program.

Example 45

```
map f [] = []
map f (x : xs) = f x : map f xs
```

Reynolds' method works by creating a data type to represent all values that f may take anywhere in the whole program. For instance, it might be:

```
data Function = Head | Tail

apply Head x = head x
apply Tail  x = tail  x

map f [] = []
map f (x : xs) = apply f x : map f xs
```

Now all calls to map head are replaced by map Head. □

Reynolds' method works on all programs. Defunctionalised code is still type safe, but type checking would require a dependently typed language. Others have proposed variants of Reynolds' method that are type safe in the simply typed lambda calculus (Bell et al. 1997), and within a polymorphic type system (Pottier and Gauthier 2004).

The method is complete, removing all possible higher-order functions, and preserves space and time behaviour. The disadvantage is that the transformation essentially embeds a mini-interpreter for the original program into the new program. The control flow is complicated by the extra level of indirection and in practice the `apply` interpreter is a bottleneck for analysis. Various analysis methods have been proposed to reduce the size of the `apply` function, by statically determining a safe subset of the possible functional values at a call site (Cejtin et al. 2000; Boquist and Johnsson 1996).

Reynolds' method has been used as a tool in program calculation (Danvy and Nielsen 2001; Hutton and Wright 2006), often as a mechanism for removing introduced continuations. Another use of Reynolds' method is for optimisation (Meacham 2008), allowing flow control information to be recovered without the complexity of higher-order transformation.

5.9.2 Removing Functional Values

The closest work to ours is by Chin and Darlington (1996), which itself is similar to that of Nelan (1991). They define a higher-order removal method, with similar goals of removing functional values from a program. Their work shares some of the simplification rules, the arity raising and function specialisation. Despite these commonalities, there are big differences between their method and ours.

- Their method makes use of the *types* of expressions, information that must be maintained and extended to work with additional type systems.

- Their method has *no inlining* step, or any notion of boxed lambdas. Functional values within constructors are ignored. The authors suggest the use of deforestation (Wadler 1988) to help remove them, but deforestation transforms the program more than necessary, and still fails to eliminate many functional values.

- Their specialisation step only applies to outermost lambda expressions, not lambdas within constructors.

- To ensure termination of the specialisation step, they *never specialise a recursive function* unless it has all functional arguments passed identically in all recursive calls. This restriction is satisfied by higher-order functions such as `map`, but fails in many other cases.

In addition, functional programs now use monads, IO continuations and type classes as a matter of course. Such features were still experimental when Chin and Darlington developed their method and it did not handle them. Our work can be seen as a successor to theirs, indeed we achieve most of the aims set out in their future work section. We have tried their examples, and can confirm that all of them are successfully handled by our system. Some of their observations and extensions apply equally to our work: for example, they suggest possible methods of removing accumulating functions such as in Example 39.

5.9.3 Partial Evaluation and Supercompilation

The specialisation and inlining steps are taken from existing program optimisers, as is the termination strategy of homeomorphic embedding. A lot of program optimisers include some form of specialisation and so remove some higher-order functions, such as partial evaluation (Jones et al. 1993) and supercompilation (Turchin 1986). We have certainly benefited from ideas in both these areas in developing our algorithms. Our initial attempt at removing functional values involved modifying the supercompiler described in Chapter 4. But the optimiser is not attempting to preserve correspondence to the original program, so will optimise all aspects of the program equally, instead of focusing on the higher-order elements. Overall, the results were poor.

Chapter 6

Pattern-Match Analysis

This chapter describes an automated analysis to check for pattern match errors, which we have called Catch. If Catch reports that a program has no pattern-match errors, then the program is guaranteed not to fail with a pattern-match error at runtime. A proof of the soundness of the analysis is given in Appendix A. §6.1 gives a small example, and §6.2 gives an overview of the checking process for this example. §6.3 introduces a small core functional language and a mechanism for reasoning about this language, §6.4 describes two constraint languages. §6.5 evaluates Catch on programs from the Nofib suite, on a widely-used library and on a larger application program. §6.6 offers comparisons with related work.

6.1 Motivation

Many functional languages support case-by-case definition of functions over algebraic data types, matching arguments against alternative constructor patterns. In the most widely used languages, such as Haskell and ML, alternative patterns need not exhaust all possible values of the relevant datatype; it is often more convenient for pattern matching to be partial. Common simple examples include functions that select components from specific constructions — in Haskell `tail` applies to (:)-constructed lists and `fromJust` to `Just`-constructed values of a `Maybe`-type.

Partial matching does have a disadvantage. Programs may fail at run-time because a case arises that matches none of the available alternatives. Such pattern-match failures are clearly undesirable, and the motivation for this chapter is to avoid them without denying the convenience of partial matching. Our goal is an automated analysis of Haskell 98 programs to check statically that, despite the possible use of partial pattern matching, no pattern-match failure can occur.

The problem of pattern-match failures is a serious one. The *darcs* project (Roundy 2005) is one of the most successful large scale programs written in Haskell. Taking a look at the darcs bug tracker, 13 problems are errors related to the selector function fromJust and 19 are direct pattern-match failures.

Example 46

```
risers :: Ord α ⇒ [α] → [[α]]
risers [] = []
risers [x] = [[x]]
risers (x : y : etc) = if x ⩽ y then (x : s) : ss else [x] : (s : ss)
  where (s : ss) = risers (y : etc)
```

A sample application of this function is:

```
> risers [1, 2, 3, 1, 2]
[[1, 2, 3], [1, 2]]
```

In the last line of the definition, (s : ss) is matched against the result of risers (y : etc). If the result is in fact an empty list, a pattern-match error will occur. It takes a few moments to check manually that no pattern-match failure is possible – and a few more to be sure one has not made a mistake! Turning the risers function over to our analysis tool (which we call Catch), the output is:

```
Checking "Incomplete pattern on line 5"
Program is Safe
```

□

In other examples, where Catch cannot verify pattern-match safety, it can provide information such as sufficient conditions on arguments for safe application of a function.

The problem of checking if a program will terminate with an error is undecidable, because the halting problem (Turing 1937) can be reduced to it.

```
main = case computation of
            () → error "failure"
```

If computation terminates there will be an error, otherwise there will not. Therefore, it will be necessary to make conservative approximations in order to obtain a powerful but correct analysis.

6.1.1 Contributions

The contributions of this chapter include:

- A method for reasoning about pattern-match failures, in terms of a parameterisable constraint language. The method calculates *preconditions* of functions.
- Two separate constraint languages that can be used with our method.
- Details of the Catch implementation which supports the full Haskell 98 language (Peyton Jones 2003), by transforming Haskell 98 programs to a first-order language.
- Results showing success on a number of small examples drawn from the Nofib suite (Partain et al. 2008), and for three larger examples, investigating the scalability of the checker.

6.2 Overview of the Risers Example

This section sketches the process of checking that the risers function from Example 46 does not crash with a pattern-match error.

6.2.1 Conversion to a Core Language

Rather than analyse full Haskell, Catch analyses a first-order core language, without lambda expressions, partial application or let bindings. The result

```
risers x = case x of
   [] → []
   (y : ys) → case ys of
      [] → (y : []) : []
      (z : zs) → risers2 (risers3 z zs) (y ⩽ z) y

risers2 x y z = case y of
   True → (z : snd x) : (fst x)
   False → (z : []) : (snd x : fst x)

risers3 x y = risers4 (risers (x : y))

risers4 x = case x of
   (y : ys) → (ys, y)
   [] → error "Pattern Match Failure, 11:12."
```

Figure 6.1: risers in the core language.

of converting the risers program to Core Haskell, with identifiers renamed for ease of human reading, is shown in Figure 6.1.

The type of risers is polymorphic over types in the Ord class. Catch can check risers assuming that Ord methods do not raise pattern-match errors, and may return any value. Or a type instance such as Int can be specified with a type signature. To keep the example simple, we have chosen the latter.

6.2.2 Analysis of risers – a brief sketch

In the Core language every pattern match covers all possible constructors of the appropriate type. The alternatives for constructor cases not originally given are calls to error. The analysis starts by finding calls to error, then tries to prove that these calls will not be reached. The one error call in risers4 is avoided under the precondition (see §6.3.4):

risers4, x⋖(:)

That is, all callers of risers4 must supply an argument x which is a (:)-constructed value. For the proof that this precondition holds, two entailments are required (see §6.3.5):

```
x⋖(:) ⇒ (risers x       )⋖(:)
True  ⇒ (risers2 x y z)⋖(:)
```

```
type Selector = (CtorName, Int)

var   :: VarName → Maybe (Expr, Selector)
isRec :: Selector  → Bool
```

Figure 6.2: Operations on Core.

The first line says that if the argument to risers is a (:)-constructed value, the result will be. The second states that the result from risers2 is always (:)-constructed.

6.3 Pattern Match Analysis

This section describes the method used to calculate preconditions for functions. Our method is composed of two parts – an analysis algorithm and a constraint language. The analysis works on a Core functional language, and requires only a small number of constraint operations, none of which work with the Core language. By changing the constraint language, we can change the accuracy and execution time of the tool.

We first give the core language for the analysis in §6.3.1, then the operations that constraints must provide in §6.3.2. We then introduce a simple constraint language in §6.3.3, which we use to illustrate our analysis. First we define three terms:

- A **constraint** describes a (possibly infinite) set of values. We say a value *satisfies* a constraint if the value is within the set.
- A **precondition** is a proposition combining constraints on the arguments to a function, to ensure the result does not contain $\perp$, where $\perp$ is the result of evaluating error. For example, the precondition on tail xs is that xs is (:)-constructed.
- An **entailment** is a proposition combining constraints on the arguments to a function, to ensure the result satisfies a further constraint. For example, xs is (:)-constructed ensures null xs evaluates to False.

6.3.1 Reduced Core language

We use a first-order core language, a subset of the Core language presented in §2.1. In order to eliminate all constructs which either create or make use of functional values we eliminate lambda expressions and general application, and require all function and constructor applications to be fully-applied.

Figure 6.2 gives the signatures for two helper functions over the core data types. The var function returns Nothing for a variable bound as the argument of a top-level function, and Just $(\mathsf{e}, (\mathsf{c}, \mathsf{i}))$ for a variable bound as the ith component in the c-constructed alternative of a case-expression whose scrutinee is e. We assume unique variable names throughout the program.

Example 47

Given the definition:

```
map f xs = case xs of
                [ ]    → [ ]
                y : ys → f y : map f ys
```

We would obtain the following results:

```
var "f"  = Nothing
var "xs" = Nothing
var "y"  = Just (⟦"xs"⟧, (":", 0))
var "ys" = Just (⟦"xs"⟧, (":", 1))
```

□

The isRec (c, i) function returns true if the constructor c has a recursive ith component. For example, let hd $=$ $(":", 0)$ and tl $=$ $(":", 1)$ then isRec hd $=$ False but isRec tl $=$ True. We require that for any value of a particular type, a non-recursive selector must only occur some maximum number of times.

Algebraic Abstractions of Primitive Types

Our Core language only has algebraic data types. Catch allows for primitive types such as characters and integers by abstracting them into algebraic types. Two abstractions used in Catch are:

```
data Prop α

(∧), (∨)   :: Prop α → Prop α → Prop α
andP, orP :: [Prop α] → Prop α
mapP      :: (α → Prop β) → Prop α → Prop β
true, false :: Prop α
lit        :: α → Prop α

bool :: Bool → Prop α
bool b = if b then true else false

isTrue :: Prop () → Bool
isTrue = (≡) true

tautP :: (α → Bool) → Prop α → Bool
tautP f = isTrue ∘ mapP (bool ∘ f)
```

Figure 6.3: Proposition data type.

```
data Int = Neg | Zero | One | Pos
data Char = Char
```

Knowledge about values is encoded as *a set of* possible constructions. In our experience, integers are most often constrained to be a natural, or to be non-zero. Addition or subtraction of one is the most common operation. Though very simple, the Int abstraction models the common properties and operations quite well. For characters, we have found little benefit in any refinement other than considering all characters to be abstracted to the same value.

The final issue of abstraction relates to primitive functions in the IO monad, such as getArgs (which returns the command-line arguments), or readFile (which reads from the file-system). In most cases an IO function is modelled as returning *any* value of the correct type, using a function primitive to the checker.

6.3.2 Constraint Essentials and Notation

We write Sat x c to assert that the value of expression x must be a member of the set described by the constraint c, i.e. that x *satisfies* c. If any component of x evaluates to $\perp$, the constraint is automatically satisfied: in our method,

```
data Sat α = Sat α Constraint

(<) :: α → [CtorName] → Prop (Sat α)
(▷) :: Selector → Constraint → Constraint
(◁) :: CtorName → Constraint → Prop (Sat Int)
```

Figure 6.4: Constraint operations.

```
precond :: FuncName → Prop (Sat VarName)
prePost :: FuncName → Constraint → Prop (Sat VarName)
reduce  :: Prop (Sat Expr) → Prop (Sat VarName)

substP :: Eq α ⇒ [(α, β)] → Prop (Sat α) → Prop (Sat β)
substP xs = mapP (λ(Sat i k) → lit $ Sat (fromJust $ lookup i xs) k)
```

Figure 6.5: Operations to generate preconditions and entailments.

for a component of x to evaluate to $\perp$, some other constraint must have been violated, so an error is still reported. Atomic constraints can be combined into propositions, using the proposition data type and operations in Figure 6.3.

Several underlying constraint models are possible. To keep the introduction of the algorithms simple we first use *basic pattern constraints* (§6.3.3), which are unsuitable for reasons given in §6.3.8. We then describe *regular expression constraints* in §6.4.1 – a variant of the constraints used in earlier versions of Catch. Finally we present *multi-pattern constraints* in §6.4.2 – used in the current Catch tool to enable scaling to much larger problems.

Three operations must be provided by every constraint model, whose signatures are given in Figure 6.4. The lifting and splitting operators (▷) and (◁) are discussed in §6.3.5. The expression x<cs generates a predicate ensuring that the value x must be constructed by one of the constructors in cs.

The type signatures for the functions calculating preconditions and entailments are given in Figure 6.5. The precond function takes a function name, and gives a proposition imposing constraints on the arguments to that function, denoted by argument position. The prePost function takes a function name and a postcondition, and gives a precondition sufficient to ensure the postcondition. During the manipulation of constraints, we often need to talk about constraints on expressions, rather than argument positions: the

```
data Constraint = Any
                | Con CtorName [Constraint]
```

Figure 6.6: Basic pattern constraints.

reduce function converts propositions of constraints on expressions to equivalent propositions of constraints on arguments. The substP function goes in the opposite direction, replacing constraints on argument positions with the substituted argument expressions.

6.3.3 Basic Pattern (BP) Constraints

For simplicity, our analysis framework will be introduced using basic pattern constraints (BP-constraints). BP-constraints are defined in Figure 6.6, and correspond to Haskell pattern matching, where Any represents an unrestricted match. A data structure satisfies a BP-constraint if it matches the pattern. For example, the requirement for a value to be (:)-constructed would be expressed as (Con ":" [Any, Any]). The BP-constraint language is limited in expressivity, for example it is impossible to state that all the elements of a boolean list are True.

As an example of an operator definition for the BP-constraint language, (⋖) can be defined:

```
a⋖xs = orP [lit (a `Sat` anys x) | x ← xs]
  where anys x = Con x (replicate (arity x) Any)
```

So, for example:

```
e⋖["True"]     = lit (e `Sat` Con "True" [])
e⋖[":"]        = lit (e `Sat` Con ":" [Any, Any])
e⋖[":", "[]"] = lit (e `Sat` Con ":" [Any, Any]) ∨
                 lit (e `Sat` Con "[]" [])
```

6.3.4 Preconditions for Pattern Safety

Our intention is that for every function, a proposition combining constraints on the arguments forms a precondition to ensure the result does not contain

```
pre :: Expr → Prop (Sat Expr)
pre ⟦v⟧              = true
pre ⟦c xs⟧           = andP (map pre xs)
pre ⟦f xs⟧           = pre′ f xs ∧ andP (map pre xs)
  where pre′ f xs = substP (zip (args f) xs) (precond f)
pre ⟦case x of as⟧ = pre x ∧ andP (map alt as)
  where alt ⟦c vs → y⟧ = x⊲(ctors c \ [c]) ∨ pre y
```

Figure 6.7: Precondition of an expression, pre.

⊥. The precondition for error is False. A program is safe if the precondition on main is True. Our analysis method derives these preconditions. Given precond which returns the precondition of a function, we can determine the precondition of an *expression* using the pre function in Figure 6.7. The intuition behind pre is that in all subexpressions f xs, the arguments xs must satisfy the precondition for f. The only exception is that a case expression is safe if the scrutinee is safe, and each alternative is either safe, or never taken.

Example 48

```
safeTail xs = case null xs of
                   True  → [ ]
                   False → tail xs
```

The precondition for safeTail is computed as:

```
pre′ null [xs] ∧ (null xs⊲["True"] ∨ pre′ tail [xs])
```

This predicate states that the invocation of null xs must be safe, and either null xs is True or tail xs must be safe. □

Stable Preconditions

The iterative algorithm for calculating preconditions is given in Figure 6.8. Initially all preconditions are assumed to be true, apart from the error precondition, which is false. In each iteration we calculate the precondition using the pre function from Figure 6.7, using the previous value of precond.

```
precond :: FuncName → Prop (Sat VarName)
precond_0   f = if f ≡ "error" then false else true
precond_n+1 f = precond_n f ∧ reduce (pre{precond_n}(body f))
```

Figure 6.8: Precondition calculation.

Each successive precondition is conjoined with the previous one, and is therefore more restrictive. So *if all chains of increasingly restrictive propositions of constraints are finite*, termination is guaranteed – a topic we return to in §6.3.8.

We can improve the efficiency of the algorithm by tracking dependencies between preconditions, and performing the minimum amount of recalculation. Finding strongly connected components in the static call graph of a program allows parts of the program to be checked separately.

Preconditions and Laziness

The pre function defined in Figure 6.7 does not exploit laziness. The function application equation demands that preconditions hold on *all* arguments – which is unnecessarily restrictive unless a function is strict in all arguments. For example, the precondition on `False && error "here"` is False, when it should be True. In general, preconditions may be more restrictive than necessary. However, investigation of a range of examples suggests that inlining (&&) and (||) captures many of the common cases where laziness would be required.

6.3.5 Manipulating constraints

The pre function generates constraints in terms of expressions, which the precond function transforms into constraints on function arguments, using reduce. The reduce function is defined in Figure 6.9. We will first give an example of how reduce works, followed by a description of each rule corresponding to an equation in the definition of red.

```
reduce :: Prop (Sat Expr) → Prop (Sat VarName)
reduce = mapP (λ(Sat x k) → red x k)

red :: Expr → Constraint → Prop (Sat VarName)
red ⟦v⟧            k = case var v of
                           Nothing    → lit (v `Sat` k)
                           Just (x, s) → red x (s ▷ k)
red ⟦c x̅s̅⟧          k = reduce $ substP (zip [0 ..] x̅s̅) (c ◁ k)
red ⟦f x̅s̅⟧          k = reduce $ substP (zip (args f) x̅s̅) (prePost f k)
red ⟦case x of a̅s̅⟧ k = andP (map alt a̅s̅)
  where alt ⟦c v̅s̅ → y⟧ = reduce (x⋖(ctors c \ [c])) ∨ red y k
```

Figure 6.9: Specification of constraint reduction, reduce.

Example 48 (revisited)

The precondition for the safeTail function is:

$\mathsf{pre}'\ \mathsf{null}\ [\mathsf{xs}]\ \wedge\ (\mathsf{null}\ \mathsf{xs}\lessdot[\texttt{"True"}]\ \vee\ \mathsf{pre}'\ \mathsf{tail}\ [\mathsf{xs}])$

We can use the preconditions computed for tail and null to rewrite the precondition as:

$\mathsf{null}\ \mathsf{xs}\lessdot[\texttt{"True"}]\ \vee\ \mathsf{xs}\lessdot[\texttt{":"}]$

Now we use an entailment calculated by prePost to turn the constraint on null's result into a constraint on its argument:

$\mathsf{xs}\lessdot[\texttt{"[]"}]\ \vee\ \mathsf{xs}\lessdot[\texttt{":"}]$

Which can be shown to be a tautology. □

The variable rule has two alternatives. The first alternative deals with top-level bound arguments, which are already in the correct form. The other alternative applies to variables bound by patterns in case alternatives. It lifts conditions on a bound variable to the scrutinee of the case expression in which they occur. The ▷ operator lifts a constraint on one part of a data structure to a constraint on the entire data structure. For BP-constraints, ▷ can be defined as:

```
(c, i) ▷ k = Con c  [ if i ≡ j then k else Any
                    | j ← [0 .. arity c − 1]]
```

```
prePost :: FuncName → Constraint → Prop (Sat VarName)
prePost_0   f k = true
prePost_n+1 f k = prePost_n f k ∧ reduce_n (lit $ body f `Sat` k)
  where reduce_n = reduce using prePost_n
```

Figure 6.10: Fixed point calculation for prePost.

Example 49

```
case xs of
    []     → []
    y : ys → tail y
```

Here the initial precondition will be y<[":"], which evaluates to the result y `Sat` Con ":" [Any, Any]. The var function on y gives Right (xs, (":", 0)). After the application of $\rhd$ the revised constraint refers to xs instead of y, and will be xs `Sat` Con ":" [Con ":" [Any, Any], Any]. We have gone from a constraint on y, using the knowledge that y is bound to a portion of xs, to a constraint on xs. □

The constructor application rule deals with an application of a constructor. The $\lhd$ operator splits a constraint on an entire structure into a proposition combining constraints on each part.

```
c ◁ Any        = true
c ◁ Con c2 xs = bool (c2 ≡ c) ∧ andP (map lit (zipWith Sat [0..] xs))
```

The intuition is that given knowledge of the root constructor of a data value, we can reformulate the constraint in terms of what the constructor fields must satisfy. Some sample applications:

```
"True"  ◁ Con "True" [] = true
"False" ◁ Con "True" [] = false
":" ◁ Con ":" [Con "True" [], Any] =
  lit (0 `Sat` Con "True" []) ∧ lit (1 `Sat` Any)
```

The case rule generates a conjunct for each alternative. An alternative satisfies a constraint if *either* it is never taken, *or* it meets the constraint when taken.

The function application rule relies on the prePost function defined in Figure 6.10. This function calculates the precondition necessary to ensure a given postcondition on a function, which forms an entailment. Like the precondition calculation in §6.3.4, the prePost function works iteratively, with each result becoming increasingly restrictive. Initially, all postconditions are assumed to be true. The iterative step takes the body of the function, and uses the reduce transformation to obtain a predicate in terms of the arguments to the function, using the previous value of prePost. If refinement chains of constraint/function pairs are finite, termination is guaranteed. Here again, a speed up can be obtained by tracking the dependencies between constraints, and additionally caching all calculated results.

6.3.6 Semantics of Constraints

The semantics of a constraint are determined by which values satisfy it. We can model values in our first-order Core language with the data type:

```
data Value = Value CtorName [Value]
           | Bottom
```

Given this value representation, we can use the sat function to determine whether a value satisfies a constraint, implemented in terms of the ◁ operator:

```
sat :: Sat Value → Bool
sat (Sat Bottom       k) = True
sat (Sat (Value c xs) k) = sat′ $ substP (zip [0 ..] xs) (c ◁ k)

sat′ :: Prop (Sat Value) → Bool
sat′ = tautP sat
```

The first equation returns True given a value of type Bottom, as if a value contains ⊥ then any constraint is true. In order to be consistent, the constraint operations must respect the following two properties – both of which permit constraints to be more restrictive than necessary. In Appendix A we use these properties to prove soundness of the analysis.

Property C1

sat′ (Value c xs⋖cs) ⇒ c ∈ cs

```
satE′ :: Prop (Sat Expr) → Bool
satE′ = tautP satE

satE :: Sat Expr → Bool
satE (Sat x k) = sat (Sat (eval x) k)

isBottom :: Value → Bool
isBottom Bottom = True
isBottom (Value c xs) = any isBottom xs

eval :: Expr → Value
eval = ...   -- evaluate an expression to normal form
```

Figure 6.11: Auxiliary definitions for the soundness theorem.

The first property requires that v<cs must not match values constructed by constructors not in cs.

Property C2

sat $ Sat (Value c xs) ((c, i) ▷ k) ⇒ sat $ Sat (xs !! i) k

The second property requires that if a constraint satisfies a value after they have both been extended, then the original value must have satisfied the original constraint. For example, if Just x `Sat` (("Just", 0) ▷ k) is true, then x `Sat` k must be true.

6.3.7 Soundness Theorem

Our analysis is sound if for any expression e, provided the precondition of e is calculated to be true, then the evaluation of e will not result in ⊥. Using the auxiliary definitions given in Figure 6.11 we can express this theorem as:

satE′ $ pre e ⇒ not $ isBottom $ eval e

In order to evaluate an expression to normal form, it is necessary for the expression to be closed and for evaluation of the expression to terminate. If the expression e can be evaluated to normal form, and both e and the program under analysis are first-order Core as described in §6.3.1, then we prove the soundness theorem in Appendix A.

6.3.8 Finite Refinement of Constraints

With unbounded recursion in patterns, the BP-constraint language does *not* have only finite chains of refinement, as constraints can become infinitely long. As we saw in §6.3.4, we need this property for termination of the iterative analysis. In the next section we introduce two alternative constraint systems. Both share a key property: *for any type, there are finitely many constraints.*

6.4 Richer but Finite Constraint Systems

There are many ways of defining a richer constraint system, while also ensuring the necessary finiteness properties. Here we outline two – both implemented in Catch. Neither is strictly more powerful than the other; each is capable of expressing constraints that the other cannot express.

When designing a constraint system, the main decision is which distinctions between data values to ignore. Since the constraint system must be finite, there must be sets of data values which no constraint within the system can distinguish between. As the constraint system stores more information, it will distinguish more values, but will likely take longer to obtain fixed points. The two constraint systems in this section were developed by looking at examples, and trying to find systems offering sufficient power to solve real problems, but still remain bounded.

6.4.1 Regular Expression (RE) Constraints

An implementation of regular expression based constraints (RE-constraints) is given in Figure 6.12. In a constraint of the form $(\mathsf{r} \rightsquigarrow \mathsf{cs})$, r is a regular expression and cs is a set of constructors. Such a constraint is satisfied by a data structure d if every sequence of selectors which is applicable to d, and described by r, reaches a constructor in the set cs. If no such sequence of selectors has a well-defined result then the constraint is vacuously true.

Concerning the helper functions needed to define $\rhd$ and $\lhd$ in Figure 6.12, the differentiate function is from Conway (1971); integrate is its inverse; ewp is the empty word property.

```
data Constraint = RegExp ⇝ [CtorName]
type RegExp     = [RegItem]
data RegItem    = Atom Selector | Star [Selector]

(⋖) :: α → [CtorName] → Prop (Sat α)
e⋖cs = lit $ e `Sat` ([] ⇝ cs)

(▷) :: Selector → Constraint → Constraint
p ▷ (r ⇝ cs) = integrate p r ⇝ cs

(◁) :: CtorName → Constraint → Prop (Sat Int)
c ◁ (r ⇝ cs) = bool (not (ewp r) || c ∈ cs) ∧
  andP (map f [0 . . arity c − 1])
  where
  f i = case differentiate (c, i) r of
            Nothing → true
            Just r₂  → lit $ i `Sat` (r₂ ⇝ cs)

ewp :: RegExp → Bool
ewp x = all isStar x
  where isStar (Star  _) = True
        isStar (Atom _) = False

integrate :: Selector → RegExp → RegExp
integrate p r | not (isRec p) = Atom p : r
integrate p (Star ps : r)     = Star (nub (p : ps)) : r
integrate p r                 = Star [p] : r

differentiate :: Selector → RegExp → Maybe RegExp
differentiate p [] = Nothing
differentiate p (Atom r : rs) | p ≡ r      = Just rs
                              | otherwise = Nothing
differentiate p (Star  r : rs) | p ∈ r      = Just (Star r : rs)
                              | otherwise = differentiate p rs
```

Figure 6.12: RE-constraints.

In earlier versions of Catch, regular expressions were unrestricted and quickly grew to an unmanageable size, preventing analysis of larger programs. In general, a regular expression takes one of six forms:

$\mathsf{r}_1 + \mathsf{r}_2$	union of regular expressions r_1 and r_2
$\mathsf{r}_1 \cdot \mathsf{r}_2$	concatenation of regular expressions r_1 then r_2
$\mathsf{r}_1{}^*$	any number (possibly zero) occurrences of r_1
sel	a selector, i.e. hd for the head of a list
0	the language is the empty set
1	the language is the set containing the empty string

We implement REs using the data type RegExp from Figure 6.12, with RegExp being a list of concatenated RegItem. In addition to the restrictions imposed by the data type, we require: (1) within Atom the Selector is not recursive; (2) within Star there is a non-empty list of Selectors, each of which is recursive; (3) no two Star constructors are adjacent in a concatenation. Our restricted regular expressions have the following grammar:

$$\begin{aligned}
\mathsf{re}' &= \mathsf{item} \mid \mathsf{item}\cdot\mathsf{sel}_\mathsf{N}\cdot\mathsf{re}' \mid \mathsf{sel}_\mathsf{N}\cdot\mathsf{re}' \mid 1\\
\mathsf{item} &= \mathsf{stars}^*\\
\mathsf{stars} &= \mathsf{sel}_\mathsf{R} \mid \mathsf{sel}_\mathsf{R} + \mathsf{stars}\\
\mathsf{sel}_\mathsf{N} &= \text{non-recursive selector}\\
\mathsf{sel}_\mathsf{R} &= \text{recursive selector}
\end{aligned}$$

For example, the regular expression $\mathsf{tl}^* \cdot \mathsf{tl}^*$ is disallowed as there are two adjacent stars, hd^* is disallowed as a non-recursive selector under a star, and tl is disallowed as a recursive selector without a star. When used within RE-constraints, these restrictions ensure three properties:

- Because of static typing, constructor-sets must all be of the same type.
- There are finitely many restricted regular expressions for any type. Combined with the finite number of constructors, this property is sufficient to guarantee termination when computing a fixed-point iteration on constraints.
- The restricted REs with 0 are closed under integration and differentiation. (The 0 alternative is catered for by the Maybe return type in the differentiation. As $0 \rightsquigarrow \mathsf{c}$ always evaluates to True, $\lhd$ replaces Nothing by True.)

Example 50

(head xs) is safe if xs evaluates to a non-empty list. The RE-constraint generated by Catch is: xs \`Sat\` $(1 \rightsquigarrow \{:\})$. This may be read: from the root of the value xs, after following an empty path of selectors, we reach a (:)-constructed value. □

Example 51

(map head xs) is safe if xs evaluates to a list of non-empty lists. The RE-constraint is: xs \`Sat\` $(\mathsf{tl}^{*} \cdot \mathsf{hd} \rightsquigarrow \{:\})$. From the root of xs, following any number of tails, then exactly one head, we reach a (:). If xs is [], it still satisfies the constraint, as there are no well defined paths containing a hd selector. If xs is infinite then all its infinitely many elements must be (:)-constructed. □

Example 52

(map head (reverse xs)) is safe if every item in xs is (:)-constructed, or if xs is infinite – so reverse does not terminate. The RE-constraint is: xs \`Sat\` $(\mathsf{tl}^{*} \cdot \mathsf{hd} \rightsquigarrow \{:\}) \vee$ xs \`Sat\` $(\mathsf{tl}^{*} \rightsquigarrow \{:\})$. The second term specifies the infinite case: if the list xs is (:)-constructed, it will have a tl selector, and therefore the tl path is well defined and requires the tail to be (:). Each step in the chain ensures the next path is well defined, and therefore the list is infinite. □

Finite Number of RE-Constraints

We require that for any type, there are finitely many constraints (see §6.3.8). We can model types as:

```
data Type = Type [Ctor]
type Ctor  = [Maybe Type]
```

Each Type has a number of constructors. For each constructor Ctor, every component has either a recursive type (represented as Nothing) or a non-recursive type t (represented as Just t). As each non-recursive type is structurally smaller than the original, a function that recurses on the type will terminate. We define a function count which takes a type and returns the number of possible RE-constraints.

```
count :: Type → Integer
count (Type t) = 2 ^ rec * (2 ^ ctor + sum (map count nonrec))
  where
  rec = length (filter isNothing (concat t))
  nonrec = [x | Just x ← concat t]
  ctor = length t
```

The 2 ^ rec term corresponds to the number of possible constraints under Star. The 2 ^ ctor term accounts for the case where the selector path is empty.

RE-Constraint Propositions

Catch computes over propositional formulae with constraints as atomic propositions. Among other operators on propositions, they are compared for equality to obtain a fixed point. All the fixed-point algorithms given in this chapter stop once equal constraints are found. We use Binary Decision Diagrams (BDD) (Lee 1959) to make these equality tests fast. Since the complexity of performing an operation is often proportional to the number of atomic constraints in a proposition, we apply simplification rules to reduce this number. For example, the three simplest of the nineteen rules are:

Exhaustion: In the constraint x `Sat` (r ⇝ [":", "[]"]) the condition lists all the possible constructors. Because of static typing, x must be one of these constructors. Any such constraint simplifies to True.

And merging: The conjunction e `Sat` (r ⇝ c_1) ∧ e `Sat` (r ⇝ c_2) can be replaced by e `Sat` (r ⇝ ($c_1 \cap c_2$)).

Or merging: The disjunction e `Sat` (r ⇝ c_1) ∨ e `Sat` (r ⇝ c_2) can be replaced by e `Sat` (r ⇝ c_2) if $c_1 \subseteq c_2$.

```
type Constraint = [Val]
data Val        = [Pattern] ⋆ [Pattern] | Any
data Pattern    = Pattern CtorName [Val]

  -- useful auxiliaries, non recursive selectors
nonRecs :: CtorName → [Int]
nonRecs c = [i | i ← [0 .. arity c − 1], not (isRec (c, i))]

  -- a complete Pattern on c
complete :: CtorName → Pattern
complete c = Pattern c (map (const Any) (nonRecs c))

(⋖) :: α → [CtorName] → Prop (Sat α)
e⋖cs = lit $ Sat e [ map complete cs
                   ⋆ map complete (ctors (head cs))
                   | not (null cs)]

(▷) :: Selector → Constraint → Constraint
(c, i) ▷ k = map f k
  where
  f Any = Any
  f (ms1 ⋆ ms2) | isRec (c, i) = [complete c] ⋆ merge ms1 ms2
  f v = [Pattern c [if i ≡ j then v else Any | j ← nonRecs c]]
        ⋆ map complete (ctors c)

(◁) :: CtorName → Constraint → Prop (Sat Int)
c ◁ vs = orP (map f vs)
  where
  (rec, non) = partition (isRec ∘ (, ) c) [0 .. arity c − 1]

  f Any = true
  f (ms1 ⋆ ms2) = orP [andP $ map lit $ g vs1
                      | Pattern c1 vs1 ← ms1, c1 ≡ c]
    where g vs = zipWith Sat non (map (:[]) vs) ++
                 map ( `Sat` [ms2 ⋆ ms2]) rec

(⊓) :: Val → Val → Val
(a1 ⋆ b1) ⊓ (a2 ⋆ b2) = merge a1 a2 ⋆ merge b1 b2
x         ⊓ y         = if x ≡ Any then y else x

merge :: [Pattern] → [Pattern] → [Pattern]
merge ms1 ms2 = [Pattern c1 (zipWith (⊓) vs1 vs2) |
      Pattern c1 vs1 ← ms1, Pattern c2 vs2 ← ms2, c1 ≡ c2]
```

Figure 6.13: MP-constraints.

6.4.2 Multipattern (MP) Constraints & Simplification

Although RE-constraints are capable of solving many examples, they suffer from a problem of scale. As programs become more complex the size of the propositions grows quickly, slowing Catch unacceptably. Multipattern constraints (MP-constraints, defined in Figure 6.13) are an alternative which scales better.

MP-constraints are similar to BP-constraints, but can constrain an infinite number of items. A value v satisfies a constraint $\mathsf{p_1} \star \mathsf{p_2}$ if v itself satisfies the pattern $\mathsf{p_1}$ and *all its subcomponents of the same type as* v satisfy $\mathsf{p_2}$. We call $\mathsf{p_1}$ the root pattern, and $\mathsf{p_2}$ the recursive pattern. Each of $\mathsf{p_1}$ and $\mathsf{p_2}$ is given as a set of matches similar to BP-constraints, but each Pattern only specifies the values for the non-recursive selectors, all recursive selectors are handled by $\mathsf{p_2}$. A constraint is a disjunctive list of $\star$ patterns.

The intuition behind the definition of $(\mathsf{c}, \mathsf{i}) \rhd \mathsf{ps}$ is that if the selector (c, i) is recursive, given a pattern $\alpha \star \beta$, the new root pattern requires the value to be c-constructed, and the recursive patterns become $\mathsf{merge}\ \alpha\ \beta$ – i.e. all recursive values must satisfy both the root and recursive patterns of the original pattern. If the selector is non-recursive, then each new pattern contains the old pattern within it, as the appropriate non-recursive field. So, for example:

$$
\begin{array}{lll}
\mathsf{hd} \rhd (\alpha \star \beta) = \{(:)\ (\alpha \star \beta)\} \star \{[\,], (:)\ \mathsf{Any}\} \\
\mathsf{tl}\ \ \rhd (\alpha \star \beta) = \{(:)\ \mathsf{Any} \quad\ \ \} \star (\mathsf{merge}\ \alpha\ \beta)
\end{array}
$$

For the $\lhd$ operator, if the root pattern matches, then all non-recursive fields are matched to their non-recursive constraints, and all recursive fields have their root and recursive patterns become their recursive pattern. In the result, each field is denoted by its argument position. So, for example:

$$
\begin{array}{l}
\texttt{":"} \lhd (\{[\,] \quad\ \} \star \beta) = \mathsf{false} \\
\texttt{":"} \lhd (\{(:)\ \alpha\} \star \beta) = 0\ \text{`Sat`}\ \alpha\ \wedge\ 1\ \text{`Sat`}\ (\beta \star \beta)
\end{array}
$$

Example 50 (revisited)

Safe evaluation of (head xs) requires xs to be non-empty. The MP-constraint generated by Catch on xs is: $\{(:)\ \mathsf{Any}\} \star \{[\,], (:)\ \mathsf{Any}\}$. This constraint can be read in two portions: the part to the left of $\star$ requires the value to be (:)-constructed, with an unrestricted hd field; the right allows either a [] or a (:) with an unrestricted hd field, and a tl field restricted by the constraint on the right of the $\star$. In this particular case, the right of the $\star$ places no restrictions on the value. This constraint is longer than the corresponding RE-constraint as it makes explicit that both the head and the recursive tails are unrestricted. □

Example 51 (revisited)

Safe evaluation of (map head xs) requires xs to be a list of non-empty lists. The MP-constraint on xs is:

$\{[\,], (:)\ (\{(:)\ \mathsf{Any}\} \star \{[\,], (:)\ \mathsf{Any}\})\} \star$
$\{[\,], (:)\ (\{(:)\ \mathsf{Any}\} \star \{[\,], (:)\ \mathsf{Any}\})\}$

□

Example 52 (revisited)

(map head (reverse x)) requires xs to be a list of non-empty lists *or* infinite. The MP-constraint for an infinite list is: $\{(:)\ \mathsf{Any}\} \star \{(:)\ \mathsf{Any}\}$ □

MP-constraints also have simplification rules. For example, the two simplest of the eight rules are:

Val-list simplification: Given a Val-list, if the value Any is in this list, the list is equal to [Any]. If a value occurs more than once in the list, one copy can be removed.

Val simplification: If both p_1 and p_2 cover all constructors and all their components have Any as their constraint, the constraint $\mathsf{p}_1 \star \mathsf{p}_2$ can be replaced with Any.

Finitely Many MP-Constraints per Type

As in §6.4.1, we show there are finitely many constraints per type by defining a count function:

```
count :: Type → Integer
count (Type t) = 2^val t
  where val t = 1 + 2 * 2^(pattern t)

pattern t = sum (map f t)
  where f c = product [count t2 | Just t2 ← c]
```

The val function counts the number of possible Val constructions. The pattern function performs a similar role for Pattern constructions.

MP-Constraint Propositions and Uncurrying

A big advantage of MP-constraints is that if two constraints on the same expression are combined at the proposition level, they can be reduced into one atomic constraint:

$$(\mathsf{Sat}\ \mathsf{e}\ \mathsf{v}_1)\ \vee\ (\mathsf{Sat}\ \mathsf{e}\ \mathsf{v}_2) = \mathsf{Sat}\ \mathsf{e}\ (\mathsf{v}_1 +\!\!+\ \mathsf{v}_2)$$
$$(\mathsf{Sat}\ \mathsf{e}\ \mathsf{v}_1)\ \wedge\ (\mathsf{Sat}\ \mathsf{e}\ \mathsf{v}_2) = \mathsf{Sat}\ \mathsf{e}\ [\mathsf{a} \sqcap \mathsf{b} \mid \mathsf{a} \leftarrow \mathsf{v}_1, \mathsf{b} \leftarrow \mathsf{v}_2]$$

This ability to combine constraints on equal expressions can be exploited further by translating the program to be analysed. After applying reduce, all constraints will be in terms of the arguments to a function. So if all functions took exactly one argument then *all* the constraints associated with a function could be collapsed into one. We therefore *uncurry* all functions.

Example 53

```
(||) x y = case x of
              True  → True
              False → y
```

in uncurried form becomes:

```
(||) a = case a of
           (x, y) → case x of
                       True  → True
                       False → y
```

□

Combining MP-constraint reduction rules with the uncurrying transformation makes Sat α equivalent in power to Prop (Sat α). This simplification reduces the number of different propositional constraints, making fixed-point computations faster. In the RE-constraint system uncurrying would do no harm, but it would be of no use, as no additional simplification rules would apply.

6.4.3 Comparison of Constraint Systems

As we discussed in §6.3.8, it is not possible to use BP-constraints, as they do not have finite chains of refinement. Both RE-constraints and MP-constraints are capable of expressing a wide range of value-sets, but neither subsumes the other. We give examples where one constraint language can differentiate between a pair of values, and the other cannot.

Example 54

Let $\mathsf{v}_1 = (\mathsf{T} : [\,])$ and $\mathsf{v}_2 = (\mathsf{T} : \mathsf{T} : [\,])$ and consider the MP-constraint $\{(:)\ \mathsf{Any}\} \star \{[\,]\}$. This constraint is satisfied by v_1 but not by v_2. No proposition over RE-constraints can separate these two values. □

Example 55

Consider a data type:

$$\begin{aligned}\textbf{data}\ \mathsf{Tree}\ \alpha = {} & \mathsf{Branch}\{\mathsf{left} :: \mathsf{Tree}\ \alpha, \mathsf{right} :: \mathsf{Tree}\ \alpha\} \\ \mid {} & \mathsf{Leaf}\quad \{\mathsf{leaf} :: \alpha\}\end{aligned}$$

and two values of the type $\mathsf{Tree\ Bool}$

$$\begin{aligned}\mathsf{v}_1 &= \mathsf{Branch}\ (\mathsf{Leaf\ True})\ (\mathsf{Leaf\ False}) \\ \mathsf{v}_2 &= \mathsf{Branch}\ (\mathsf{Leaf\ False})\ (\mathsf{Leaf\ True})\end{aligned}$$

The RE-constraint $(\mathsf{left}^{*}\cdot\mathsf{leaf} \leadsto \mathsf{True})$ is satisfied by v_1 but not v_2. No MP-constraint separates the two values. □

We have implemented both constraint systems in Catch. Factors to consider when choosing which constraint system to use include: how readable the constraints are, expressive power, implementation complexity and scalability. In practice the issue of scalability is key: how large do constraints become, how quickly can they be manipulated, how expensive is their simplification. Catch uses MP-constraints by default, as they allow much larger examples to be checked.

6.5 Results and Evaluation

The best way to see the power of Catch is by example. §6.5.1 discusses in general how some programs may need to be modified to obtain provable safety. §6.5.2 investigates all the examples from the Imaginary section of the Nofib suite (Partain et al. 2008). To illustrate results for larger and widely-used applications, §6.5.3 investigates the FiniteMap library, §6.5.4 investigates the HsColour program and §6.5.5 reports on XMonad.

6.5.1 Modifications for Verifiable Safety

Take the following example:

```
average xs = sum xs `div` length xs
```

If xs is [] then a division by zero occurs, modelled in Catch as a pattern-match error. One small local change could be made which would remove this pattern match error:

```
average xs = if null xs then 0 else sum xs `div` length xs
```

Now if xs is [], the program simply returns 0, and no pattern match error occurs. In general, pattern-match errors can be avoided in two ways:

Widen the domain of definition: In the example, we widen the domain of definition for the average function. The modification is made in one place only – in the definition of average itself.

Narrow the domain of application: In the example, we narrow the domain of application for the div function. Note that we narrow this domain only for the div application in average – other div applications may remain unsafe. Another alternative would be to narrow the domain of application for average, ensuring that [] is not passed as the argument. This alternative

Name	**Source**	**Core**	**Error**	**Pre**	**Sec**	**Mb**
Bernoulli*	35	652	5	11	4.1	0.8
Digits of E1*	44	377	3	8	0.3	0.6
Digits of E2	54	455	5	19	0.5	0.8
Exp3-8	29	163	0	0	0.1	0.1
Gen-Regexps*	41	776	1	1	0.3	0.4
Integrate	39	364	3	3	0.3	1.9
Paraffins*	91	1153	2	2	0.8	1.9
Primes	16	241	6	13	0.2	0.1
Queens	16	283	0	0	0.2	0.2
Rfib	9	100	0	0	0.1	1.7
Tak	12	155	0	0	0.1	0.1
Wheel Sieve 1*	37	570	7	10	7.5	0.9
Wheel Sieve 2*	45	636	2	2	0.3	0.6
X2n1	10	331	2	5	1.8	1.9
FiniteMap*	670	1829	13	17	1.6	1.0
HsColour*	823	5060	4	9	2.1	2.7

Name is the name of the checked program (a starred name indicates that changes were needed before safe pattern-matching could be verified); **Source** is the number of lines in the original source code; **Core** is the number of lines of first-order Core, *including all needed Prelude and library definitions*, just before analysis; **Error** is the number of calls to `error` (missing pattern cases); **Pre** is the number of functions which have a precondition which is not simply 'True'; **Sec** is the time taken for transformations and analysis; **Mb** is the maximum residency of Catch at garbage-collection time.

Table 6.1: Results of Catch checking

would require a deeper understanding of the flow of the program, requiring rather more work.

In the following sections, where modifications are required, we prefer to make the minimum number of changes. Consequently, we widen the domain of definition.

6.5.2 Nofib Benchmark Tests

The entire Nofib suite (Partain et al. 2008) is large. We concentrate on the 'Imaginary' section. These programs are all under a page of text, *excluding* any Prelude or library definitions used, and particularly stress list operations and numeric computations.

Results are given in Table 6.1. Only four programs contain no calls to error as all pattern-matches are exhaustive. Four programs use the list-indexing operator (!!), which requires the index to be non-negative and less than the length of the list; Catch can only prove this condition if the list is infinite. Eight programs include applications of either head or tail, most of which can be proven safe. Seven programs have incomplete patterns, often in a **where** binding and Catch performs well on these. Nine programs use division, with the precondition that the divisor must not be zero; most of these can be proven safe.

Three programs have preconditions on the main function, all of which state that the test parameter must be a natural number. In all cases the generated precondition is a necessary one – if the input violates the precondition then pattern-match failure will occur.

We now discuss general modifications required to allow Catch to begin checking the programs, followed by the six programs which required changes. We finish with the Digits of E2 program – a program with complex pattern matching that Catch is able to prove safe without modification.

Modifications for Checking Take a typical benchmark, Primes. The main function is:

```
main = do [arg] ← getArgs
          print $ primes !! (read arg)
```

The first unsafe pattern is [arg] ← getArgs, as getArgs is a primitive which may return any value. Additionally, if read fails to parse the value extracted from getArgs, it will evaluate to $\bot$. Instead, we check the revised program:

```
main = do args ← getArgs
          case map reads args of
               [[(x, s)]] | all isSpace s → print $ primes !! x
               _ → putStrLn "Bad command line"
```

Instead of crashing on malformed command line arguments, the modified program informs the user.

Bernoulli This program has one instance of tail (tail x). MP-constraints are unable to express that a list must be of at least length two, so Catch conservatively strengthens this to the condition that the list must be infinite – a condition that Bernoulli does not satisfy. One remedy is to replace tail (tail x) with drop 2 x. After this change, the program still has several non-exhaustive pattern matches, but all are proven safe.

Another approach would be to increase the power of MP-constraints. Currently MP-constraints store the root of a value separately from its recursive components. If they were modified to also store the first recursive component separately, then the Bernoulli example could be proved safe. The disadvantage of increasing the power of MP-constraints is that the checking process would take longer.

Digits of E1 This program contains the following equation:

```
ratTrans (a, b, c, d) xs |
  ((signum c ≡ signum d) || (abs c < abs d)) &&
  (c + d) ∗ q ⩽ a + b && (c + d) ∗ q + (c + d) > a + b
      = q : ratTrans (c, d, a − q ∗ c, b − q ∗ d) xs
  where q = b `div` d
```

Catch is able to prove that the division by d is only unsafe if both c and d are zero, but it is not able to prove that this invariant is maintained. Widening the domain of application of div allows the program to be proved safe.

As the safety of this program depends on quite deep results in number theory, it is no surprise that it is beyond the scope of an automatic checker such as Catch.

Gen-Regexps This program expects valid regular expressions as input. There are many ways to crash this program, including entering "", "[" or "<". One potential error comes from head ∘ lines, which can be replaced by takeWhile (≢ '\n'). Two potential errors take the form (a, _: b) = span f xs. At first glance this pattern definition is similar to the one in risers. But here the pattern is only safe if for one of the elements in the list xs, f returns True. The test f is actually (≢ '-'), and the only safe condition Catch can express is that xs is an infinite list. With the amendment (a, b) = safeSpan f xs, where safeSpan is defined by:

```
safeSpan p xs = (a, drop 1 b) where (a, b) = span p xs
```

Catch verifies pattern safety.

Wheel Sieve 1 This program defines a data type Wheel, and a function sieve:

```
data Wheel = Wheel Int [Int]
sieve :: [Wheel] → [Int] → [Int] → [Int]
```

The lists are infinite, and the integers are positive, but the program is too complex for Catch to infer these properties in full. To prove safety a variant of mod is required which does not raise division by zero and a pattern in notDivBy has to be completed. Even with these two modifications, Catch takes 7.5 seconds to check the other non-exhaustive pattern matches.

Wheel Sieve 2 This program has similar datatypes and invariants, but much greater complexity. Catch is able to prove very few of the necessary invariants. Only after widening the domain of definition in three places – replacing tail with drop 1, head with a version returning a default on the empty list, and mod with a safe variant – is Catch able to prove safety.

Paraffins Again the program can only be validated by Catch after modification. There are two reasons: laziness and arrays. Laziness allows the following odd-looking definition:

```
radical_generator n = radicals undefined
  where radicals unused = big_memory_computation
```

If radicals had a zero-arity definition it would be computed once and retained as long as there are references to it. To prevent this behaviour, a dummy argument (undefined) is passed. If the analysis was more lazy (as discussed in §6.3.4) then this example would succeed using Catch. As it is, simply changing undefined to () resolves the problem.

The Paraffins program uses the function array :: Ix a ⇒ (a, a) → [(a, b)] → Array a b which takes a list of index/value pairs and builds an array. The precondition on this function is that all indexes must be in the range specified.

This precondition is too complex for Catch, but simply using listArray, which takes a list of elements one after another, the program can be validated. Use of listArray actually makes the program shorter and more readable. The array indexing operator (!) is also troublesome. The precondition requires that the index is in the bounds given when the array was constructed, something Catch does not currently model.

Digits of E2 This program is quite complex, featuring a number of possible pattern-match errors. To illustrate, consider the following fragment:

```
carryPropagate base (d : ds) = ...
  where carryguess = d `div` base
        remainder = d `mod` base
        nextcarry : fraction = carryPropagate (base + 1) ds
```

There are four potential pattern-match errors in as many lines. Two of these are the calls to div and mod, both requiring base to be non-zero. A possibly more subtle pattern match error is the nextcarry : fraction left-hand side of the third line. Catch is able to prove that none of these pattern-matches fails. Now consider:

```
e = ("2."++) $
    tail ∘ concat $
    map (show ∘ head) $
    iterate (carryPropagate 2 ∘ map (10*) ∘ tail) $
    2 : [1, 1 ..]
```

Two uses of tail and one of head occur in quite complex functional pipelines. Catch is again able to prove that no pattern-match fails.

6.5.3 The FiniteMap library

The FiniteMap library for Haskell has been widely distributed for over 10 years. The library uses balanced binary trees, based on (Adams 1993). There are 14 non-exhaustive pattern matches.

The first challenge is that there is no main function. Catch uses all the exports from the library, and checks each of them as if it had main status.

Catch is able to prove that all but one of the non-exhaustive patterns are safe. The definition found unsafe has the form:

```
delFromFM (Branch key ...) del_key | del_key > key = ...
                                   | del_key < key = ...
                                   | del_key ≡ key = ...
```

At first glance the cases appear to be exhaustive. The law of trichotomy leads us to expect one of the guards to be true. However, the Haskell Ord class does not enforce this law. There is nothing to prevent an instance for a type with partially ordered values, some of which are incomparable. So Catch cannot verify the safety of delFromFM as defined as above.

The solution is to use the compare function which returns one of GT, EQ or LT. This approach has several advantages: (1) the code is free from non-exhaustive patterns; (2) the assumption of trichotomy is explicit in the return type; (3) the library is faster.

6.5.4 The HsColour Program

Artificial benchmarks are not necessarily intended to be fail-proof. But a real program, with real users, should *never* fail with a pattern-match error. We have taken the HsColour program[1] and analysed it using Catch. HsColour has 12 modules, is 5 years old and has had patches from 6 different people. We have contributed patches back to the author of HsColour, with the result that the development version can be proved free from pattern-match errors.

Catch required 4 small patches to the HsColour program before it could be verified free of pattern-match failures. Details of the checking process are given in Table 6.1. Of the 4 patches, 3 were genuine pattern-match errors which could be tripped by constructing unexpected input. The issues were: (1) read was called on a preferences file from the user, this could crash given a malformed preferences file; (2) by giving the document consisting of a single double quote character `"`, and passing the "-latex" flag, a crash occurred; (3) by giving the document `` (`) ``, namely open bracket, backtick, close bracket, and passing "-html -anchor" a crash occurred. The one patch which did not (as far as we are able to ascertain) fix a real bug could still be considered an improvement, and was minor in nature (a single line).

Examining the read error in more detail, by default Catch outputs the potential error message, and a list of potentially unsafe functions in a call

[1]http://www.cs.york.ac.uk/fp/darcs/hscolour/

stack:

```
Checking "Prelude.read: no parse"
Partial Prelude.read$252
Partial Language.Haskell.HsColour.Colourise.parseColourPrefs
...
Partial Main.main
```

We can see that parseColourPrefs calls read, which in turn calls error. The read function is specified to crash on incorrect parses, so the blame probably lies in parseColourPrefs. By examining this location in the source code we are able to diagnose and correct the problem. Catch optionally reports all the preconditions it has deduced, although in our experience problems can usually be fixed from source-position information alone.

6.5.5 The XMonad Program

XMonad (Stewart and Sjanssen 2007) is a window manager, which automatically manages the layout of program windows on the screen. The central module of XMonad contains a pure API, which is used to manipulate a data structure containing information regarding window layout. Catch has been run on this central module, several times, as XMonad has evolved. The XMonad API contains 36 exported functions, most of which are intended to be total. Within the implementation of these functions, there are a number of incomplete patterns and calls to partial functions.

When the Catch tool was first used, it detected six issues which were cause for concern – including unsafe uses of partial functions, API functions which contained incomplete pattern matches, and unnecessary assumptions about the Ord class. All these issues were subsequently fixed. The XMonad developers have said: "QuickCheck and Catch can be used to provide mechanical support for developing a clean, orthogonal API for a complex system" (Stewart and Sjanssen 2007).

In E-mail correspondence, the XMonad developers have summarised their experience using Catch as follows: "XMonad made heavy use of Catch in the development of its core data structures and logic. Catch caught several suspect error cases, and helped us improve robustness of the window manager core by weeding out partial functions. It helps encourage a healthy skepticism to partiality, and the quality of code was improved as a result.

We'd love to see a partiality checker integrated into GHC."

6.6 Related Work

6.6.1 Mistake Detectors

There has been a long history of writing tools to analyse programs to detect potential bugs, going back at least to the classic C Lint tool (Johnson 1978). In the functional arena there is the Dialyzer tool (Lindahl and Sagonas 2004) for Erlang (Virding et al. 1996). The aim is to have a static checker that works on unmodified code, with no additional annotations. However, a key difference is that in Dialyzer all warnings indicate a genuine problem that needs to be fixed. Because Erlang is a dynamically typed language, a large proportion of Dialyzer's warnings relate to mistakes a type checker would have detected.

The Catch tool tries to prove that `error` calls are unreachable. The Reach tool (Naylor and Runciman 2007) also checks for reachability, trying to find values which will cause a certain expression to be evaluated. Unlike Catch, if the Reach tool cannot find a way to reach an expression, this is no guarantee that the expression is indeed unreachable. So the tools are complementary: Reach can be used to find examples causing non-exhaustive patterns to fail, Catch can be used to prove there are no such examples.

6.6.2 Proving Incomplete Patterns Safe

Despite the seriousness of the problem of pattern matching, there are very few other tools for checking pattern-match safety. The closest other work we are aware of is ESC/Haskell (Xu 2006) and its successor Sound Haskell (Xu et al. 2007). The Sound Haskell approach requires the programmer to give explicit preconditions and contracts which the program obeys. Contracts have more expressive power than our constraints – one of the examples involves an invariant on an ordered list, something beyond Catch. But the programmer has more work to do. We eagerly await prototypes of either tool, to permit a full comparison against Catch.

6.6.3 Eliminating Incomplete Patterns

One way to guarantee that a program does not crash with an incomplete pattern is to ensure that all pattern matching is exhaustive. The GHC compiler (The GHC Team 2007) has an option flag to warn of any incomplete patterns. Unfortunately the Bugs section (12.2.1) of the manual notes that the checks are sometimes wrong, particularly with string patterns or guards, and that this part of the compiler "needs an overhaul really" (The GHC Team 2007). A more precise treatment of when warnings should be issued is given in Maranget (2007). These checks are only local: defining head will lead to a warning, even though the definition is correct; using head will not lead to a warning, even though it may raise a pattern-match error.

A more radical approach is to build exhaustive pattern matching into the design of the language, as part of a total programming system (Turner 2004). The Catch tool could perhaps allow the exhaustive pattern matching restriction to be lifted somewhat.

6.6.4 Type System Safety

One method for specifying properties about functional programs is to use the type system. This approach is taken in the tree automata work done on XML and XSLT (Tozawa 2001), which can be seen as an algebraic data type and a functional language. Another soft typing system with similarities is by Aiken and Murphy (1991), on the functional language FL. This system tries to assign a type to each function using a set of constructors, for example head takes the type Cons and not Nil.

Types can sometimes be used to explicitly encode invariants on data in functional languages. One approach is the use of *phantom types* (Fluet and Pucella 2002), for example a safe variant of tail can be written as in Figure 6.14. The List type is not exported, ensuring that all lists with a Cons tag are indeed non-empty. The types Cons and Unknown are phantom types – they exist only at the type level, and have no corresponding value.

Using GADTs (Peyton Jones et al. 2006), an encoding of lists can be written as in Figure 6.15. Notice that fromList requires a locally quantified type. The type-directed approach can be pushed much further with *dependent types*, which allow types to depend on values. There has been much work

```
data Cons
data Unknown

newtype List α τ = List [α]

cons :: α → [α] → List α Cons
cons a as = List (a : as)

nil :: List α Unknown
nil = List [ ]

fromList :: [α] → List α Unknown
fromList xs = List xs

safeTail :: List α Cons → [α]
safeTail (List (a : as)) = as
```

Figure 6.14: A safeTail function with Phantom types.

```
data ConsT α
data NilT

data List α τ where
  Cons :: α → List α τ → List α (ConsT τ)
  Nil   :: List α NilT

safeTail :: List α (ConsT τ) → List α τ
safeTail (Cons a b) = b

fromList :: [α] → (∀ τ • List α τ → β) → β
fromList [ ]        f = f Nil
fromList (x : xs) f = fromList xs (f ∘ Cons x)
```

Figure 6.15: A safeTail function using GADTs.

on dependent types, using undecidable type systems (McBride and McKinna 2004), using extensible kinds (Sheard 2004) and using type systems restricted to a decidable fragment (Xi and Pfenning 1999). The downside to all these type systems is that they require the programmer to make explicit annotations, and require the user to learn new techniques for computation.

Chapter 7

Conclusions

In this thesis we have presented a boilerplate reduction library (Uniplate), an optimiser (Supero), a defunctionalisation method (Firstify) and an analysis tool (Catch), all for the Haskell language. In this chapter we first describe some of the high-level contributions we have made in §7.1, give areas for future work in §7.2, then summarise our approach in §7.3.

7.1 Contributions

Specific technical contributions have been given in each chapter. In this section we instead focus on the higher-level contributions – the overall results that are of benefit to functional programmers.

7.1.1 Shorter Programs

Some of our work enables programmers to write shorter programs. In particular the Uniplate library defines a small set of operations to perform queries and transformations. We have illustrated by example that the boilerplate required in our system is less than in other systems (§3.7.1).

7.1.2 Faster Programs

Some of our work helps programs execute faster. Using Supero in conjunction with GHC we obtain an average runtime improvement of 16% for the imaginary section of the nofib suite. To quote Simon Peyton Jones, "an average runtime improvement of 10%, against the baseline of an already well-optimised compiler, is an excellent result" (Peyton Jones 2007). The Programming Language Shootout[1] has shown that low-level Haskell can compete with low-level imperative languages such as C. We hope that our optimiser will allow programs to be written in a high-level declarative style, yet still perform competitively.

We have also invested effort in optimising the Uniplate library. As a result we can express concise traversals without sacrificing speed (§3.7.2). In particular, we show a substantial speed up over the SYB library (Lämmel and Peyton Jones 2003).

We developed the Firstify tool for analysis, not performance. However, for many simple examples, the resultant program performs better than the original. If we restricted rules that reduce sharing, our defunctionalisation method may be appropriate for integration into an optimising compiler.

7.1.3 Safer Programs

The Catch tool allows programs to have non-exhaustive patterns, yet still have verifiable pattern-match safety. In practical use the Catch tool has found real bugs in real programs, which have subsequently been fixed (see §6.5.4). The XMonad developers found that using the Catch tool encouraged a safer style of programming, paying more attention to partial functions (see §6.5.5).

The Uniplate library also encourages a style of programming which can lead to fewer errors. By reducing the volume of code, particularly repetitive code, bugs become easier to spot.

7.2 Future Work

7.2.1 Robust and Widely Applicable Tools

We have implemented all the tools described in this thesis. The Uniplate library is already robust and used in real programs. The other tools serve

[1] `http://shootout.alioth.debian.org/`

more as prototypes, and have not seen sufficient real-world use to declare them production ready. With the exception of Uniplate, the tools are based around the core language from the Yhc compiler. Currently this core language is generated by the Yhc compiler, as described in §2.3. Yhc restricts our input programs to the Haskell 98 language. By making use of the GHC front end, we would be able to deal with many language extensions.

Supero, Firstify and Catch all operate on a whole program at a time, requiring sources for all function definitions. This requirement both increases the time required, and precludes the use of closed source libraries. We may be able to relax this requirement, precomputing partial results of libraries, or permitting some components of the program to be ignored. We already supply abstractions of IO functions for Catch, and this mechanism could be extended.

7.2.2 Uniplate

The use of boilerplate reduction strategies in Haskell is not yet ubiquitous, as we feel it should be. The ideas behind the Uniplate library have been used extensively, in projects including the Yhc compiler (Golubovsky et al. 2007), the Catch tool, the Reach tool (Naylor and Runciman 2007) and the Reduceron (Naylor and Runciman 2008). In previous versions of Catch there were over 100 Uniplate traversals.

There is scope for further speed improvements: for example, use of continuation passing style may eliminate tuple construction and consumption, and enhanced fusion may be able to eliminate some of the intermediate structures in the `uniplate` function. We have made extensive practical use of the Uniplate library, but there may be other traversals which deserve to be added.

Another area of future work, which others have already begun to explore, is the implementation of Uniplate in other languages. So far, we are aware of versions in ML[2] (Milner et al. 1997) and Curry[3] (Hanus et al. 1995). People have also proposed variations on Uniplate, including merging the Uniplate/Biplate distinction[4], and using `descend` as the underlying basis for

[2]http://mlton.org/cgi-bin/viewsvn.cgi/*checkout*/mltonlib/trunk/com/ssh/generic/unstable/public/value/uniplate.sig

[3]http://www.informatik.uni-kiel.de/~pakcs/lib/CDOC/Traversal.html

[4]http://www-ps.informatik.uni-kiel.de/~sebf/projects/traversal.html

the library[5].

7.2.3 Supero

Within Supero, there are three main areas for future work. Firstly, we would like to obtain results for larger programs, including all the remaining benchmarks in the nofib suite. Additional benchmarks will give further insight into the performance benefits that Supero provides.

Secondly, we would like to increase the runtime performance. Earlier versions of Supero (Mitchell and Runciman 2007b) managed to obtain substantial speed ups on benchmarks such as exp3_8. The Bernouilli benchmark is currently problematic. There is still scope for improvement.

Finally, we would like to increase compilation speed. The compilation times are tolerable for benchmarking and a final optimised release, but not for general use. We have described the major bottlenecks in §4.4.2, along with possible strategies for alleviating them.

7.2.4 Firstify

The Firstify library currently meets all the needs of the Catch tool. Within the algorithm, the use of a numeric termination bound in the homeomorphic embedding is regrettable, but practically motivated. We need further research to determine if such a numeric bound is necessary, or if other measures could be used.

7.2.5 Catch

The Catch tool has been applied to a range of benchmarks, and has shown promising results. However, there are obviously safe programs (for example Bernouilli in §6.5.2) which cannot be proven safe using MP-constraints. In addition to having insufficient power for some examples, MP-constraints also lack a normal form, requiring simplification rules. While MP-constraints are useful, we suspect there exist better constraint models which still fit into

[5] `http://tomschrijvers.blogspot.com/2007/11/extension-proposal-for-uniplate.html`

the Catch framework. One option would be to combine constraint models, allowing different constraint models to check different error calls.

The tests so far have not included any particularly large applications, such as the darcs program, a Haskell compiler, or even Catch itself. Further evaluation on large programs would give a better idea of what limits within Catch are most pressing. While we have released the Catch tool, it has not seen much use outside of our evaluation – end users are likely to have additional requests.

7.3 Concluding Remarks

Throughout this thesis we have been motivated by the idea of simplicity. We have attempted to reduce the complexity of our methods, both for implementation and for use. In particular, none of our tools requires any annotations to programs.

The Uniplate library restricts traversals to a uniformly typed value set, allowing the power of well-developed techniques for list processing such as list-comprehensions to be exploited. We feel this decision plays to Haskell's strengths, without being limiting in practice. Hopefully by not requiring complicated language features (particularly 'scary' types) we will allow a wider base of users to enjoy the benefits of boilerplate-free programming.

Our supercompiler is simple – the Core transformation is expressed in just 300 lines of Haskell. Yet it replicates many of the performance enhancements of GHC in a more general way. By simplifying the design, we are able to reduce the unintended interactions between optimisations, a problem that has been referred to as "swings and roundabouts" (Marlow and Peyton Jones 2006).

Many analysis methods, in fields such as strictness analysis and termination analysis, start out first-order and are gradually extended to work on a higher-order language. Defunctionalisation offers an alternative approach: instead of extending the analysis method, we transform the functional values away. The analysis method can remain simple, and still work on all programs.

For the Catch tool we have made two decisions that significantly simplify the design: (1) the target of analysis is a very small, first-order core language; (2)

there are finitely many value-set-defining constraints per type. Decision (1) allows for a much simpler analysis method, without the added complexity of higher-order programs. Decision (2) inevitably limits the expressive power of constraints; yet it does not prevent the expression of uniform recursive constraints on the deep structure of values, as in MP-constraints.

Functional programs are well suited to analysis and transformation. In this thesis we have presented a number of techniques, which have been refined in response to practical experiments. We hope that the ideas presented will be of real benefit to functional programmers.

Appendix A

Soundness of Pattern-Match Analysis

This appendix shows that the algorithms and constraint languages presented in Chapter 6 are sound – if Catch informs the user that a program is safe, then that program is guaranteed not to call error. §A.1 describes the style of proof and §A.2 defines an evaluator for expressions. §A.3 states the soundness theorem, along with necessary side conditions. §A.4 gives two lemmas that constraint languages must satisfy, and shows they hold for BP-constraints and MP-constraints. §A.5 gives eight lemmas, mainly about the constraint operations. §A.6 gives the proof of soundness and §A.7 discusses the results.

A.1 Proof-Style and Notation

Proofs in this appendix are detailed outlines based on equational reasoning. They make use of induction, application of equational laws, case analysis and inlining of function definitions. When performing case analysis we ignore the possibility of $\perp$ as a Haskell value, which is valid provided all expressions within the proof do not result in $\perp$. The proofs are structured as a series of rewrites, either preserving equality (marked with $\equiv$), or implying the previous statement (marked with $\Leftarrow$). Some expressions may refer to locally bound definitions, which we do not show until necessary. All the properties are boolean valued expressions, typically implications.

```
data Value = Value CtorName [Value]
           | Bottom

data Expr = Make CtorName [Expr]   -- Constructor application
          | Call FuncName [Expr]   -- Function application
          | Var VarName            -- Function variable
          | Sel  Expr Selector     -- Case alternative variable
          | Case Expr [Alt]        -- Case expression

data Alt = Alt CtorName [VarName] Expr

eval :: Expr → Value
eval (Sel x (c, i)  )  | c ≡ c′ = xs !! i
  where Value c′ xs = eval x
eval (Make c xs   ) = Value c (map eval xs)
eval (Call f   xs  )  | f ≡ "error" = Bottom
                      | otherwise     = eval $ body f / (args f, xs)
eval (Case x as    ) = case eval x of
  Value c xs → head [eval y | Alt c′ vs y ← as, c ≡ c′]
  Bottom → Bottom
```

Figure A.1: Evaluator for expressions.

In order to reduce the size of some of the intermediate expressions, we have replaced one side of an implication with either LHS or RHS, designating either the left-hand side or right-hand side. We also make heavy use of the function composition and function application operators, namely (∘) and ($). These can be read with the translations:

$(f \circ g)\ x = f\ \$\ g\ x = f\ (g\ x)$

A.2 Evaluator

We start with an evaluator for a Core expression language defined in Figure A.1. The value Bottom corresponds to program failure, *not* divergence. The Expr data type represents the first-order Core language described in §6.3.1. We have introduced Sel, which represents variables bound in case alternatives. For the algorithms presented previously, the Sel expression contains the information returned by the var function. While evaluating a Sel expression we know that we are beneath a Case on the same expression x, and that x evaluates to the constructor mentioned in the selector. The Alt

```
isBottom :: Value → Bool
isBottom Bottom = True
isBottom (Value c xs) = any isBottom xs

valCtor :: Value → Maybe CtorName
valCtor (Value c xs) = Just c
valCtor Bottom = Nothing

satE′ :: Prop (Sat Expr) → Bool
satE′ = tautP satE

satE :: Sat Expr → Bool
satE (Sat x k) = sat (Sat (eval x) k)

sat′ :: Prop (Sat Value) → Bool
sat′ = tautP sat

sat :: Sat Value → Bool
sat (Sat Bottom       k) = True
sat (Sat (Value c xs) k) = sat′ $ (c ◁ k)/([0 ..], xs)
```

Figure A.2: Auxiliary functions.

constructor has a list of variables which are unnecessary because selectors are used instead of local variable names – but they are retained to allow direct use of the algorithms from Chapter 6.

There is no case in eval for Var, and the behaviour of eval with free variables is undefined. The Call equation will replace any free variables in the body of a function with the supplied arguments.

We make use of (/), which we have redefined as a substitution operator – we write x / (vs, ys) to denote replacing the free variables vs in x with ys. We use (/) instead of substP in the proofs, where substP (zip vs ys) x = x/(vs, ys).

We also make use of a number of auxiliaries defined in Figure A.2. The sat function tests whether a value satisfies a constraint, using the (◁) operator from the constraint language. The satE function tests whether the result of evaluating an expression satisfies a constraint. The sat′ and satE′ functions operate over a proposition.

A.3 Soundness Theorem

We wish to prove that our analysis is sound, as previously discussed in §6.3.7. Our analysis is sound if for any expression e, provided the precondition of e is calculated to be true, then the evaluation of e will not result in $\perp$. Using the definitions from §A.2 we can express this theorem as:

$\mathsf{satE}' \ \$ \ \mathsf{pre\ e} \Rightarrow \mathsf{not} \ \$ \ \mathsf{isBottom} \ \$ \ \mathsf{eval\ e}$

In order to evaluate an expression to normal form, it is necessary for the expression to be closed and for evaluation of the expression to terminate. Therefore we prove that provided the expression e can be evaluated to normal form, and that both e and the program under analysis are first-order Core as described in §6.3.1, the above theorem is true. The above restrictions allow us to assume that all constructors have the correct arity, and that there are no free variables. We will prove this result in A.6.

A.4 Constraint Lemmas

We shall need the following two lemmas about each constraint language:

Lemma C1

$\mathsf{sat}' \ (\mathsf{Value\ c\ xs} \lessdot \mathsf{cs}) \Rightarrow \mathsf{c} \in \mathsf{cs}$

Lemma C2

$\mathsf{sat} \ \$ \ \mathsf{Sat} \ (\mathsf{Value\ c\ xs}) \ ((\mathsf{c}, \mathsf{i}) \rhd \mathsf{k}) \Rightarrow \mathsf{sat} \ \$ \ \mathsf{Sat} \ (\mathsf{xs} \ !! \ \mathsf{i}) \ \mathsf{k}$

Both lemmas are given in §6.3.6, as properties. These properties require the ($\lessdot$) and ($\rhd$) operators to be consistent with sat. The sat function is defined in terms of ($\lhd$), and therefore all three operators must be consistent with each other. We prove them for the BP-constraint and MP-constraint systems in §A.4.1 and §A.4.2. Since RE-constraints do not scale sufficiently, we do not recommend their use, and have not attempted to prove these lemmas for them.

```
data Constraint = Any
                | Con CtorName [Constraint]

a⋖xs = orP [lit (a `Sat` anys x) | x ← xs]
  where anys x = Con x (replicate (arity x) Any)

(c, i) ▷ k = Con c  [ if i ≡ j then k else Any
                    | j ← [0 . . arity c − 1]]

c ◁ Any          = true
c ◁ Con c2 xs    = bool (c2 ≡ c) ∧ andP (map lit (zipWith Sat [0 . .] xs))
```

Figure A.3: BP-Constraint operations.

A.4.1 BP-Constraint Lemmas

Since BP-constraints are presented piecemeal throughout §6.3, we present all the relevant definitions in Figure A.3.

Lemma C1 (BP)

```
sat′ (Value c xs⋖cs) ⇒ c ∈ cs
≡ {inline (⋖)}
sat′ $ orP [lit (Value c xs `Sat` anys c′) | c′ ← cs] ⇒ c ∈ cs
≡ {inline sat′}
any (λc′ → sat $ Value c xs `Sat` anys c′) cs ⇒ c ∈ cs
≡ {inline sat}
any (λc′ → sat′ $ (c ◁ anys c′)/([0 . .], xs)) cs ⇒ c ∈ cs
≡ {inline anys}
any (λc′ → sat′ $ (c ◁ Con c′ (replicate (arity c′) Any))/
  ([0 . .], xs)) cs ⇒ c ∈ cs
≡ {inline ◁}
any (λc′ → sat′ $ (bool (c′ ≡ c) ∧
  andP (map lit (zipWith Sat [0 . .] (replicate (arity c′) Any))))
  /([0 . .], xs)) cs ⇒ c ∈ cs
≡ {inline (/)}
any (λc′ → sat′ $ (bool (c′ ≡ c) ∧
  andP (map lit (zipWith Sat [0 . .] (replicate (arity c′) Any)
  /([0 . .], xs))))) cs ⇒ c ∈ cs
≡ {inline sat′}
any (λc′ → (c′ ≡ c) && all sat (zipWith Sat [0 . .] (replicate (arity c′) Any)
  /([0 . .], xs))) cs ⇒ c ∈ cs
⇐ {weaken implication}
any (λc′ → c′ ≡ c) cs ⇒ c ∈ cs
```

$\equiv \{\textit{simplify}\}$
$\mathsf{any}\ (\equiv \mathsf{c})\ \mathsf{cs} \Rightarrow \mathsf{c} \in \mathsf{cs}$
$\equiv \{\textit{definition of } \mathsf{elem}\}$
$\mathsf{c} \in \mathsf{cs} \Rightarrow \mathsf{c} \in \mathsf{cs}$
$\equiv \{\textit{tautology}\}$
True

Lemma C2 (BP)

$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value\ c\ xs})\ ((\mathsf{c}, \mathsf{i}) \rhd \mathsf{k}) \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ \mathsf{k}$

Case: $\mathsf{k} = \mathsf{Any}$

$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value\ c\ xs})\ ((\mathsf{c}, \mathsf{i}) \rhd \mathsf{k}) \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ \mathsf{k}$
$\equiv \{\mathsf{k} = \mathsf{Any}\}$
$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value\ c\ xs})\ ((\mathsf{c}, \mathsf{i}) \rhd \mathsf{Any}) \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ \mathsf{Any}$
$\equiv \{\textit{inline } \mathsf{sat}\ \textit{on RHS}\}$
$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value\ c\ xs})\ ((\mathsf{c}, \mathsf{i}) \rhd \mathsf{Any}) \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ \mathsf{Any}$

Case: $\mathsf{k} = \mathsf{Any}$ **;** $\mathsf{xs}\ !!\ \mathsf{i} = \mathsf{Bottom}$

$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value\ c\ xs})\ ((\mathsf{c}, \mathsf{i}) \rhd \mathsf{Any}) \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ \mathsf{Any}$
$\equiv \{\mathsf{xs}\ !!\ \mathsf{i} = \mathsf{Bottom}\}$
$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value\ c\ xs})\ ((\mathsf{c}, \mathsf{i}) \rhd \mathsf{Any}) \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat\ Bottom\ Any}$
$\equiv \{\textit{inline } \mathsf{sat}\ \textit{on RHS}\}$
$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value\ c\ xs})\ ((\mathsf{c}, \mathsf{i}) \rhd \mathsf{Any}) \Rightarrow \mathsf{True}$
$\equiv \{\textit{implication}\}$
True

Case: $\mathsf{k} = \mathsf{Any}$ **;** $\mathsf{xs}\ !!\ \mathsf{i} = \mathsf{Value\ c'\ ys}$

$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value\ c\ xs})\ ((\mathsf{c}, \mathsf{i}) \rhd \mathsf{Any}) \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ \mathsf{Any}$
$\equiv \{\mathsf{xs}\ !!\ \mathsf{i} = \mathsf{Value\ c'\ ys}\}$
$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value\ c\ xs})\ ((\mathsf{c}, \mathsf{i}) \rhd \mathsf{Any}) \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value\ c'\ ys})\ \mathsf{Any}$
$\equiv \{\textit{inline } \mathsf{sat}\ \textit{on RHS}\}$
$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value\ c\ xs})\ ((\mathsf{c}, \mathsf{i}) \rhd \mathsf{Any}) \Rightarrow \mathsf{sat'}\ \$\ (\mathsf{c'} \lhd \mathsf{Any})/([0\,..], \mathsf{ys})$
$\equiv \{\textit{inline } \lhd\}$
$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value\ c\ xs})\ ((\mathsf{c}, \mathsf{i}) \rhd \mathsf{Any}) \Rightarrow \mathsf{sat'}\ \$\ \mathsf{true}/([0\,..], \mathsf{ys})$
$\equiv \{\textit{inline } (/)\ \textit{on RHS}\}$

$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}\ \mathsf{xs})\ ((\mathsf{c},\mathsf{i}) \rhd \mathsf{Any}) \Rightarrow \mathsf{sat'}\ \mathsf{true}$
$\equiv \{\textit{inline}\ \mathsf{sat'}\ \textit{on RHS}\}$
$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}\ \mathsf{xs})\ ((\mathsf{c},\mathsf{i}) \rhd \mathsf{Any}) \Rightarrow \mathsf{True}$
$\equiv \{\textit{implication}\}$
True

Case: $\mathsf{k} = \mathsf{Con}\ \mathsf{c'}\ \mathsf{ys}$

$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}\ \mathsf{xs})\ ((\mathsf{c},\mathsf{i}) \rhd \mathsf{k}) \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ \mathsf{k}$
$\equiv \{\mathsf{k} = \mathsf{Con}\ \mathsf{c'}\ \mathsf{ys}\}$
$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}\ \mathsf{xs})\ ((\mathsf{c},\mathsf{i}) \rhd \mathsf{Con}\ \mathsf{c'}\ \mathsf{ys}) \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ (\mathsf{Con}\ \mathsf{c'}\ \mathsf{ys})$
$\equiv \{\textit{inline}\ \mathsf{sat}\ \textit{on LHS}\}$
$\mathsf{sat'}\ \$\ (\mathsf{c} \lhd ((\mathsf{c},\mathsf{i}) \rhd \mathsf{Con}\ \mathsf{c'}\ \mathsf{ys}))/([0\mathinner{..}],\mathsf{xs}) \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{inline}\ \rhd\}$
$\mathsf{sat'}\ \$\ (\mathsf{c} \lhd (\mathsf{Con}\ \mathsf{c}\ [\mathbf{if}\ \mathsf{i} \equiv \mathsf{j}\ \mathbf{then}\ (\mathsf{Con}\ \mathsf{c'}\ \mathsf{ys})\ \mathbf{else}\ \mathsf{Any} \mid \mathsf{j} \leftarrow [0\mathinner{..}\mathsf{arity}\ \mathsf{c} - 1]]))$
$\quad /([0\mathinner{..}],\mathsf{xs}) \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{inline}\ \lhd\}$
$\mathsf{sat'}\ \$\ (\mathsf{bool}\ (\mathsf{c} \equiv \mathsf{c})\ \wedge\ \mathsf{andP}\ (\mathsf{map}\ \mathsf{lit}\ (\mathsf{zipWith}\ \mathsf{Sat}\ [0\mathinner{..}]$
$\quad [\mathbf{if}\ \mathsf{i} \equiv \mathsf{j}\ \mathbf{then}\ (\mathsf{Con}\ \mathsf{c'}\ \mathsf{ys})\ \mathbf{else}\ \mathsf{Any} \mid \mathsf{j} \leftarrow [0\mathinner{..}\mathsf{arity}\ \mathsf{c} - 1]])))$
$\quad /([0\mathinner{..}],\mathsf{xs}) \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{inline}\ \mathsf{c} \equiv \mathsf{c}\}$
$\mathsf{sat'}\ \$\ (\mathsf{true}\ \wedge\ \mathsf{andP}\ (\mathsf{map}\ \mathsf{lit}\ (\mathsf{zipWith}\ \mathsf{Sat}\ [0\mathinner{..}]$
$\quad [\mathbf{if}\ \mathsf{i} \equiv \mathsf{j}\ \mathbf{then}\ (\mathsf{Con}\ \mathsf{c'}\ \mathsf{ys})\ \mathbf{else}\ \mathsf{Any} \mid \mathsf{j} \leftarrow [0\mathinner{..}\mathsf{arity}\ \mathsf{c} - 1]])))$
$\quad /([0\mathinner{..}],\mathsf{xs}) \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{inline}\ (\wedge)\}$
$\mathsf{sat'}\ \$\ (\mathsf{andP}\ (\mathsf{map}\ \mathsf{lit}\ (\mathsf{zipWith}\ \mathsf{Sat}\ [0\mathinner{..}]$
$\quad [\mathbf{if}\ \mathsf{i} \equiv \mathsf{j}\ \mathbf{then}\ (\mathsf{Con}\ \mathsf{c'}\ \mathsf{ys})\ \mathbf{else}\ \mathsf{Any} \mid \mathsf{j} \leftarrow [0\mathinner{..}\mathsf{arity}\ \mathsf{c} - 1]])))$
$\quad /([0\mathinner{..}],\mathsf{xs}) \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{inline}\ (/)\}$
$\mathsf{sat'}\ \$\ \mathsf{andP}\ \$\ \mathsf{map}\ \mathsf{lit}\ \$\ \mathsf{zipWith}\ \mathsf{Sat}\ [0\mathinner{..}]$
$\quad [\mathbf{if}\ \mathsf{i} \equiv \mathsf{j}\ \mathbf{then}\ (\mathsf{Con}\ \mathsf{c'}\ \mathsf{ys})\ \mathbf{else}\ \mathsf{Any} \mid \mathsf{j} \leftarrow [0\mathinner{..}\mathsf{arity}\ \mathsf{c} - 1]]$
$\quad /([0\mathinner{..}],\mathsf{xs}) \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{inline}\ \mathsf{sat'}\}$
$\mathsf{all}\ \mathsf{sat}\ \$\ \mathsf{zipWith}\ \mathsf{Sat}\ [0\mathinner{..}]$
$\quad [\mathbf{if}\ \mathsf{i} \equiv \mathsf{j}\ \mathbf{then}\ (\mathsf{Con}\ \mathsf{c'}\ \mathsf{ys})\ \mathbf{else}\ \mathsf{Any} \mid \mathsf{j} \leftarrow [0\mathinner{..}\mathsf{arity}\ \mathsf{c} - 1]]$
$\quad /([0\mathinner{..}],\mathsf{xs}) \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{inline}\ (/)\}$
$\mathsf{all}\ \mathsf{sat}\ \$\ \mathsf{zipWith}\ \mathsf{Sat}\ \mathsf{xs}$
$\quad [\mathbf{if}\ \mathsf{i} \equiv \mathsf{j}\ \mathbf{then}\ (\mathsf{Con}\ \mathsf{c'}\ \mathsf{ys})\ \mathbf{else}\ \mathsf{Any} \mid \mathsf{j} \leftarrow [0\mathinner{..}\mathsf{arity}\ \mathsf{c} - 1]] \Rightarrow \mathrm{RHS}$
$\Leftarrow \{\textit{weaken LHS}\}$
$\mathsf{sat}\ \$\ \mathsf{zipWith}\ \mathsf{Sat}\ \mathsf{xs}$
$\quad [\mathbf{if}\ \mathsf{i} \equiv \mathsf{j}\ \mathbf{then}\ (\mathsf{Con}\ \mathsf{c'}\ \mathsf{ys})\ \mathbf{else}\ \mathsf{Any} \mid \mathsf{j} \leftarrow [0\mathinner{..}\mathsf{arity}\ \mathsf{c} - 1]]\ !!\ \mathsf{i} \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{inline}\ (!!)\}$
$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ (\mathsf{Con}\ \mathsf{c'}\ \mathsf{ys}) \Rightarrow \mathrm{RHS}$

$\equiv \{\textit{restore RHS}\}$
$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ (\mathsf{Con}\ \mathsf{c}'\ \mathsf{ys}) \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ (\mathsf{Con}\ \mathsf{c}'\ \mathsf{ys})$
$\equiv \{\textit{tautology}\}$
True

A.4.2 MP-Constraint Lemmas

The proofs in this section appeal to definitions in Figure 6.13. To perform some of these proofs, we will rely on two auxiliary lemmas about MP-constraints.

Lemma MP1

$\mathsf{sat}\ (\mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}\ \mathsf{xs})\ \mathsf{ks}) \equiv \mathsf{any}\ (\lambda \mathsf{k} \rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}\ \mathsf{xs})\ [\mathsf{k}])\ \mathsf{ks}$

We argue as follows:

$\mathsf{sat}\ (\mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}\ \mathsf{xs})\ \mathsf{ks}) \equiv \mathsf{any}\ (\lambda \mathsf{k} \rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}\ \mathsf{xs})\ [\mathsf{k}])\ \mathsf{ks}$
$\equiv \{\textit{inline}\ \mathsf{sat}\}$
$\mathsf{sat}'\ ((\mathsf{c} \lhd \mathsf{ks})/([0\,..], \mathsf{xs})) \equiv \mathrm{RHS}$
$\equiv \{\textit{inline}\ \lhd\}$
$\mathsf{sat}'\ (\mathsf{orP}\ (\mathsf{map}\ \mathsf{f}\ \mathsf{ks})/([0\,..], \mathsf{xs})) \equiv \mathrm{RHS}$
$\equiv \{\textit{inline}\ (/)\}$
$\mathsf{sat}'\ (\mathsf{orP}\ (\mathsf{map}\ \mathsf{f}\ \mathsf{ks}/([0\,..], \mathsf{xs}))) \equiv \mathrm{RHS}$
$\equiv \{\textit{inline}\ \mathsf{sat}'\}$
$\mathsf{any}\ \mathsf{sat}'\ (\mathsf{map}\ \mathsf{f}\ \mathsf{ks}/([0\,..], \mathsf{xs})) \equiv \mathrm{RHS}$
$\equiv \{\textit{inline}\ (/)\}$
$\mathsf{any}\ \mathsf{sat}'\ (\mathsf{map}\ ((/([0\,..], \mathsf{xs})) \circ \mathsf{f})\ \mathsf{ks}) \equiv \mathrm{RHS}$
$\equiv \{\textit{combine}\ \mathsf{any}\ \textit{and}\ \mathsf{map}\}$
$\mathsf{any}\ (\mathsf{sat}' \circ (/([0\,..], \mathsf{xs})) \circ \mathsf{f})\ \mathsf{ks} \equiv \mathrm{RHS}$
$\equiv \{\textit{insert}\ \mathsf{k}\}$
$\mathsf{any}\ (\lambda \mathsf{k} \rightarrow \mathsf{sat}'\ \$\ \mathsf{f}\ \mathsf{k}/([0\,..], \mathsf{xs}))\ \mathsf{ks} \equiv \mathrm{RHS}$
$\equiv \{\textit{insert RHS}\}$
$\mathsf{any}\ (\lambda \mathsf{k} \rightarrow \mathsf{sat}'\ \$\ \mathsf{f}\ \mathsf{k}/([0\,..], \mathsf{xs}))\ \mathsf{ks} \equiv \mathsf{any}\ (\lambda \mathsf{k} \rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}\ \mathsf{xs})\ [\mathsf{k}])\ \mathsf{ks}$
$\equiv \{\textit{unwrap common parts}\}$
$\mathsf{sat}'\ (\mathsf{f}\ \mathsf{k}/([0\,..], \mathsf{xs})) \equiv \mathsf{sat}\ (\mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}\ \mathsf{xs})\ [\mathsf{k}])$
$\equiv \{\textit{inline}\ \mathsf{sat}\}$
$\mathrm{LHS} \equiv \mathsf{sat}'\ ((\mathsf{c} \lhd [\mathsf{k}])/([0\,..], \mathsf{xs}))$
$\equiv \{\textit{inline}\ \lhd\}$
$\mathrm{LHS} \equiv \mathsf{sat}'\ (\mathsf{orP}\ (\mathsf{map}\ \mathsf{f}\ [\mathsf{k}])/([0\,..], \mathsf{xs}))$
$\equiv \{\textit{inline}\ \mathsf{map}\}$
$\mathrm{LHS} \equiv \mathsf{sat}'\ (\mathsf{orP}\ [\mathsf{f}\ \mathsf{k}]/([0\,..], \mathsf{xs}))$

$\equiv$ {*inline* orP}
sat′ (f k/([0 . .], xs)) $\equiv$ sat′ (f k/([0 . .], xs))
$\equiv$ {*tautology*}
True

Lemma MP2

sat \$ Sat (Value c xs) [merge ms_1 ms_2 $\star$ merge ms_1 ms_2] $\Rightarrow$
   sat \$ Sat (Value c xs) [ms_1 $\star$ ms_2]

We have *not* proved this lemma, as we suspect the proof is very long. Instead we have used Lazy SmallCheck (Lindblad et al. 2007) to test the property. Lazy SmallCheck exhaustively tests properties up to some depth of input values. Here is the property we have tested, along with the invariants on values:

```
prop :: (Value, [Pattern], [Pattern]) → Bool
prop (v, ms1, ms2) =
   validValue v && validPatterns ms1 && validPatterns ms2 &&
   sat (Sat v [ms ⋆ ms]) ⇒ sat (Sat v [ms1 ⋆ ms2])
   where ms = merge ms1 ms2

validValue Bottom       = True
validValue (Value c xs) = arity c ≡ length xs && all validValue xs

validVal Any           = True
validVal (ms1 ⋆ ms2) = validPatterns ms1 && validPatterns ms2

validPatterns = all validPattern
validPattern (Pattern c xs) = fields c ≡ length xs && all validVal xs

fields c = length [isRec (c, i) | i ← [0 . . arity c − 1]]
```

We check that values and patterns are well-formed using validValue and validPatterns. These functions both check that the arity of constructors are correct, and that patterns have an appropriate number of non-recursive fields. We have defined depth so that the length-constrained list structure present in both values and patterns *does not* count towards the depth of a structure. For example, the following is a Value structure of depth 3:

```
Value "(,)"
   [Value ":"
      [Value "True" []
```

```
    , Value "[]" []]
  , Value "Nothing" []]
```

Testing all values and patterns up to depth 4 (446,105,404 tests) no counterexample is found. These tests represent 692,363,920,494,602 possible inputs. We do not believe it is feasible to test this property at a greater depth, but depth 4 gives us reasonable confidence that the property is indeed true.

Lemma C1 (MP)

sat′ (Value c xs⋖cs) ⇒ c ∈ cs
≡ {*inline* (⋖)}
sat′ \$ lit \$ Sat (Value c xs)
  [map complete cs ⋆ map complete (ctors (head cs)) | not (null cs)] ⇒ c ∈ cs

Case: cs = [ ]

sat′ \$ lit \$ Sat (Value c xs)
  [map complete cs ⋆ map complete (ctors (head cs)) | not (null cs)] ⇒ c ∈ cs
≡ {cs = [ ]}
sat′ \$ lit \$ Sat (Value c xs)
  [map complete [ ] ⋆ map complete (ctors (head [ ])) | not (null [ ])] ⇒ c ∈ [ ]
≡ {*inline* null}
sat′ \$ lit \$ Sat (Value c xs)
  [map complete [ ] ⋆ map complete (ctors (head [ ])) | not True] ⇒ c ∈ [ ]
≡ {*inline* not}
sat′ \$ lit \$ Sat (Value c xs)
  [map complete [ ] ⋆ map complete (ctors (head [ ])) | False] ⇒ c ∈ [ ]
≡ {*reduce list comprehension*}
sat′ \$ lit \$ Sat (Value c xs) [ ] ⇒ c ∈ [ ]
≡ {*inline* elem}
sat′ \$ lit \$ Sat (Value c xs) [ ] ⇒ False
≡ {*inline* sat′}
sat \$ Sat (Value c xs) [ ] ⇒ False
≡ {*Lemma MP1*}
any (λk → sat \$ Sat (Value c xs) [k]) [ ] ⇒ False
≡ {*inline* any}
False ⇒ False
≡ {*tautology*}
True

Case: $\mathsf{cs} \not\equiv [\,]$

$\mathsf{sat}' \,\$\, \mathsf{lit} \,\$\, \mathsf{Sat}\ (\mathsf{Value\ c\ xs})$
$\quad [\mathsf{map\ complete\ cs} \star \mathsf{map\ complete\ (ctors\ (head\ cs))} \mid \mathsf{not\ (null\ cs)}] \Rightarrow \mathsf{c} \in \mathsf{cs}$
$\equiv \{\mathsf{null\ cs} \equiv \mathsf{False}\}$
$\mathsf{sat}' \,\$\, \mathsf{lit} \,\$\, \mathsf{Sat}\ (\mathsf{Value\ c\ xs})$
$\quad [\mathsf{map\ complete\ cs} \star \mathsf{map\ complete\ (ctors\ (head\ cs))} \mid \mathsf{not\ False}] \Rightarrow \mathsf{c} \in \mathsf{cs}$
$\equiv \{\textit{inline}\ \mathsf{not}\}$
$\mathsf{sat}' \,\$\, \mathsf{lit} \,\$\, \mathsf{Sat}\ (\mathsf{Value\ c\ xs})$
$\quad [\mathsf{map\ complete\ cs} \star \mathsf{map\ complete\ (ctors\ (head\ cs))} \mid \mathsf{True}] \Rightarrow \mathsf{c} \in \mathsf{cs}$
$\equiv \{\textit{simplify list comprehension}\}$
$\mathsf{sat}' \,\$\, \mathsf{lit} \,\$\, \mathsf{Sat}\ (\mathsf{Value\ c\ xs})$
$\quad [\mathsf{map\ complete\ cs} \star \mathsf{map\ complete\ (ctors\ (head\ cs))}] \Rightarrow \mathsf{c} \in \mathsf{cs}$
$\equiv \{\textit{inline}\ \mathsf{sat}'\}$
$\mathsf{sat} \,\$\, \mathsf{Sat}\ (\mathsf{Value\ c\ xs})$
$\quad [\mathsf{map\ complete\ cs} \star \mathsf{map\ complete\ (ctors\ (head\ cs))}] \Rightarrow \mathsf{c} \in \mathsf{cs}$
$\equiv \{\textit{inline}\ \mathsf{sat}\}$
$\mathsf{sat}' \,\$\, (\mathsf{c} \lhd [\mathsf{map\ complete\ cs} \star \mathsf{map\ complete\ (ctors\ (head\ cs))}])$
$\quad /([0\mathinner{..}], \mathsf{xs}) \Rightarrow \mathsf{c} \in \mathsf{cs}$
$\equiv \{\textit{inline}\ \lhd\}$
$\mathsf{sat}' \,\$\, (\mathsf{orP} \,\$\, \mathsf{map\ f}\ [\mathsf{map\ complete\ cs} \star \mathsf{map\ complete\ (ctors\ (head\ cs))}])$
$\quad /([0\mathinner{..}], \mathsf{xs}) \Rightarrow \mathsf{c} \in \mathsf{cs}$
$\equiv \{\textit{inline}\ \mathsf{map}\}$
$\mathsf{sat}' \,\$\, (\mathsf{orP}\ [\mathsf{f} \,\$\, \mathsf{map\ complete\ cs} \star \mathsf{map\ complete\ (ctors\ (head\ cs))}])$
$\quad /([0\mathinner{..}], \mathsf{xs}) \Rightarrow \mathsf{c} \in \mathsf{cs}$
$\equiv \{\textit{inline}\ (\vee)\}$
$\mathsf{sat}' \,\$\, (\mathsf{f} \,\$\, \mathsf{map\ complete\ cs} \star \mathsf{map\ complete\ (ctors\ (head\ cs))})$
$\quad /([0\mathinner{..}], \mathsf{xs}) \Rightarrow \mathsf{c} \in \mathsf{cs}$
$\equiv \{\textit{inline}\ \mathsf{f}\}$
$\mathsf{sat}' \,\$\, \mathsf{orP}\ [\mathsf{andP} \,\$\, \mathsf{map\ lit} \,\$\, \mathsf{g\ vs_1} \mid \mathsf{Pattern\ c_1\ vs_1} \leftarrow \mathsf{map\ complete\ cs}, \mathsf{c_1} \equiv \mathsf{c}]$
$\quad /([0\mathinner{..}], \mathsf{xs}) \Rightarrow \mathsf{c} \in \mathsf{cs}$
$\equiv \{\textit{inline}\ (/)\}$
$\mathsf{sat}' \,\$\, \mathsf{orP}\ [\mathsf{andP} \,\$\, \mathsf{map\ lit} \,\$\, \mathsf{g\ vs_1}/([0\mathinner{..}], \mathsf{xs}) \mid$
$\quad \mathsf{Pattern\ c_1\ vs_1} \leftarrow \mathsf{map\ complete\ cs}, \mathsf{c_1} \equiv \mathsf{c}] \Rightarrow \mathsf{c} \in \mathsf{cs}$
$\equiv \{\textit{inline}\ \mathsf{sat}'\}$
$\mathsf{or}\ [\mathsf{sat}' \,\$\, \mathsf{andP} \,\$\, \mathsf{map\ lit} \,\$\, \mathsf{g\ vs_1}/([0\mathinner{..}], \mathsf{xs}) \mid$
$\quad \mathsf{Pattern\ c_1\ vs_1} \leftarrow \mathsf{map\ complete\ cs}, \mathsf{c_1} \equiv \mathsf{c}] \Rightarrow \mathsf{c} \in \mathsf{cs}$
$\equiv \{\textit{move the guard}\}$
$\mathsf{or}\ [\mathsf{c_1} \equiv \mathsf{c}\ \&\&\ \mathsf{sat}'\ (\mathsf{andP} \,\$\, \mathsf{map\ lit} \,\$\, \mathsf{g\ vs_1}/([0\mathinner{..}], \mathsf{xs})) \mid$
$\quad \mathsf{Pattern\ c_1\ vs_1} \leftarrow \mathsf{map\ complete\ cs}] \Rightarrow \mathsf{c} \in \mathsf{cs}$
$\equiv \{\textit{remove the list comprehension}\}$
$\mathsf{any}\ (\lambda(\mathsf{Pattern\ c_1\ vs_1}) \rightarrow \mathsf{c_1} \equiv \mathsf{c}\ \&\&$
$\quad \mathsf{sat}'\ (\mathsf{andP} \,\$\, \mathsf{map\ lit} \,\$\, \mathsf{g\ vs_1}/([0\mathinner{..}], \mathsf{xs})))\ (\mathsf{map\ complete\ cs}) \Rightarrow \mathsf{c} \in \mathsf{cs}$
$\equiv \{\mathsf{any\ f\ (map\ g\ xs)} = \mathsf{any\ (f \circ g)\ xs}\}$
$\mathsf{any}\ ((\lambda(\mathsf{Pattern\ c_1\ vs_1}) \rightarrow \mathsf{c_1} \equiv \mathsf{c}\ \&\&$
$\quad \mathsf{sat}'\ (\mathsf{andP} \,\$\, \mathsf{map\ lit} \,\$\, \mathsf{g\ vs_1}/([0\mathinner{..}], \mathsf{xs}))) \circ \mathsf{complete})\ \mathsf{cs} \Rightarrow \mathsf{c} \in \mathsf{cs}$

$\equiv$ {c $\in$ cs = any ($\equiv$ c) cs}
any ((λ(Pattern c_1 vs_1) $\rightarrow$ $c_1 \equiv$ c &&
  sat′ (andP \$ map lit \$ g vs_1/([0 ..], xs))) $\circ$ complete) cs $\Rightarrow$ any ($\equiv$ c) cs
$\Leftarrow$ {*lift implication over* any}
(λ(Pattern c_1 vs_1) $\rightarrow$ $c_1 \equiv$ c &&
  sat′ (andP \$ map lit \$ g vs_1/([0 ..], xs))) (complete c′) $\Rightarrow$ c′ $\equiv$ c
$\equiv$ {*inline* complete}
c′ $\equiv$ c && sat′ (andP \$ map lit \$ g (map (const Any) (nonRecs c′))
  /([0 ..], xs)) $\Rightarrow$ c′ $\equiv$ c
$\Leftarrow$ {*weaken implication*}
c′ $\equiv$ c $\Rightarrow$ c′ $\equiv$ c
$\equiv$ {*tautology*}
True

Lemma C2 (MP)

As f is a local function definition of both $\lhd$ and $\rhd$, we use $f_\lhd$ and $f_\rhd$ to indicate which f we are referring to.

sat \$ Sat (Value c xs) ((c, i) $\rhd$ k) $\Rightarrow$ sat \$ Sat (xs !! i) k
$\equiv$ {*inline* $\rhd$}
sat \$ Sat (Value c xs) (map f k) $\Rightarrow$ sat \$ Sat (xs !! i) k
$\equiv$ {*inline* sat}
sat′ \$ (c $\lhd$ map $f_\rhd$ k)/([0 ..], xs) $\Rightarrow$ sat \$ Sat (xs !! i) k
$\equiv$ {*inline* $\lhd$}
sat′ \$ orP (map $f_\lhd$ (map $f_\rhd$ k))/([0 ..], xs) $\Rightarrow$ sat \$ Sat (xs !! i) k
$\equiv$ {map f $\circ$ map g = map (f $\circ$ g)}
sat′ \$ orP (map ($f_\lhd \circ f_\rhd$) k)/([0 ..], xs) $\Rightarrow$ sat \$ Sat (xs !! i) k

We now proceed by induction over k. We assume k may take the values [ ], Any : ks and ($ms_1 \star ms_2$) : ks.

Case: k = [ ]

sat′ \$ orP (map ($f_\lhd \circ f_\rhd$) k)/([0 ..], xs) $\Rightarrow$ sat \$ Sat (xs !! i) k
$\equiv$ {k = [ ]}
sat′ \$ orP (map ($f_\lhd \circ f_\rhd$) [ ])/([0 ..], xs) $\Rightarrow$ sat \$ Sat (xs !! i) [ ]
$\equiv$ {*inline* map}
sat′ \$ orP [ ]/([0 ..], xs) $\Rightarrow$ sat \$ Sat (xs !! i) [ ]
$\equiv$ {*inline* orP}
sat′ \$ false/([0 ..], xs) $\Rightarrow$ sat \$ Sat (xs !! i) [ ]
$\equiv$ {*inline* (/)}

$\mathsf{sat'}\ \mathsf{false} \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ [\,]$
$\equiv \{\textit{inline}\ \mathsf{sat'}\}$
$\mathsf{False} \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ [\,]$
$\equiv \{\textit{implication}\}$
True

Case: $\mathsf{k} = \mathsf{Any} : \mathsf{ks}$

$\mathsf{sat'}\ \$\ \mathsf{orP}\ (\mathsf{map}\ (\mathsf{f}_{\lhd} \circ \mathsf{f}_{\rhd})\ \mathsf{k})/([0\,..], \mathsf{xs}) \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ \mathsf{k}$
$\equiv \{\mathsf{k} = \mathsf{Any} : \mathsf{ks}\}$
$\mathsf{sat'}\ \$\ \mathsf{orP}\ (\mathsf{map}\ (\mathsf{f}_{\lhd} \circ \mathsf{f}_{\rhd})\ (\mathsf{Any} : \mathsf{ks}))/([0\,..], \mathsf{xs}) \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ (\mathsf{Any} : \mathsf{ks})$
$\Leftarrow \{\textit{weaken implication}\}$
$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ (\mathsf{Any} : \mathsf{ks})$

Case: $\mathsf{k} = \mathsf{Any} : \mathsf{ks}$ **;** $\mathsf{xs}\ !!\ \mathsf{i} = \mathsf{Bottom}$

$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ (\mathsf{Any} : \mathsf{ks})$
$\equiv \{\mathsf{xs}\ !!\ \mathsf{i} = \mathsf{Bottom}\}$
$\mathsf{sat}\ \$\ \mathsf{Sat}\ \mathsf{Bottom}\ (\mathsf{Any} : \mathsf{ks})$
$\equiv \{\textit{inline}\ \mathsf{sat}\}$
True

Case: $\mathsf{k} = \mathsf{Any} : \mathsf{ks}$ **;** $\mathsf{xs}\ !!\ \mathsf{i} = \mathsf{Value}\ \mathsf{c'}\ \mathsf{ys}$

$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ (\mathsf{Any} : \mathsf{ks})$
$\equiv \{\mathsf{xs}\ !!\ \mathsf{i} = \mathsf{Value}\ \mathsf{c'}\ \mathsf{ys}\}$
$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}\ \mathsf{ys})\ (\mathsf{Any} : \mathsf{ks})$
$\equiv \{\textit{inline}\ \mathsf{sat}\}$
$\mathsf{sat'}\ \$\ (\mathsf{c'} \lhd (\mathsf{Any} : \mathsf{ks}))/([0\,..], \mathsf{ys})$
$\equiv \{\textit{inline}\ \lhd\}$
$\mathsf{sat'}\ \$\ \mathsf{orP}\ (\mathsf{map}\ \mathsf{f}\ (\mathsf{Any} : \mathsf{ks}))/([0\,..], \mathsf{ys})$
$\equiv \{\textit{inline}\ \mathsf{map}\ \mathsf{f}\}$
$\mathsf{sat'}\ \$\ \mathsf{orP}\ (\mathsf{true} : \mathsf{map}\ \mathsf{f}\ \mathsf{ks})/([0\,..], \mathsf{ys})$
$\equiv \{\textit{inline}\ \mathsf{orP}\}$
$\mathsf{sat'}\ \$\ \mathsf{true}/([0\,..], \mathsf{ys})$
$\equiv \{\textit{inline}\ (/)\}$
$\mathsf{sat'}\ \mathsf{true}$
$\equiv \{\textit{inline}\ \mathsf{sat'}\}$
True

Case: $\mathsf{k} = (\mathsf{ms}_1 \star \mathsf{ms}_2) : \mathsf{ks}$

$\mathsf{sat}' \, \$ \, \mathsf{orP} \, (\mathsf{map} \, (\mathsf{f}_\lhd \circ \mathsf{f}_\rhd) \, \mathsf{k})/([0\,..], \mathsf{xs}) \Rightarrow \mathsf{sat} \, \$ \, \mathsf{Sat} \, (\mathsf{xs} \, !! \, \mathsf{i}) \, \mathsf{k}$
$\equiv \{\mathsf{k} = (\mathsf{ms}_1 \star \mathsf{ms}_2) : \mathsf{ks}\}$
$\mathsf{sat}' \, \$ \, \mathsf{orP} \, (\mathsf{map} \, (\mathsf{f}_\lhd \circ \mathsf{f}_\rhd) \, ((\mathsf{ms}_1 \star \mathsf{ms}_2) : \mathsf{ks}))/([0\,..], \mathsf{xs}) \Rightarrow$
$\quad \mathsf{sat} \, \$ \, \mathsf{Sat} \, (\mathsf{xs} \, !! \, \mathsf{i}) \, ((\mathsf{ms}_1 \star \mathsf{ms}_2) : \mathsf{ks})$
$\equiv \{\textit{inline} \; \mathsf{map}\}$
$\mathsf{sat}' \, \$ \, \mathsf{orP} \, ((\mathsf{f}_\lhd \, \$ \, \mathsf{f}_\rhd \, \$ \, \mathsf{ms}_1 \star \mathsf{ms}_2) : \mathsf{map} \, (\mathsf{f}_\lhd \circ \mathsf{f}_\rhd) \, \mathsf{ks})/([0\,..], \mathsf{xs}) \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{inline} \; (/)\}$
$\mathsf{sat}' \, \$ \, \mathsf{orP} \, (((\mathsf{f}_\lhd \, \$ \, \mathsf{f}_\rhd \, \$ \, \mathsf{ms}_1 \star \mathsf{ms}_2)/([0\,..], \mathsf{xs})) :$
$\quad (\mathsf{map} \, (\mathsf{f}_\lhd \circ \mathsf{f}_\rhd) \, \mathsf{ks}/([0\,..], \mathsf{xs}))) \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{inline} \; \mathsf{orP}\}$
$\mathsf{sat}' \, \$ \, ((\mathsf{f}_\lhd \, \$ \, \mathsf{f}_\rhd \, \$ \, \mathsf{ms}_1 \star \mathsf{ms}_2)/([0\,..], \mathsf{xs})) \; \vee$
$\quad \mathsf{orP} \, (\mathsf{map} \, (\mathsf{f}_\lhd \circ \mathsf{f}_\rhd) \, \mathsf{ks}/([0\,..], \mathsf{xs})) \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{inline} \; \mathsf{sat}'\}$
$\mathsf{sat}' \, ((\mathsf{f}_\lhd \, \$ \, \mathsf{f}_\rhd \, \$ \, \mathsf{ms}_1 \star \mathsf{ms}_2)/([0\,..], \mathsf{xs})) \; ||$
$\quad \mathsf{sat}' \, (\mathsf{orP} \, (\mathsf{map} \, (\mathsf{f}_\lhd \circ \mathsf{f}_\rhd) \, \mathsf{ks}/([0\,..], \mathsf{xs}))) \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{reinstate RHS}\}$
$\mathsf{sat}' \, ((\mathsf{f}_\lhd \, \$ \, \mathsf{f}_\rhd \, \$ \, \mathsf{ms}_1 \star \mathsf{ms}_2)/([0\,..], \mathsf{xs})) \; ||$
$\quad \mathsf{sat}' \, (\mathsf{orP} \, (\mathsf{map} \, (\mathsf{f}_\lhd \circ \mathsf{f}_\rhd) \, \mathsf{ks}/([0\,..], \mathsf{xs}))) \Rightarrow$
$\quad \mathsf{sat} \, \$ \, \mathsf{Sat} \, (\mathsf{xs} \, !! \, \mathsf{i}) \, ((\mathsf{ms}_1 \star \mathsf{ms}_2) : \mathsf{ks})$

Case: $\mathsf{k} = (\mathsf{ms}_1 \star \mathsf{ms}_2) : \mathsf{ks} \quad ; \quad \mathsf{xs} \, !! \, \mathsf{i} = \mathsf{Bottom}$

$\mathsf{sat}' \, ((\mathsf{f}_\lhd \, \$ \, \mathsf{f}_\rhd \, \$ \, \mathsf{ms}_1 \star \mathsf{ms}_2)/([0\,..], \mathsf{xs})) \; ||$
$\quad \mathsf{sat}' \, (\mathsf{orP} \, (\mathsf{map} \, (\mathsf{f}_\lhd \circ \mathsf{f}_\rhd) \, \mathsf{ks}/([0\,..], \mathsf{xs}))) \Rightarrow$
$\quad \mathsf{sat} \, \$ \, \mathsf{Sat} \, (\mathsf{xs} \, !! \, \mathsf{i}) \, ((\mathsf{ms}_1 \star \mathsf{ms}_2) : \mathsf{ks})$
$\Leftarrow \{\textit{weaken implication}\}$
$\mathsf{sat} \, \$ \, \mathsf{Sat} \, (\mathsf{xs} \, !! \, \mathsf{i}) \, ((\mathsf{ms}_1 \star \mathsf{ms}_2) : \mathsf{ks})$
$\equiv \{\mathsf{xs} \, !! \, \mathsf{i} = \mathsf{Bottom}\}$
$\mathsf{sat} \, \$ \, \mathsf{Sat} \, \mathsf{Bottom} \, ((\mathsf{ms}_1 \star \mathsf{ms}_2) : \mathsf{ks})$
$\equiv \{\textit{inline} \; \mathsf{sat}\}$
True

Case: $\mathsf{k} = (\mathsf{ms}_1 \star \mathsf{ms}_2) : \mathsf{ks} \quad ; \quad \mathsf{xs} \, !! \, \mathsf{i} = \mathsf{Value} \, \mathsf{c}' \, \mathsf{ys}$

$\mathsf{sat}' \, ((\mathsf{f}_\lhd \, \$ \, \mathsf{f}_\rhd \, \$ \, \mathsf{ms}_1 \star \mathsf{ms}_2)/([0\,..], \mathsf{xs})) \; ||$
$\quad \mathsf{sat}' \, (\mathsf{orP} \, (\mathsf{map} \, (\mathsf{f}_\lhd \circ \mathsf{f}_\rhd) \, \mathsf{ks}/([0\,..], \mathsf{xs}))) \Rightarrow$
$\quad \mathsf{sat} \, \$ \, \mathsf{Sat} \, (\mathsf{xs} \, !! \, \mathsf{i}) \, ((\mathsf{ms}_1 \star \mathsf{ms}_2) : \mathsf{ks})$
$\equiv \{\mathsf{xs} \, !! \, \mathsf{i} = \mathsf{Value} \, \mathsf{c}' \, \mathsf{ys}\}$
$\mathrm{LHS} \Rightarrow \mathsf{sat} \, \$ \, \mathsf{Sat} \, (\mathsf{Value} \, \mathsf{c}' \, \mathsf{ys}) \, ((\mathsf{ms}_1 \star \mathsf{ms}_2) : \mathsf{ks})$
$\equiv \{\textit{Lemma MP1}\}$
$\mathrm{LHS} \Rightarrow \mathsf{any} \, (\lambda \mathsf{k} \rightarrow \mathsf{sat} \, \$ \, \mathsf{Sat} \, (\mathsf{Value} \, \mathsf{c}' \, \mathsf{ys}) \, [\mathsf{k}]) \, ((\mathsf{ms}_1 \star \mathsf{ms}_2) : \mathsf{ks})$

$\equiv \{\textit{inline}\ \mathsf{any}\}$
$\text{LHS} \Rightarrow \mathsf{sat}\ (\mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}'\ \mathsf{ys})\ [\mathsf{ms}_1 \star \mathsf{ms}_2]) \mid\mid$
$\quad \mathsf{any}\ (\lambda \mathsf{k} \rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}'\ \mathsf{xs})\ [\mathsf{k}])\ \mathsf{ks}$
$\equiv \{\textit{Lemma MP1}\}$
$\text{LHS} \Rightarrow \mathsf{sat}\ (\mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}'\ \mathsf{ys})\ [\mathsf{ms}_1 \star \mathsf{ms}_2]) \mid\mid \mathsf{sat}\ (\mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}'\ \mathsf{ys})\ \mathsf{ks})$
$\equiv \{\textit{reinstate LHS}\}$
$\mathsf{sat}'\ ((\mathsf{f}_\triangleleft\ \$\ \mathsf{f}_\triangleright\ \$\ \mathsf{ms}_1 \star \mathsf{ms}_2)/([0\,..], \mathsf{xs})) \mid\mid$
$\quad \mathsf{sat}'\ (\mathsf{orP}\ (\mathsf{map}\ (\mathsf{f}_\triangleleft \circ \mathsf{f}_\triangleright)\ \mathsf{ks}/([0\,..], \mathsf{xs}))) \Rightarrow$
$\quad \mathsf{sat}\ (\mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}\ \mathsf{ys})\ [\mathsf{ms}_1 \star \mathsf{ms}_2]) \mid\mid \mathsf{sat}\ (\mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}'\ \mathsf{ys})\ \mathsf{ks})$
$\Leftarrow \{\textit{split the implication}\}$
$(\mathsf{sat}'\ ((\mathsf{f}_\triangleleft\ \$\ \mathsf{f}_\triangleright\ \$\ \mathsf{ms}_1 \star \mathsf{ms}_2)/([0\,..], \mathsf{xs})) \Rightarrow$
$\quad \mathsf{sat}\ (\mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}'\ \mathsf{ys})\ [\mathsf{ms}_1 \star \mathsf{ms}_2]))\ \&\&$
$\quad (\mathsf{sat}'\ (\mathsf{orP}\ (\mathsf{map}\ (\mathsf{f}_\triangleleft \circ \mathsf{f}_\triangleright)\ \mathsf{ks}/([0\,..], \mathsf{xs}))) \Rightarrow \mathsf{sat}\ (\mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}'\ \mathsf{ys})\ \mathsf{ks}))$
$\equiv \{\mathsf{xs}\ !!\ \mathsf{i} = \mathsf{Value}\ \mathsf{c}'\ \mathsf{ys}\}$
$(\mathsf{sat}'\ ((\mathsf{f}_\triangleleft\ \$\ \mathsf{f}_\triangleright\ \$\ \mathsf{ms}_1 \star \mathsf{ms}_2)/([0\,..], \mathsf{xs})) \Rightarrow \mathsf{sat}\ (\mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ [\mathsf{ms}_1 \star \mathsf{ms}_2]))\ \&\&$
$\quad (\mathsf{sat}'\ (\mathsf{orP}\ (\mathsf{map}\ (\mathsf{f}_\triangleleft \circ \mathsf{f}_\triangleright)\ \mathsf{ks}/([0\,..], \mathsf{xs}))) \Rightarrow \mathsf{sat}\ (\mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ \mathsf{ks}))$
$\equiv \{\textit{inductive hypothesis}\}$
$\mathsf{sat}'\ \$\ (\mathsf{f}_\triangleleft\ \$\ \mathsf{f}_\triangleright\ \$\ \mathsf{ms}_1 \star \mathsf{ms}_2)/([0\,..], \mathsf{xs}) \Rightarrow$
$\quad \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ [\mathsf{ms}_1 \star \mathsf{ms}_2]$

Case: $\mathsf{k} = (\mathsf{ms}_1 \star \mathsf{ms}_2) : \mathsf{ks} \quad ; \quad \mathsf{xs}\ !!\ \mathsf{i} = \mathsf{Value}\ \mathsf{c}'\ \mathsf{ys} \quad ; \quad \mathsf{isRec}\ (\mathsf{c}, \mathsf{i}) = \mathsf{False}$

$\mathsf{sat}'\ \$\ (\mathsf{f}_\triangleleft\ \$\ \mathsf{f}_\triangleright\ \$\ \mathsf{ms}_1 \star \mathsf{ms}_2)/([0\,..], \mathsf{xs}) \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ [\mathsf{ms}_1 \star \mathsf{ms}_2]$
$\equiv \{\textit{inline}\ \mathsf{f}_\triangleright,\ \textit{assuming}\ \mathsf{isRec}\ (\mathsf{c}, \mathsf{i}) = \mathsf{False}\}$
$\mathsf{sat}'\ \$\ (\mathsf{f}_\triangleleft\ \$\ [\mathsf{Pattern}\ \mathsf{c}\ [\textbf{if}\ \mathsf{i} \equiv \mathsf{j}\ \textbf{then}\ \mathsf{ms}_1 \star \mathsf{ms}_2\ \textbf{else}\ \mathsf{Any} \mid \mathsf{j} \leftarrow \mathsf{nonRecs}\ \mathsf{c}]] \star$
$\quad \mathsf{map}\ \mathsf{complete}\ (\mathsf{ctors}\ \mathsf{c}))/([0\,..], \mathsf{xs}) \Rightarrow \text{LHS}$
$\equiv \{\textit{inline}\ \mathsf{f}_\triangleleft\}$
$\mathsf{sat}'\ \$\ \mathsf{orP}\ [\mathsf{andP}\ \$\ \mathsf{map}\ \mathsf{lit}\ \$\ \mathsf{g}\ \mathsf{vs}_1 \mid \mathsf{Pattern}\ \mathsf{c}_1\ \mathsf{vs}_1 \leftarrow$
$\quad [\mathsf{Pattern}\ \mathsf{c}\ [\textbf{if}\ \mathsf{i} \equiv \mathsf{j}\ \textbf{then}\ \mathsf{ms}_1 \star \mathsf{ms}_2\ \textbf{else}\ \mathsf{Any} \mid \mathsf{j} \leftarrow \mathsf{nonRecs}\ \mathsf{c}]],$
$\qquad \mathsf{c}_1 \equiv \mathsf{c}]/([0\,..], \mathsf{xs}) \Rightarrow \text{LHS}$
$\equiv \{\textit{simplify list comprehension}\}$
$\mathsf{sat}'\ \$\ \mathsf{orP}\ [\mathsf{andP}\ \$\ \mathsf{map}\ \mathsf{lit}\ \$\ \mathsf{g}$
$\quad [\textbf{if}\ \mathsf{i} \equiv \mathsf{j}\ \textbf{then}\ \mathsf{ms}_1 \star \mathsf{ms}_2\ \textbf{else}\ \mathsf{Any} \mid \mathsf{j} \leftarrow \mathsf{nonRecs}\ \mathsf{c}]]$
$\quad /([0\,..], \mathsf{xs}) \Rightarrow \text{LHS}$
$\equiv \{\textit{inline}\ \mathsf{orP}\}$
$\mathsf{sat}'\ \$\ (\mathsf{andP}\ \$\ \mathsf{map}\ \mathsf{lit}\ \$\ \mathsf{g}$
$\quad [\textbf{if}\ \mathsf{i} \equiv \mathsf{j}\ \textbf{then}\ \mathsf{ms}_1 \star \mathsf{ms}_2\ \textbf{else}\ \mathsf{Any} \mid \mathsf{j} \leftarrow \mathsf{nonRecs}\ \mathsf{c}])$
$\quad /([0\,..], \mathsf{xs}) \Rightarrow \text{LHS}$
$\equiv \{\textit{inline}\ (/)\}$
$\mathsf{sat}'\ \$\ \mathsf{andP}\ \$\ \mathsf{map}\ \mathsf{lit}\ \$\ \mathsf{g}$
$\quad [\textbf{if}\ \mathsf{i} \equiv \mathsf{j}\ \textbf{then}\ \mathsf{ms}_1 \star \mathsf{ms}_2\ \textbf{else}\ \mathsf{Any} \mid \mathsf{j} \leftarrow \mathsf{nonRecs}\ \mathsf{c}]$
$\quad /([0\,..], \mathsf{xs}) \Rightarrow \text{LHS}$
$\equiv \{\textit{inline}\ \mathsf{sat}'\}$
$\mathsf{all}\ \mathsf{sat}\ \$\ \mathsf{g}\ [\textbf{if}\ \mathsf{i} \equiv \mathsf{j}\ \textbf{then}\ \mathsf{ms}_1 \star \mathsf{ms}_2\ \textbf{else}\ \mathsf{Any} \mid \mathsf{j} \leftarrow \mathsf{nonRecs}\ \mathsf{c}]$

```
  /([0 ..], xs) ⇒ LHS
≡ {inline g}
all sat $ (zipWith Sat non (map (:[])
  [if i ≡ j then ms1 ⋆ ms2 else Any | j ← nonRecs c]) ++
  map ( `Sat` [map complete (ctors c) ⋆ map complete (ctors c)]) rec)
  /([0 ..], xs) ⇒ LHS
≡ {inline (/)}
all sat $ zipWith Sat (non/([0 ..], xs))
  (map (:[]) [if i ≡ j then ms1 ⋆ ms2 else Any | j ← nonRecs c]) ++
  map ( `Sat` [map complete (ctors c) ⋆ map complete (ctors c)])
  (rec/([0 ..], xs)) ⇒ LHS
≡ {inline all}
all sat (zipWith Sat (non/([0 ..], xs))
  (map (:[]) [if i ≡ j then ms1 ⋆ ms2 else Any | j ← nonRecs c])) &&
  all sat (map ( `Sat` [map complete (ctors c) ⋆ map complete (ctors c)])
  (rec/([0 ..], xs))) ⇒ LHS
⇐ {weaken implication}
all sat $ zipWith Sat (non/([0 ..], xs))
  (map (:[]) [if i ≡ j then ms1 ⋆ ms2 else Any | j ← nonRecs c]) ⇒ LHS
≡ {inline map}
all sat $ zipWith Sat (non/([0 ..], xs))
  [if i ≡ j then [ms1 ⋆ ms2] else [Any] | j ← nonRecs c] ⇒ LHS
≡ {non = nonRecs c, by definition of nonRecs}
all sat $ zipWith Sat (non/([0 ..], xs))
  [if i ≡ j then [ms1 ⋆ ms2] else [Any] | j ← non] ⇒ LHS
≡ {rewrite list comprehension}
all sat $ zipWith Sat (non/([0 ..], xs))
  (map (λj → if i ≡ j then [ms1 ⋆ ms2] else [Any]) non) ⇒ LHS
≡ {inline (/)}
all sat $ zipWith Sat (map (/([0 ..], xs)) non)
  (map (λj → if i ≡ j then [ms1 ⋆ ms2] else [Any]) non) ⇒ LHS
≡ {zipWith f (map g xs) (map h xs) = map (λx → f (g x) (h x)) xs}
all sat $ map (λj → Sat (j/([0 ..], xs))
  (if i ≡ j then [ms1 ⋆ ms2] else [Any])) non ⇒ LHS
⇐ {weaken implication, using i ∈ non because of false isRec test}
all sat $ map (λj → Sat (j/([0 ..], xs))
  (if i ≡ j then [ms1 ⋆ ms2] else [Any])) [i] ⇒ LHS
≡ {inline map}
all sat $ [Sat (i/([0 ..], xs)) (if i ≡ i then [ms1 ⋆ ms2] else [Any])] ⇒ LHS
≡ {inline all}
sat $ Sat (i/([0 ..], xs)) (if i ≡ i then [ms1 ⋆ ms2] else [Any]) ⇒ LHS
≡ {inline (≡)}
sat $ Sat (i/([0 ..], xs)) (if True then [ms1 ⋆ ms2] else [Any]) ⇒ LHS
≡ {simplify if}
sat $ Sat (i/([0 ..], xs)) [ms1 ⋆ ms2] ⇒ LHS
≡ {inline (/)}
```

$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ [\mathsf{ms}_1 \star \mathsf{ms}_2] \Rightarrow \mathrm{LHS}$
$\equiv \{\textit{reinstate LHS}\}$
$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ [\mathsf{ms}_1 \star \mathsf{ms}_2] \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ [\mathsf{ms}_1 \star \mathsf{ms}_2]$
$\equiv \{\textit{tautology}\}$
True

Case: $\mathsf{k} = (\mathsf{ms}_1 \star \mathsf{ms}_2) : \mathsf{ks}$; $\mathsf{xs}\ !!\ \mathsf{i} = \mathsf{Value}\ \mathsf{c}'\ \mathsf{ys}$; $\mathsf{isRec}\ (\mathsf{c}, \mathsf{i}) = \mathsf{True}$

$\mathsf{sat}'\ \$\ (\mathsf{f}_{\lhd}\ \$\ \mathsf{f}_{\rhd}\ \$\ \mathsf{ms}_1 \star \mathsf{ms}_2)/([0\,..], \mathsf{xs}) \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ [\mathsf{ms}_1 \star \mathsf{ms}_2]$
$\equiv \{\textit{inline}\ \mathsf{f}_{\rhd}\textit{, assuming}\ \mathsf{isRec}\ (\mathsf{c}, \mathsf{i})\}$
$\mathsf{sat}'\ \$\ (\mathsf{f}_{\lhd}\ \$\ [\mathsf{complete}\ \mathsf{c}] \star \mathsf{merge}\ \mathsf{ms}_1\ \mathsf{ms}_2)/([0\,..], \mathsf{xs}) \Rightarrow \mathrm{LHS}$
$\equiv \{\textit{inline}\ \mathsf{f}_{\lhd}\}$
$\mathsf{sat}'\ \$\ \mathsf{orP}\ [\mathsf{andP}\ \$\ \mathsf{map}\ \mathsf{lit}\ \$\ \mathsf{g}\ \mathsf{vs}_1 \mid$
$\quad \mathsf{Pattern}\ \mathsf{c}_1\ \mathsf{vs}_1 \leftarrow [\mathsf{complete}\ \mathsf{c}], \mathsf{c}_1 \equiv \mathsf{c}]/([0\,..], \mathsf{xs}) \Rightarrow \mathrm{LHS}$
$\equiv \{\textit{inline}\ \mathsf{complete}\ \mathsf{c}\}$
$\mathsf{sat}'\ \$\ \mathsf{orP}\ [\mathsf{andP}\ \$\ \mathsf{map}\ \mathsf{lit}\ \$\ \mathsf{g}\ \mathsf{vs}_1 \mid \mathsf{Pattern}\ \mathsf{c}_1\ \mathsf{vs}_1 \leftarrow$
$\quad [\mathsf{Pattern}\ \mathsf{c}\ (\mathsf{map}\ (\mathsf{const}\ \mathsf{Any})\ (\mathsf{nonRecs}\ \mathsf{c}))], \mathsf{c}_1 \equiv \mathsf{c}]/([0\,..], \mathsf{xs}) \Rightarrow \mathrm{LHS}$
$\equiv \{\textit{simplify list comprehension}\}$
$\mathsf{sat}'\ \$\ \mathsf{orP}\ [\mathsf{andP}\ \$\ \mathsf{map}\ \mathsf{lit}\ \$\ \mathsf{g}\ \$\ \mathsf{map}\ (\mathsf{const}\ \mathsf{Any})\ (\mathsf{nonRecs}\ \mathsf{c})]$
$\quad /([0\,..], \mathsf{xs}) \Rightarrow \mathrm{LHS}$
$\equiv \{\textit{inline}\ \mathsf{orP}\}$
$\mathsf{sat}'\ \$\ \mathsf{andP}\ (\mathsf{map}\ \mathsf{lit}\ \$\ \mathsf{g}\ \$\ \mathsf{map}\ (\mathsf{const}\ \mathsf{Any})\ (\mathsf{nonRecs}\ \mathsf{c}))$
$\quad /([0\,..], \mathsf{xs}) \Rightarrow \mathrm{LHS}$
$\equiv \{\textit{inline}\ (/)\}$
$\mathsf{sat}'\ \$\ \mathsf{andP}\ \$\ \mathsf{map}\ \mathsf{lit}\ \$\ (\mathsf{g}\ \$\ \mathsf{map}\ (\mathsf{const}\ \mathsf{Any})\ (\mathsf{nonRecs}\ \mathsf{c}))$
$\quad /([0\,..], \mathsf{xs}) \Rightarrow \mathrm{LHS}$
$\equiv \{\textit{inline}\ \mathsf{sat}'\}$
$\mathsf{all}\ \mathsf{sat}\ \$\ (\mathsf{g}\ \$\ \mathsf{map}\ (\mathsf{const}\ \mathsf{Any})\ (\mathsf{nonRecs}\ \mathsf{c}))/([0\,..], \mathsf{xs}) \Rightarrow \mathrm{LHS}$
$\equiv \{\textit{inline}\ \mathsf{g}\}$
$\mathsf{all}\ \mathsf{sat}\ \$\ (\mathsf{zipWith}\ \mathsf{Sat}\ \mathsf{non}\ (\mathsf{map}\ (:[\,])\ \mathsf{vs}) +\!\!+$
$\quad \mathsf{map}\ (\ \text{`}\mathsf{Sat}\text{`}\ [\mathsf{merge}\ \mathsf{ms}_1\ \mathsf{ms}_2 \star \mathsf{merge}\ \mathsf{ms}_1\ \mathsf{ms}_2])\ \mathsf{rec})/([0\,..], \mathsf{xs}) \Rightarrow \mathrm{LHS}$
$\equiv \{\textit{inline}\ (/)\}$
$\mathsf{all}\ \mathsf{sat}\ \$\ \mathsf{zipWith}\ \mathsf{Sat}\ \mathsf{non}\ (\mathsf{map}\ (:[\,])\ \mathsf{vs})/([0\,..], \mathsf{xs}) +\!\!+$
$\quad \mathsf{map}\ (\ \text{`}\mathsf{Sat}\text{`}\ [\mathsf{merge}\ \mathsf{ms}_1\ \mathsf{ms}_2 \star \mathsf{merge}\ \mathsf{ms}_1\ \mathsf{ms}_2])\ \mathsf{rec}/([0\,..], \mathsf{xs}) \Rightarrow \mathrm{LHS}$
$\equiv \{\textit{inline}\ \mathsf{all}\}$
$\mathsf{all}\ \mathsf{sat}\ (\mathsf{zipWith}\ \mathsf{Sat}\ \mathsf{non}\ (\mathsf{map}\ (:[\,])\ \mathsf{vs})/([0\,..], \mathsf{xs}))\ \&\&$
$\quad \mathsf{all}\ \mathsf{sat}\ (\mathsf{map}\ (\ \text{`}\mathsf{Sat}\text{`}\ [\mathsf{merge}\ \mathsf{ms}_1\ \mathsf{ms}_2 \star \mathsf{merge}\ \mathsf{ms}_1\ \mathsf{ms}_2])\ \mathsf{rec}$
$\quad /([0\,..], \mathsf{xs})) \Rightarrow \mathrm{LHS}$
$\equiv \{\textit{weaken implication}\}$
$\mathsf{all}\ \mathsf{sat}\ \$\ \mathsf{map}\ (\ \text{`}\mathsf{Sat}\text{`}\ [\mathsf{merge}\ \mathsf{ms}_1\ \mathsf{ms}_2 \star \mathsf{merge}\ \mathsf{ms}_1\ \mathsf{ms}_2])\ \mathsf{rec}$
$\quad /([0\,..], \mathsf{xs}) \Rightarrow \mathrm{LHS}$
$\equiv \{\textit{eta expand}\}$
$\mathsf{all}\ \mathsf{sat}\ \$\ \mathsf{map}\ (\lambda \mathsf{j} \rightarrow \mathsf{Sat}\ \mathsf{j}\ [\mathsf{merge}\ \mathsf{ms}_1\ \mathsf{ms}_2 \star \mathsf{merge}\ \mathsf{ms}_1\ \mathsf{ms}_2])\ \mathsf{rec}$
$\quad /([0\,..], \mathsf{xs}) \Rightarrow \mathrm{LHS}$

$\equiv \{\textit{inline } (/)\}$
all sat \$ map (λj $\rightarrow$ Sat (j/([0 . .], xs))
  [merge ms_1 $\mathsf{ms}_2 \star$ merge ms_1 ms_2]) rec $\Rightarrow$ LHS
$\equiv$ {*combine* all *and* map}
all (λj $\rightarrow$ sat \$ Sat (j/([0 . .], xs))
  [merge ms_1 $\mathsf{ms}_2 \star$ merge ms_1 ms_2]) rec $\Rightarrow$ LHS
$\Leftarrow$ {*weaken implication, as* i $\in$ rec*, because of* isRec *test*}
all (λj $\rightarrow$ sat \$ Sat (j/([0 . .], xs))
  [merge ms_1 $\mathsf{ms}_2 \star$ merge ms_1 ms_2]) [i] $\Rightarrow$ LHS
$\equiv$ {*inline* all}
sat \$ Sat (i/([0 . .], xs)) [merge ms_1 $\mathsf{ms}_2 \star$ merge ms_1 ms_2] $\Rightarrow$ LHS
$\equiv \{\textit{inline } (/)\}$
sat \$ Sat (xs !! i) [merge ms_1 $\mathsf{ms}_2 \star$ merge ms_1 ms_2] $\Rightarrow$ RHS
$\equiv$ {*reinstate RHS*}
sat \$ Sat (xs !! i) [merge ms_1 $\mathsf{ms}_2 \star$ merge ms_1 ms_2] $\Rightarrow$
  sat \$ Sat (xs !! i) [$\mathsf{ms}_1 \star \mathsf{ms}_2$]
$\equiv$ {xs !! i = Value c′ ys}
sat \$ Sat (Value c′ ys) [merge ms_1 $\mathsf{ms}_2 \star$ merge ms_1 ms_2] $\Rightarrow$
  sat \$ Sat (Value c′ ys) [$\mathsf{ms}_1 \star \mathsf{ms}_2$]
$\Leftarrow$ {*Lemma MP2*}
sat \$ Sat (Value c′ ys) [$\mathsf{ms}_1 \star \mathsf{ms}_2$] $\Rightarrow$ sat \$ Sat (Value c′ ys) [$\mathsf{ms}_1 \star \mathsf{ms}_2$]
$\equiv$ {*tautology*}
True

A.5 Auxiliary Lemmas

To prove the main result we shall need the following lemmas.

Lemma A1

$(\mathsf{x} \in (\mathsf{ys} \setminus [\mathsf{x}])) \equiv \mathsf{False}$

The elem application is only true if ys \ [x] contains x. The expression cannot contain x, as if it existed in ys it was removed, therefore this application is always False.

Lemma A2

sat′ (eval x<cs) $\equiv$ satE′ (x<cs)

We argue as follows:

$\mathsf{sat}'\ (\mathsf{eval}\ \mathsf{x}{\lessdot}\mathsf{cs}) \equiv \mathsf{satE}'\ (\mathsf{x}{\lessdot}\mathsf{cs})$
$\equiv \{\textit{inline}\ \mathsf{sat}'\ \textit{and}\ \mathsf{satE}'\}$
$\mathsf{tautP}\ \mathsf{sat}\ (\mathsf{eval}\ \mathsf{x}{\lessdot}\mathsf{cs}) \equiv \mathsf{tautP}\ \mathsf{satE}\ (\mathsf{x}{\lessdot}\mathsf{cs})$
$\equiv \{\textit{inline}\ \mathsf{tautP}\ \textit{and reduce common bits}\}$
$\mathsf{mapP}\ (\mathsf{bool} \circ \mathsf{sat})\ (\mathsf{eval}\ \mathsf{x}{\lessdot}\mathsf{cs}) \equiv \mathsf{mapP}\ (\mathsf{bool} \circ \mathsf{satE})\ (\mathsf{x}{\lessdot}\mathsf{cs})$

Now we can use the type signature of $\lessdot$:

$(\lessdot) :: \alpha \to [\mathsf{CtorName}] \to \mathsf{Prop}\ (\mathsf{Sat}\ \alpha)$

The theorems for free work (Wadler 1989) shows that the first argument will end up as the first argument of the Sat constructor, unmodified and unexamined. The satE function applies eval to the first argument of Sat, therefore the equivalence holds.

Lemma A3

precond `"error"` $\equiv$ false

The initial computation of precond will return false. All successive computations will be at least as restrictive, therefore the result must be false.

Lemma A4

precond f $\Rightarrow$ reduce \$ pre \$ body f

The definition of precond f is a conjunction where the second conjunct is reduce \$ pre \$ body f, therefore precond f is at least as restrictive as the alternative.

Lemma A5

prePost f k $\Rightarrow$ reduce (lit \$ body f `` `Sat` `` k)

The definition of prePost f k is a conjunction where the second conjunct is reduce (lit \$ body f `` `Sat` `` k), therefore prePost f k is at least as restrictive as the alternative.

Lemma A6

$\mathsf{satE}'\ (\mathsf{pre\ e}\ /\ (\mathsf{vs}, \mathsf{ys}))\ \&\&\ \mathsf{all}\ (\mathsf{satE}' \circ \mathsf{pre})\ \mathsf{ys} \Rightarrow \mathsf{satE}'\ \$\ \mathsf{pre}\ \$\ \mathsf{e}\ /\ (\mathsf{vs}, \mathsf{ys})$

We assume that all free variables in e are bound in vs. To shorten the proofs we do not explicitly write the all (satE′ ∘ pre) ys term, as it is never manipulated. We proceed by induction on e.

Case: e = Var v

satE′ \$ pre e / (vs, ys) ⇒ satE′ \$ pre \$ e / (vs, ys)
≡ {e = Var v}
satE′ \$ pre (Var v) / (vs, ys) ⇒ satE′ \$ pre \$ Var v / (vs, ys)
≡ {*inline* pre *on LHS*}
satE′ \$ true / (vs, ys) ⇒ satE′ \$ pre \$ Var v / (vs, ys)
≡ {*inline (/) on LHS*}
satE′ true ⇒ satE′ \$ pre \$ Var v / (vs, ys)
≡ {*inline* satE′}
True ⇒ satE′ \$ pre \$ Var v / (vs, ys)
≡ {*reintroduce hidden term*}
all (satE′ ∘ pre) ys ⇒ satE′ \$ pre \$ Var v / (vs, ys)

We know that v will be a member of vs, and that the result will be pre y, where y is drawn from ys. Since all ys satisfy the precondition, then so will the particular y we substitute.

Case: e = Sel x s

satE′ \$ pre e / (vs, ys) ⇒ satE′ \$ pre \$ e / (vs, ys)
≡ {e = Sel x s}
satE′ \$ pre (Sel x s) / (vs, ys) ⇒ satE′ \$ pre \$ Sel x s / (vs, ys)
≡ {*inline (/) on RHS*}
satE′ \$ pre (Sel x s) / (vs, ys) ⇒ satE′ \$ pre \$ Sel (x / (vs, ys)) s
≡ {*inline* pre *on RHS*}
satE′ \$ pre (Sel x s) / (vs, ys) ⇒ satE′ \$ true
≡ {*inline* satE′ *on RHS*}
satE′ \$ pre (Sel x s) / (vs, ys) ⇒ True
≡ {*implication*}
True

Case: $\mathsf{e = Make\ c\ xs}$

$\mathsf{satE'\ \$\ pre\ x\ /\ (vs, ys) \Rightarrow satE'\ \$\ pre\ \$\ x\ /\ (vs, ys)}$
$\equiv \{\mathsf{x = Make\ c\ xs}\}$
$\mathsf{satE'\ \$\ pre\ (Make\ c\ xs)\ /\ (vs, ys) \Rightarrow satE'\ \$\ pre\ \$\ Make\ c\ xs\ /\ (vs, ys)}$
$\equiv \{\textit{inline}\ (/)\ \textit{on RHS}\}$
$\mathsf{satE'\ \$\ pre\ (Make\ c\ xs)\ /\ (vs, ys) \Rightarrow}$
$\quad \mathsf{satE'\ \$\ pre\ \$\ Make\ c\ \$\ map\ (/(vs, ys))\ xs}$
$\equiv \{\textit{inline}\ \mathsf{pre}\ \textit{on both sides}\}$
$\mathsf{satE'\ \$\ andP\ (map\ pre\ xs)\ /\ (vs, ys) \Rightarrow}$
$\quad \mathsf{satE'\ \$\ andP\ \$\ map\ (pre \circ (/(vs, ys)))\ xs}$
$\equiv \{\textit{inline}\ (/)\ \textit{on LHS}\}$
$\mathsf{satE'\ \$\ andP\ \$\ map\ ((/(vs, ys)) \circ pre)\ xs \Rightarrow}$
$\quad \mathsf{satE'\ \$\ andP\ \$\ map\ (pre \circ (/(vs, ys)))\ xs}$
$\equiv \{\textit{inline}\ \mathsf{satE'}\ \textit{on both sides}\}$
$\mathsf{all\ (satE' \circ (/(vs, ys)) \circ pre)\ xs \Rightarrow all\ (satE' \circ pre \circ (/(vs, ys)))\ xs}$
$\Leftarrow \{\textit{by induction}\}$
True

Case: $\mathsf{e = Call\ f\ xs}$

$\mathsf{satE'\ \$\ pre\ e\ /\ (vs, ys) \Rightarrow satE'\ \$\ pre\ \$\ e\ /\ (vs, ys)}$
$\equiv \{\mathsf{e = Call\ f\ xs}\}$
$\mathsf{satE'\ \$\ pre\ (Call\ f\ xs)\ /\ (vs, ys) \Rightarrow satE'\ \$\ pre\ \$\ Call\ f\ xs\ /\ (vs, ys)}$
$\equiv \{\textit{inline}\ (/)\ \textit{on RHS}\}$
$\mathsf{satE'\ \$\ pre\ (Call\ f\ xs)\ /\ (vs, ys) \Rightarrow satE'\ \$\ pre\ \$\ Call\ f\ \$\ map\ (/(vs, ys))\ xs}$
$\equiv \{\textit{inline}\ \mathsf{pre}\}$
$\mathsf{satE'\ \$\ (pre'\ f\ xs\ \land\ andP\ (map\ pre\ xs))\ /\ (vs, ys)} \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{inline}\ (/)\}$
$\mathsf{satE'\ \$\ (pre'\ f\ xs\ /\ (vs, ys))\ \land\ andP\ (map\ (pre \circ (/(vs, ys)))\ xs)} \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{inline}\ \mathsf{satE'}\}$
$\mathsf{satE'\ (pre'\ f\ xs\ /\ (vs, ys))\ \&\&\ all\ (satE' \circ pre \circ (/(vs, ys)))\ xs} \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{switch to RHS}\}$
$\mathrm{LHS} \Rightarrow \mathsf{satE'\ \$\ pre\ \$\ Call\ f\ \$\ map\ (/(vs, ys))\ xs}$
$\equiv \{\textit{inline}\ \mathsf{pre}\}$
$\mathrm{LHS} \Rightarrow \mathsf{satE'\ \$\ (pre'\ f\ (map\ (/(vs, ys))\ xs))\ \land}$
$\quad \mathsf{andP\ (map\ (pre \circ (/(vs, ys)))\ xs)}$
$\equiv \{\textit{inline}\ \mathsf{satE'}\}$
$\mathrm{LHS} \Rightarrow \mathsf{satE'\ (pre'\ f\ (map\ (/(vs, ys))\ xs))\ \&\&\ all\ (satE' \circ pre \circ (/(vs, ys)))\ xs}$
$\equiv \{\textit{reinstate LHS}\}$
$\mathsf{satE'\ (pre'\ f\ xs\ /\ (vs, ys))\ \&\&\ all\ (satE' \circ pre \circ (/(vs, ys)))\ xs \Rightarrow}$
$\quad \mathsf{satE'\ (pre'\ f\ (map\ (/(vs, ys))\ xs))\ \&\&\ all\ (satE' \circ pre \circ (/(vs, ys)))\ xs}$
$\Leftarrow \{\textit{eliminate common term}\}$
$\mathsf{satE'\ \$\ pre'\ f\ xs\ /\ (vs, ys) \Rightarrow satE'\ \$\ pre'\ f\ (map\ (/(vs, ys))\ xs)}$

$\equiv \{\textit{inline}\ \text{pre}'\}$
$\text{satE}'\ \$\ (\text{precond f}/(\text{args f}, \text{xs}))\ /\ (\text{vs}, \text{ys}) \Rightarrow$
$\quad \text{satE}'\ \$\ \text{precond f}/(\text{args f}, \text{map}\ (/(\text{vs}, \text{ys}))\ \text{xs})$
$\equiv \{\textit{inline}\ (/)\ \textit{on LHS}\}$
$\text{satE}'\ \$\ \text{precond f}/(\text{args f}, \text{map}\ (/(\text{vs}, \text{ys}))\ \text{xs}) \Rightarrow$
$\quad \text{satE}'\ \$\ \text{precond f}/(\text{args f}, \text{map}\ (/(\text{vs}, \text{ys}))\ \text{xs})$
$\equiv \{\textit{tautology}\}$
True

Case: $\text{e} = \text{Case x as}$

$\text{satE}'\ \$\ \text{pre e}\ /\ (\text{vs}, \text{ys}) \Rightarrow \text{satE}'\ \$\ \text{pre}\ \$\ \text{e}\ /\ (\text{vs}, \text{ys})$
$\equiv \{\text{e} = \text{Case x as}\}$
$\text{satE}'\ \$\ \text{pre (Case x as)}\ /\ (\text{vs}, \text{ys}) \Rightarrow \text{satE}'\ \$\ \text{pre}\ \$\ \text{Case x as}\ /\ (\text{vs}, \text{ys})$
$\equiv \{\textit{inline}\ (/)\ \textit{on RHS}\}$
$\text{satE}'\ \$\ \text{pre (Case x as)}\ /\ (\text{vs}, \text{ys}) \Rightarrow$
$\quad \text{satE}'\ \$\ \text{pre}\ \$\ \text{Case}\ (\text{x}\ /\ (\text{vs}, \text{ys}))\ (\text{map}\ (/(\text{vs}, \text{ys}))\ \text{as})$
$\equiv \{\textit{inline}\ \text{pre}\ \textit{on both sides}\}$
$\text{satE}'\ \$\ (\text{pre x}\ \wedge\ \text{andP (map alt as)})\ /\ (\text{vs}, \text{ys}) \Rightarrow$
$\quad \text{satE}'\ \$\ \text{pre}\ (\text{x}\ /\ (\text{vs}, \text{ys}))\ \wedge\ \text{andP}\ (\text{map}\ (\text{alt} \circ (/(\text{vs}, \text{ys})))\ \text{as})$
$\equiv \{\textit{inline}\ (/)\ \textit{on LHS}\}$
$\text{satE}'\ \$\ (\text{pre x}\ /\ (\text{vs}, \text{ys}))\ \wedge\ \text{andP}\ (\text{map}\ ((/(\text{vs}, \text{ys})) \circ \text{alt})\ \text{as}) \Rightarrow$
$\quad \text{satE}'\ \$\ \text{pre}\ (\text{x}\ /\ (\text{vs}, \text{ys}))\ \wedge\ \text{andP}\ (\text{map}\ (\text{alt} \circ (/(\text{vs}, \text{ys})))\ \text{as})$
$\equiv \{\textit{inline}\ \text{satE}'\}$
$\text{satE}'\ (\text{pre x}\ /\ (\text{vs}, \text{ys}))\ \&\&\ \text{all}\ (\text{satE}' \circ (/(\text{vs}, \text{ys})) \circ \text{alt})\ \text{as} \Rightarrow$
$\quad \text{satE}'\ (\text{pre}\ (\text{x}\ /\ (\text{vs}, \text{ys})))\ \&\&\ \text{all}\ (\text{satE}' \circ \text{alt} \circ (/(\text{vs}, \text{ys})))\ \text{as}$
$\Leftarrow \{\textit{by induction}\}$
$\text{all}\ (\text{satE}' \circ (/(\text{vs}, \text{ys})) \circ \text{alt})\ \text{as} \Rightarrow \text{all}\ (\text{satE}' \circ \text{alt} \circ (/(\text{vs}, \text{ys})))\ \text{as}$
$\Leftarrow \{\textit{implication over}\ \text{all}\}$
$\text{satE}'\ \$\ \text{alt a}\ /\ (\text{vs}, \text{ys}) \Rightarrow \text{satE}'\ \$\ \text{alt}\ \$\ \text{a}\ /\ (\text{vs}, \text{ys})$
$\equiv \{\textit{instantiate}\ \text{a}\ \textit{as a general}\ \text{Alt}\}$
$\text{satE}'\ \$\ \text{alt (Alt c ws y)}\ /\ (\text{vs}, \text{ys}) \Rightarrow \text{satE}'\ \$\ \text{alt}\ \$\ \text{Alt c ws y}\ /\ (\text{vs}, \text{ys})$
$\equiv \{\textit{inline}\ (/)\ \textit{on RHS}\}$
$\text{satE}'\ \$\ \text{alt (Alt c ws y)}\ /\ (\text{vs}, \text{ys}) \Rightarrow \text{satE}'\ \$\ \text{alt}\ \$\ \text{Alt c ws}\ (\text{y}\ /\ (\text{vs}, \text{ys}))$
$\equiv \{\textit{inline}\ \text{alt}\}$
$\text{satE}'\ \$\ (\text{x}{\lessdot}(\text{ctors c} \setminus [\text{c}])\ \vee\ \text{pre y})\ /\ (\text{vs}, \text{ys}) \Rightarrow$
$\quad \text{satE}'\ \$\ (\text{x}\ /\ (\text{vs}, \text{ys}){\lessdot}(\text{ctors c} \setminus [\text{c}]))\ \vee\ \text{pre}\ (\text{y}\ /\ (\text{vs}, \text{ys}))$
$\equiv \{\textit{let}\ \text{cs} = \text{ctors c} \setminus [\text{c}]\}$
$\text{satE}'\ \$\ (\text{x}{\lessdot}\text{cs}\ \vee\ \text{pre y})\ /\ (\text{vs}, \text{ys}) \Rightarrow$
$\quad \text{satE}'\ \$\ (\text{x}\ /\ (\text{vs}, \text{ys}){\lessdot}\text{cs})\ \vee\ \text{pre}\ (\text{y}\ /\ (\text{vs}, \text{ys}))$
$\equiv \{\textit{inline}\ (/)\ \textit{on LHS}\}$
$\text{satE}'\ \$\ (\text{x}\ /\ (\text{vs}, \text{ys}){\lessdot}\text{cs})\ \vee\ (\text{pre y}\ /\ (\text{vs}, \text{ys})) \Rightarrow$
$\quad \text{satE}'\ \$\ (\text{x}\ /\ (\text{vs}, \text{ys}){\lessdot}\text{cs})\ \vee\ \text{pre}\ (\text{y}\ /\ (\text{vs}, \text{ys}))$
$\equiv \{\textit{inline}\ \text{satE}'\ \textit{on both sides}\}$

satE′ (x / (vs, ys)<cs) || satE′ (pre y / (vs, ys)) ⇒
  satE′ (x / (vs, ys)<cs) || satE′ (pre (y / (vs, ys)))
⇐ {*remove duplicate bits on each side*}
satE′ $ pre y / (vs, ys) ⇒ satE′ $ pre $ y / (vs, ys)
⇐ {*by induction*}
True

Lemma A7

satE′ $ red e k/sub ⇒ sat $ Sat (eval $ e / sub) k

We proceed by induction on e.

Case: e = Var v

satE′ $ red e k/sub ⇒ sat $ Sat (eval $ e / sub) k
≡ {e = Var v}
satE′ $ red (Var v) k/sub ⇒ sat $ Sat (eval $ Var v / sub) k
≡ {*inline* red}
satE′ $ lit (Sat v k)/sub ⇒ sat $ Sat (eval $ Var v / sub) k
≡ {*inline (/) on LHS*}
satE′ $ lit (Sat (v/sub) k) ⇒ sat $ Sat (eval $ Var v / sub) k
≡ {*promote* v *on LHS to* Var v *because (/) operates identically on both*}
satE′ $ lit (Sat (Var v / sub) k) ⇒ sat $ Sat (eval $ Var v / sub) k
≡ {*inline* satE′}
satE (Sat (Var v / sub) k) ⇒ sat $ Sat (eval $ Var v / sub) k
≡ {*inline* satE}
sat $ Sat (eval $ Var v / sub) k ⇒ sat $ Sat (eval $ Var v / sub) k
≡ {*tautology*}
True

Case: e = Sel x (c, i)

Here we may assume eval (x / sub) = Value c xs.

satE′ $ red e k/sub ⇒ sat $ Sat (eval $ e / sub) k
≡ {e = Sel x (c, i)}
satE′ $ red (Sel x (c, i)) k/sub ⇒ sat $ Sat (eval $ Sel x (c, i) / sub) k
≡ {*inline* red}
satE′ $ red x ((c, i) ▷ k)/sub ⇒ sat $ Sat (eval $ Sel x (c, i) / sub) k
≡ {*inline (/) on RHS*}

$\mathsf{satE}'\ \$\ \mathsf{red}\ \mathsf{x}\ ((\mathsf{c},\mathsf{i}) \rhd \mathsf{k})/\mathsf{sub} \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{eval}\ \$\ \mathsf{Sel}\ (\mathsf{x}\ /\ \mathsf{sub})\ (\mathsf{c},\mathsf{i}))\ \mathsf{k}$
$\equiv \{\textit{inline}\ \mathsf{eval}\ \textit{on RHS}\}$
$\mathsf{satE}'\ \$\ \mathsf{red}\ \mathsf{x}\ ((\mathsf{c},\mathsf{i}) \rhd \mathsf{k})/\mathsf{sub} \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{xs}\ !!\ \mathsf{i})\ \mathsf{k}$
$\Leftarrow \{\textit{Lemma C2}\}$
$\mathsf{satE}'\ \$\ \mathsf{red}\ \mathsf{x}\ ((\mathsf{c},\mathsf{i}) \rhd \mathsf{k})/\mathsf{sub} \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}\ \mathsf{xs})\ ((\mathsf{c},\mathsf{i}) \rhd \mathsf{k})$
$\equiv \{\textit{replace using the assumption}\}$
$\mathsf{satE}'\ \$\ \mathsf{red}\ \mathsf{x}\ ((\mathsf{c},\mathsf{i}) \rhd \mathsf{k})/\mathsf{sub} \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{eval}\ (\mathsf{x}\ /\ \mathsf{sub}))\ ((\mathsf{c},\mathsf{i}) \rhd \mathsf{k})$
$\Leftarrow \{\textit{by induction}\}$
True

Case: $\mathsf{e} = \mathsf{Make}\ \mathsf{c}\ \mathsf{xs}$

$\mathsf{satE}'\ \$\ \mathsf{red}\ \mathsf{e}\ \mathsf{k}/\mathsf{sub} \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{eval}\ \$\ \mathsf{e}\ /\ \mathsf{sub})\ \mathsf{k}$
$\equiv \{\mathsf{e} = \mathsf{Make}\ \mathsf{c}\ \mathsf{xs}\}$
$\mathsf{satE}'\ \$\ \mathsf{red}\ (\mathsf{Make}\ \mathsf{c}\ \mathsf{xs})\ \mathsf{k}/\mathsf{sub} \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{eval}\ \$\ \mathsf{Make}\ \mathsf{c}\ \mathsf{xs}\ /\ \mathsf{sub})\ \mathsf{k}$
$\equiv \{\textit{inline}\ \mathsf{red}\}$
$\mathsf{satE}'\ \$\ \mathsf{reduce}\ ((\mathsf{c} \lhd \mathsf{k})/([0\,..],\mathsf{xs}))/\mathsf{sub} \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{eval}\ \$\ \mathsf{Make}\ \mathsf{c}\ \mathsf{xs}\ /\ \mathsf{sub})\ \mathsf{k}$
$\Leftarrow \{\textit{Lemma A8}\}$
$\mathsf{satE}'\ \$\ ((\mathsf{c} \lhd \mathsf{k})/([0\,..],\mathsf{xs}))\ /\ \mathsf{sub} \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{eval}\ \$\ \mathsf{Make}\ \mathsf{c}\ \mathsf{xs}\ /\ \mathsf{sub})\ \mathsf{k}$
$\equiv \{\textit{inline}\ (/)\ \textit{on LHS}\}$
$\mathsf{satE}'\ \$\ (\mathsf{c} \lhd \mathsf{k})/([0\,..],\mathsf{map}\ (/\mathsf{sub})\ \mathsf{xs}) \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{eval}\ \$\ \mathsf{Make}\ \mathsf{c}\ \mathsf{xs}\ /\ \mathsf{sub})\ \mathsf{k}$
$\equiv \{\textit{inline}\ \mathsf{satE}'\ \textit{on LHS}\}$
$\mathsf{sat}'\ \$\ (\mathsf{c} \lhd \mathsf{k})/([0\,..],\mathsf{map}\ (\mathsf{eval} \circ (/\mathsf{sub}))\ \mathsf{xs}) \Rightarrow$
$\quad \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{eval}\ \$\ \mathsf{Make}\ \mathsf{c}\ \mathsf{xs}\ /\ \mathsf{sub})\ \mathsf{k}$
$\equiv \{\textit{inline}\ (/)\ \textit{on RHS}\}$
$\mathsf{sat}'\ \$\ (\mathsf{c} \lhd \mathsf{k})/([0\,..],\mathsf{map}\ (\mathsf{eval} \circ (/\mathsf{sub}))\ \mathsf{xs}) \Rightarrow$
$\quad \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{eval}\ \$\ \mathsf{Make}\ \mathsf{c}\ (\mathsf{map}\ (/\mathsf{sub})\ \mathsf{xs}))\ \mathsf{k}$
$\equiv \{\textit{inline}\ \mathsf{eval}\ \textit{on RHS}\}$
$\mathsf{sat}'\ \$\ (\mathsf{c} \lhd \mathsf{k})/([0\,..],\mathsf{map}\ (\mathsf{eval} \circ (/\mathsf{sub}))\ \mathsf{xs}) \Rightarrow$
$\quad \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}\ (\mathsf{map}\ (\mathsf{eval} \circ (/\mathsf{sub}))\ \mathsf{xs}))\ \mathsf{k}$
$\equiv \{\textit{fold back}\ \mathsf{sat}\}$
$\mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}\ (\mathsf{map}\ (\mathsf{eval} \circ (/\mathsf{sub}))\ \mathsf{xs}))\ \mathsf{k} \Rightarrow$
$\quad \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{Value}\ \mathsf{c}\ (\mathsf{map}\ (\mathsf{eval} \circ (/\mathsf{sub}))\ \mathsf{xs}))\ \mathsf{k}$
$\equiv \{\textit{tautology}\}$
True

Case: $\mathsf{e} = \mathsf{Call}\ \mathsf{f}\ \mathsf{xs}$

$\mathsf{satE}'\ \$\ \mathsf{red}\ \mathsf{e}\ \mathsf{k}/\mathsf{sub} \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{eval}\ \$\ \mathsf{e}\ /\ \mathsf{sub})\ \mathsf{k}$
$\equiv \{\mathsf{e} = \mathsf{Call}\ \mathsf{f}\ \mathsf{xs}\}$
$\mathsf{satE}'\ \$\ \mathsf{red}\ (\mathsf{Call}\ \mathsf{f}\ \mathsf{xs})\ \mathsf{k}/\mathsf{sub} \Rightarrow \mathsf{sat}\ \$\ \mathsf{Sat}\ (\mathsf{eval}\ \$\ \mathsf{Call}\ \mathsf{f}\ \mathsf{xs}\ /\ \mathsf{sub})\ \mathsf{k}$
$\equiv \{\textit{inline}\ \mathsf{red}\}$
$\mathsf{satE}'\ \$\ \mathsf{reduce}\ (\mathsf{prePost}\ \mathsf{f}\ \mathsf{k}/(\mathsf{args}\ \mathsf{f},\mathsf{xs}))/\mathsf{sub} \Rightarrow \mathrm{RHS}$

⇐ {*Lemma A8*}

satE′ $ (prePost f k/(args f, xs)) / sub ⇒ RHS

≡ {*inline* (/) *on LHS*}

satE′ $ prePost f k/(args f, map (/sub) xs) ⇒ RHS

⇐ {*Lemma A5*}

satE′ $ reduce (lit $ Sat (body f) k)/(args f, map (/sub) xs) ⇒ RHS

⇐ {*Lemma A8*}

satE′ $ (lit $ Sat (body f) k) / (args f, map (/sub) xs) ⇒ RHS

≡ {*inline* (/) *on LHS*}

satE′ $ (lit $ Sat (body f / (args f, map (/sub) xs)) k) ⇒ RHS

≡ {*inline* satE′}

satE $ Sat (body f / (args f, map (/sub) xs)) k ⇒ RHS

≡ {*inline* satE}

sat $ Sat (eval $ body f / (args f, map (/sub) xs)) k ⇒
  sat $ Sat (eval $ Call f xs / sub) k

≡ {*inline* (/) *on RHS*}

sat $ Sat (eval $ body f / (args f, map (/sub) xs)) k ⇒
  sat $ Sat (eval $ Call f (map (/sub) xs)) k

Case: e = Call f xs ; f = `"error"`

sat $ Sat (eval $ body f / (args f, map (/sub) xs)) k ⇒
  sat $ Sat (eval $ Call f (map (/sub) xs)) k

≡ {*assume* f = `"error"`}

sat $ Sat (eval $ body `"error"` / (args f, map (/sub) xs)) k ⇒
  sat $ Sat (eval $ Call `"error"` (map (/sub) xs)) k

≡ {*inline* eval *on RHS*}

sat $ Sat (eval $ body `"error"` / (args f, map (/sub) xs)) k ⇒
  sat $ Sat Bottom k

≡ {*inline* sat}

sat $ Sat (eval $ Call `"error"` [Var `"x"`] / ([`"x"`], map (/sub) xs)) k ⇒ False

≡ {*implies*}

True

Case: e = Call f xs ; f ≢ `"error"`

sat $ Sat (eval $ body f / (args f, map (/sub) xs)) k ⇒
  sat $ Sat (eval $ Call f (map (/sub) xs)) k

≡ {*inline* eval *on RHS, assuming* f ≢ `"error"`}

sat $ Sat (eval $ body f / (args f, map (/sub) xs)) k ⇒
  sat $ Sat (eval $ body f / (args f, map (/sub) xs)) k

≡ {*tautology*}

True

Case: e = Case x as

satE′ \$ red e k/sub ⇒ sat \$ Sat (eval \$ e / sub) k
≡ {e = Case x as}
satE′ \$ red (Case x as) k/sub ⇒ sat \$ Sat (eval \$ Case x as / sub) k
≡ {*inline* red}
satE′ \$ andP (map alt as)/sub ⇒ sat \$ Sat (eval \$ Case x as / sub) k
≡ {*inline (/) on LHS*}
satE′ \$ andP \$ map ((/sub) ∘ alt) as ⇒ sat \$ Sat (eval \$ Case x as / sub) k
≡ {*inline* satE′}
all (satE′ ∘ (/sub) ∘ alt) as ⇒ sat \$ Sat (eval \$ Case x as / sub) k
≡ {*inline (/) on RHS*}
all (satE′ ∘ (/sub) ∘ alt) as ⇒ sat \$ Sat (eval \$ Case (x / sub) (as / sub)) k

Case: e = Case x as **;** eval (x / sub) = Bottom

all (satE′ ∘ (/sub) ∘ alt) as ⇒ sat \$ Sat (eval \$ Case (x / sub) (as / sub)) k
≡ {*inline* eval, *assuming* eval (x / sub) = Bottom}
all (satE′ ∘ (/sub) ∘ alt) as ⇒ sat \$ Sat Bottom k
≡ {*inline* sat}
all (satE′ ∘ (/sub) ∘ alt) as ⇒ True
≡ {*implies*}
True

Case: e = Case x as **;** eval (x / sub) = Value c xs

all (satE′ ∘ (/sub) ∘ alt) as ⇒ sat \$ Sat (eval \$ Case (x / sub) (as / sub)) k
≡ {*inline* eval, *assuming* eval (x / sub) = Value c xs}
LHS ⇒ sat \$ Sat (head [eval y | Alt c′ vs y ← as / sub, c ≡ c′]) k
≡ {*expand RHS, removing list comprehension*}
all (satE′ ∘ (/sub) ∘ alt) as ⇒
  all (λ(Alt c′ vs y) → c ≡ c′ ⇒ sat \$ Sat (eval y) k) (as / sub)
⇐ {*lift implication over* all}
satE′ \$ alt a/sub ⇒
  (λ(Alt c′ vs y) → c ≡ c′ ⇒ sat \$ Sat (eval y) k) (a / sub)
≡ {*instantiate* a = Alt c′ vs y}
satE′ \$ alt a/sub ⇒
  (λ(Alt c′ vs y) → c ≡ c′ ⇒ sat \$ Sat (eval y) k) (Alt c′ vs y / sub)
≡ {*inline (/) on RHS*}
satE′ \$ alt a/sub ⇒
  (λ(Alt c′ vs y) → c ≡ c′ ⇒ sat \$ Sat (eval y) k) (Alt c′ vs \$ y / sub)
≡ {*inline lambda on RHS*}
satE′ \$ alt (Alt c′ vs y)/sub ⇒

```
   (c ≡ c′ ⇒ sat $ Sat (eval $ y / sub) k)
≡ {use knowledge from RHS in LHS}
satE′ $ alt (Alt c vs y)/sub ⇒ sat $ Sat (eval $ y / sub) k
≡ {inline alt on LHS}
satE′ $ (reduce (x⋖(ctors c \ [c])) ∨ red y k)/sub ⇒ RHS
≡ {inline (/) on LHS}
satE′ $ (reduce (x⋖(ctors c \ [c]))/sub) ∨ (red y k/sub) ⇒ RHS
≡ {inline satE′ on LHS}
satE′ (reduce (x⋖(ctors c \ [c]))/sub) || satE′ (red y k/sub) ⇒ RHS
⇐ {Lemma A8}
satE′ ((x⋖(ctors c \ [c])) / sub) || satE′ (red y k/sub) ⇒ RHS
≡ {inline (/) on LHS}
satE′ ((x / sub)⋖(ctors c \ [c])) || satE′ (red y k/sub) ⇒ RHS
≡ {Lemma A2}
sat′ (eval (x / sub)⋖(ctors c \ [c])) || satE′ (red y k/sub) ⇒ RHS
≡ {substitute eval (x / sub) = Value c xs}
sat′ (Value c xs⋖(ctors c \ [c])) || satE′ (red y k/sub) ⇒ RHS
⇐ {Lemma C1}
c ∈ (ctors c \ [c]) || satE′ (red y k/sub) ⇒ RHS
≡ {Lemma A1}
False || satE′ (red y k/sub) ⇒ RHS
≡ {inline (||)}
satE′ $ red y k/sub ⇒ sat $ Sat (eval $ y / sub) k
⇐ {by induction}
True
```

Lemma A8

```
satE′ $ reduce x/sub ⇒ satE′ $ x / sub
≡ {inline reduce}
satE′ $ mapP (λ(Sat x k) → red x k) x/sub ⇒ satE′ $ x / sub
≡ {inline (/) on LHS}
satE′ $ mapP (λ(Sat x k) → red x k/sub) x ⇒ satE′ $ x / sub
≡ {inline (/) on RHS}
satE′ $ mapP (λ(Sat x k) → red x k/sub) x ⇒
   satE′ $ mapP (λ(Sat x k) → lit $ Sat x k / sub) x
≡ {inline (/) on RHS}
satE′ $ mapP (λ(Sat x k) → red x k/sub) x ⇒
   satE′ $ mapP (λ(Sat x k) → lit $ Sat (x / sub) k) x
≡ {inline satE′}
isTrue $ mapP (λ(Sat x k) → bool $ satE′ $ red x k/sub) x ⇒
   isTrue $ mapP (λ(Sat x k) → bool $ satE′ $ lit $ Sat (x / sub) k) x
⇐ {lift mapP over (⇒)}
satE′ $ red x k/sub ⇒ satE′ $ lit $ Sat (x / sub) k
```

$\equiv$ {*inline* satE′ *on RHS*}
satE′ \$ red x k/sub $\Rightarrow$ satE \$ Sat (x / sub) k
$\equiv$ {*inline* satE *on RHS*}
satE′ \$ red x k/sub $\Rightarrow$ sat \$ Sat (eval \$ x / sub) k
$\Leftarrow$ {*Lemma A7*}
True

A.6 The Soundness Theorem

A.6.1 Theorem

satE′ \$ pre e $\Rightarrow$ not \$ isBottom \$ eval e

That is, the analysis defined in Figures 6.10, 6.7, 6.8 and 6.3 is sound.

A.6.2 Proof

We proceed by case analysis on the structure of the expression e. We inductively assume that the theorem is true for all subexpressions of e.

Case: e = Var v

We do not need to consider Var as eval cannot be called on expressions with free variables, see §A.3.

Case: e = Sel x (c, i)

By definition:

```
eval (Sel x (c, i)) | c ≡ c′ = xs !! i
   where Value c′ xs = eval x
```

We know that any Sel x _ value must be contained within an alternative of a Case x _ expression. We may assume that the original Case expression satisfied its precondition.

satE′ \$ pre e $\Rightarrow$ not \$ isBottom \$ eval e
$\equiv$ {e = Sel x (c, i)}

satE′ \$ pre \$ Sel x (c, i) ⇒ not \$ isBottom \$ eval \$ Sel x (c, i)
≡ {*inline* eval}
satE′ \$ pre \$ Sel x (c, i) ⇒ not \$ isBottom \$ xs !! i
⇐ {*strengthen implication*}
satE′ \$ pre \$ Sel x (c, i) ⇒ all (not ∘ isBottom) xs
≡ {*by definition of* isBottom}
satE′ \$ pre \$ Sel x (c, i) ⇒ not \$ isBottom \$ Value c′ xs
≡ {Value c′ xs = eval x}
satE′ \$ pre \$ Sel x (c, i) ⇒ not \$ isBottom \$ eval x
⇐ {*inductive hypothesis*}
satE′ \$ pre \$ Sel x (c, i) ⇒ satE′ \$ pre x
≡ {*assuming original* Case *satisfied its constraint*}
satE′ \$ pre \$ Case x as ⇒ satE′ \$ pre x
≡ {*inline* pre}
satE′ \$ pre x ∧ andP (map alt as) ⇒ satE′ \$ pre x
≡ {*inline* satE′}
satE′ (pre x) && satE′ (andP \$ map alt as) ⇒ satE′ \$ pre x
⇐ {*weaken implication*}
satE′ \$ pre x ⇒ satE′ \$ pre x
≡ {*tautology*}
True

Case: e = Make c xs

satE′ \$ pre e ⇒ not \$ isBottom \$ eval e
≡ {e = Make c xs}
satE′ \$ pre \$ Make c xs ⇒ not \$ isBottom \$ eval \$ Make c xs
≡ {*inline* eval}
satE′ \$ pre \$ Make c xs ⇒ not \$ isBottom \$ Value c \$ map eval xs
≡ {*inline* isBottom}
satE′ \$ pre \$ Make c xs ⇒ all (not ∘ isBottom ∘ eval) xs
⇐ {*inductive hypothesis*}
satE′ \$ pre \$ Make c xs ⇒ all (satE′ ∘ pre) xs
≡ {*inline* pre}
satE′ \$ andP \$ map pre xs ⇒ all (satE′ ∘ pre) xs
≡ {*inline* satE′}
and \$ map satE′ \$ map pre xs ⇒ all (satE′ ∘ pre) xs
≡ {map f ∘ map g = map (f ∘ g)}
and \$ map (satE′ ∘ pre) xs ⇒ all (satE′ ∘ pre) xs
≡ {and ∘ map f = all f}
all (satE′ ∘ pre) xs ⇒ all (satE′ ∘ pre) xs
≡ {*tautology*}
True

Case: $\mathsf{e} = \mathsf{Call\ f\ xs}$

$\mathsf{satE'}\ \$\ \mathsf{pre\ e} \Rightarrow \mathsf{not}\ \$\ \mathsf{isBottom}\ \$\ \mathsf{eval\ e}$
$\equiv \{\mathsf{e} = \mathsf{Call\ f\ xs}\}$
$\mathsf{satE'}\ \$\ \mathsf{pre}\ \$\ \mathsf{Call\ f\ xs} \Rightarrow \mathsf{not}\ \$\ \mathsf{isBottom}\ \$\ \mathsf{eval}\ \$\ \mathsf{Call\ f\ xs}$

Case: $\mathsf{e} = \mathsf{Call\ f\ xs}$; $\mathsf{f} = \texttt{"error"}$

$\mathsf{satE'}\ \$\ \mathsf{pre}\ \$\ \mathsf{Call\ f\ xs} \Rightarrow \mathsf{not}\ \$\ \mathsf{isBottom}\ \$\ \mathsf{eval}\ \$\ \mathsf{Call\ f\ xs}$
$\equiv \{\mathsf{f} = \texttt{"error"}\}$
$\mathsf{satE'}\ \$\ \mathsf{pre}\ \$\ \mathsf{Call}\ \texttt{"error"}\ \mathsf{xs} \Rightarrow \mathsf{not}\ \$\ \mathsf{isBottom}\ \$\ \mathsf{eval}\ \$\ \mathsf{Call}\ \texttt{"error"}\ \mathsf{xs}$
$\equiv \{\mathit{inline}\ \mathsf{eval}\}$
$\mathsf{satE'}\ \$\ \mathsf{pre}\ \$\ \mathsf{Call}\ \texttt{"error"}\ \mathsf{xs} \Rightarrow \mathsf{not}\ \$\ \mathsf{isBottom\ Bottom}$
$\equiv \{\mathit{inline}\ \mathsf{isBottom}\}$
$\mathsf{satE'}\ \$\ \mathsf{pre}\ \$\ \mathsf{Call}\ \texttt{"error"}\ \mathsf{xs} \Rightarrow \mathsf{not\ True}$
$\equiv \{\mathit{inline}\ \mathsf{not}\}$
$\mathsf{satE'}\ \$\ \mathsf{pre}\ \$\ \mathsf{Call}\ \texttt{"error"}\ \mathsf{xs} \Rightarrow \mathsf{False}$
$\equiv \{\mathit{implication}\}$
$\mathsf{not}\ \$\ \mathsf{satE'}\ \$\ \mathsf{pre}\ \$\ \mathsf{Call}\ \texttt{"error"}\ \mathsf{xs}$
$\equiv \{\mathit{inline}\ \mathsf{pre}\}$
$\mathsf{not}\ \$\ \mathsf{satE'}\ \$\ (\mathsf{precond}\ \texttt{"error"}/(\mathsf{args}\ \texttt{"error"}, \mathsf{xs}))\ \wedge\ \mathsf{andP}\ (\mathsf{map\ pre\ xs})$
$\equiv \{\mathit{inline}\ \mathsf{satE'}\}$
$\mathsf{not}\ \$\ \mathsf{satE'}\ (\mathsf{precond}\ \texttt{"error"}/(\mathsf{args}\ \texttt{"error"}, \mathsf{xs}))\ \&\&\ \mathsf{all}\ (\mathsf{satE'} \circ \mathsf{pre})\ \mathsf{xs}$
$\equiv \{\mathit{inline}\ \mathsf{not}\}$
$\mathsf{not}\ (\mathsf{satE'}\ (\mathsf{precond}\ \texttt{"error"}/(\mathsf{args}\ \texttt{"error"}, \mathsf{xs})))\ ||\ \mathsf{not}\ (\mathsf{all}\ (\mathsf{satE'} \circ \mathsf{pre})\ \mathsf{xs})$
$\Leftarrow \{\mathit{weaken\ proposition}\}$
$\mathsf{not}\ \$\ \mathsf{satE'}\ (\mathsf{precond}\ \texttt{"error"}/(\mathsf{args}\ \texttt{"error"}, \mathsf{xs}))$
$\equiv \{\mathit{Lemma\ A3}\}$
$\mathsf{not}\ \$\ \mathsf{satE'}\ \$\ (\mathsf{false}/(\mathsf{args}\ \texttt{"error"}, \mathsf{xs}))$
$\equiv \{\mathit{inline}\ (/)\}$
$\mathsf{not}\ \$\ \mathsf{satE'}\ \mathsf{false}$
$\equiv \{\mathit{inline}\ \mathsf{satE'}\}$
$\mathsf{not}\ \$\ \mathsf{False}$
$\equiv \{\mathit{inline}\ \mathsf{not}\}$
True

Case: $\mathsf{e} = \mathsf{Call\ f\ xs}$; $\mathsf{f} \not\equiv \texttt{"error"}$

$\mathsf{satE'}\ \$\ \mathsf{pre}\ \$\ \mathsf{Call\ f\ xs} \Rightarrow \mathsf{not}\ \$\ \mathsf{isBottom}\ \$\ \mathsf{eval}\ \$\ \mathsf{Call\ f\ xs}$
$\equiv \{\mathit{inline}\ \mathsf{eval},\ \mathit{assuming}\ \mathsf{f} \not\equiv \texttt{"error"}\}$
$\mathsf{satE'}\ \$\ \mathsf{pre}\ \$\ \mathsf{Call\ f\ xs} \Rightarrow \mathsf{not}\ \$\ \mathsf{isBottom}\ \$\ \mathsf{eval}\ \$\ \mathsf{body\ f}\ /\ (\mathsf{args\ f}, \mathsf{xs})$
$\Leftarrow \{\mathit{inductive\ hypothesis}\}$
$\mathsf{satE'}\ \$\ \mathsf{pre}\ \$\ \mathsf{Call\ f\ xs} \Rightarrow \mathsf{satE'}\ \$\ \mathsf{pre}\ \$\ \mathsf{body\ f}\ /\ (\mathsf{args\ f}, \mathsf{xs})$

$\equiv$ {*inline pre on LHS*}
satE′ \$ (precond f/(args f, xs)) $\wedge$ andP (map pre xs) $\Rightarrow$ RHS
$\equiv$ {*inline* satE′}
satE′ (precond f/(args f, xs)) && all (satE′ $\circ$ pre) xs $\Rightarrow$ RHS
$\Leftarrow$ {*Lemma A4*}
satE′ ((reduce \$ pre \$ body f)/(args f, xs)) && all (satE′ $\circ$ pre) xs $\Rightarrow$ RHS
$\Leftarrow$ {*Lemma A8*}
satE′ (pre (body f) / (args f, xs)) && all (satE′ $\circ$ pre) xs $\Rightarrow$ RHS
$\Leftarrow$ {*Lemma A6*}
satE′ \$ pre \$ body f / (args f, xs) $\Rightarrow$ satE′ \$ pre \$ body f / (args f, xs)
$\equiv$ {*tautology*}
True

Case: e = Case x as

satE′ \$ pre e $\Rightarrow$ not \$ isBottom \$ eval e
$\equiv$ {e = Case x as}
satE′ \$ pre \$ Case x as $\Rightarrow$ not \$ isBottom \$ eval \$ Case x as

Case: e = Case x as **;** eval x = Bottom

satE′ \$ pre \$ Case x as $\Rightarrow$ not \$ isBottom \$ eval \$ Case x as
$\equiv$ {*inline* eval, *assuming* eval x = Bottom}
satE′ \$ pre \$ Case x as $\Rightarrow$ not \$ isBottom Bottom
$\equiv$ {*inline* isBottom}
satE′ \$ pre \$ Case x as $\Rightarrow$ not True
$\equiv$ {*inline* not}
satE′ \$ pre \$ Case x as $\Rightarrow$ False
$\equiv$ {*implication*}
not \$ satE′ \$ pre \$ Case x as
$\equiv$ {*inline* pre}
not \$ satE′ \$ pre x $\wedge$ andP (map alt as)
$\equiv$ {*inline* satE′}
not \$ satE′ (pre x) && satE′ (andP \$ map alt as)
$\equiv$ {*inline* not}
not (satE′ \$ pre x) || not (satE′ \$ andP \$ map alt as)
$\Leftarrow$ {*weaken condition*}
not \$ satE′ \$ pre x
$\Leftarrow$ {*inductive hypothesis*}
not \$ not \$ isBottom \$ eval e
$\equiv$ {not (not x) = x}
isBottom \$ eval e
$\equiv$ {eval x = Bottom}

isBottom Bottom
$\equiv$ {*inline* isBottom}
True

Case: e = Case x as ; eval x = Value c xs

satE′ \$ pre \$ Case x as $\Rightarrow$ not \$ isBottom \$ eval \$ Case x as
$\equiv$ {*inline* eval, *assuming* eval x = Value c xs}
LHS $\Rightarrow$ not \$ isBottom \$ head [eval y | Alt c′ vs y $\leftarrow$ as, c $\equiv$ c′]
$\Leftarrow$ {*strengthen implication*}
LHS $\Rightarrow$ all (not $\circ$ isBottom) [eval y | Alt c′ vs y $\leftarrow$ as, c $\equiv$ c′]
$\equiv$ {*inline* all *over list comprehension*}
LHS $\Rightarrow$ and [not \$ isBottom \$ eval y | Alt c′ vs y $\leftarrow$ as, c $\equiv$ c′]
$\equiv$ {*rearrange guard as an implication*}
LHS $\Rightarrow$ and [c $\equiv$ c′ $\Rightarrow$ not \$ isBottom \$ eval y | Alt c′ vs y $\leftarrow$ as]
$\equiv$ {*rewrite list comprehension as an* all}
LHS $\Rightarrow$ all (λ(Alt c′ vs y) $\rightarrow$ c $\equiv$ c′ $\Rightarrow$ not \$ isBottom \$ eval y) as
$\Leftarrow$ {*inductive hypothesis*}
LHS $\Rightarrow$ all (λ(Alt c′ vs y) $\rightarrow$ c $\equiv$ c′ $\Rightarrow$ satE′ \$ pre y) as
$\equiv$ {*switch to LHS*}
satE′ \$ pre \$ Case x as $\Rightarrow$ RHS
$\equiv$ {*inline* pre}
satE′ \$ pre x $\wedge$ andP (map alt as) $\Rightarrow$ RHS
$\equiv$ {*inline* satE′}
satE′ (pre x) && all (satE′ $\circ$ alt) as $\Rightarrow$ RHS
$\Leftarrow$ {*weaken implication*}
all (satE′ $\circ$ alt) as $\Rightarrow$
   all (λ(Alt c′ vs y) $\rightarrow$ c $\equiv$ c′ $\Rightarrow$ satE′ \$ pre y) as
$\Leftarrow$ {*lift implies over all*}
satE′ \$ alt a $\Rightarrow$ (λ(Alt c′ vs y) $\rightarrow$ c $\equiv$ c′ $\Rightarrow$ satE′ \$ pre y) a
$\equiv$ {*instantiate* a = Alt c′ v ys}
satE′ \$ alt \$ Alt c′ v ys $\Rightarrow$ (c $\equiv$ c′ $\Rightarrow$ satE′ \$ pre y)
$\equiv$ {*rearrange implication*}
satE′ (alt \$ Alt c′ v ys) && (c $\equiv$ c′) $\Rightarrow$ satE′ \$ pre y
$\equiv$ {*substitute* c $\equiv$ c′}
satE′ \$ alt \$ Alt c v ys $\Rightarrow$ RHS
$\equiv$ {*inline* alt}
satE′ (x$\lessdot$(ctors c \ [c]) $\vee$ pre y) $\Rightarrow$ RHS
$\equiv$ {*inline* satE′}
satE′ (x$\lessdot$(ctors c \ [c])) || satE′ (pre y) $\Rightarrow$ RHS
$\equiv$ {*Lemma A2*}
sat′ (eval x$\lessdot$(ctors c \ [c])) || satE′ (pre y) $\Rightarrow$ RHS
$\equiv$ {eval x = Value c xs}
sat′ (Value c ys$\lessdot$(ctors c \ [c])) || satE′ (pre y) $\Rightarrow$ RHS

$\Leftarrow \{\textit{Lemma C1}\}$
$\mathsf{c} \in (\mathsf{ctors\ c} \setminus [\mathsf{c}]) \mathbin{||} \mathsf{satE}' (\mathsf{pre\ y}) \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{Lemma A1}\}$
$\mathsf{False} \mathbin{||} \mathsf{satE}' (\mathsf{pre\ y}) \Rightarrow \mathrm{RHS}$
$\equiv \{\textit{inline}\ (||)\}$
$\mathsf{satE}'\ \$\ \mathsf{pre\ y} \Rightarrow \mathsf{satE}'\ \$\ \mathsf{pre\ y}$
$\equiv \{\textit{tautology}\}$
True

A.7 Summary

We have outlined an argument that the Catch analysis method presented in Chapter 6 is sound. In doing so, we have characterised the properties that a constraint system must obey, and have shown these properties for BP-constraints and MP-constraints. The majority of the proof uses equational reasoning, apart from Lemma MP2 which has been tested for all small values. The proof has undergone limited checking for simple type-correctness, but has not been fully machine-checked.

Appendix B

Function Index

This chapter provides an index to all the Haskell functions used in the thesis. Some functions are defined in the body of the thesis, while others are taken from the Haskell report (Peyton Jones 2003). Many of the library functions are reproduced in §B.1.

B.1 Library Functions

The following functions are available in the Haskell standard libraries. All the functions are given a type signature, and most contain a possible implementation.

```
module Prelude where

type String = [Char]

data Bool = False | True

not :: Bool → Bool
not x = if x then False else True

(∧), (∨) :: Bool → Bool → Bool
a ∧ b = if a then b else False
a ∨ b = if a then True else b

(∘) :: (β → γ) → (α → β) → α → γ
(∘) f g x = f (g x)

flip :: (α → β → γ) → β → α → γ
flip f a b = f b a

id :: α → α
id x = x

  -- terminate with an error
error :: String → α
```

```
undefined :: α
undefined = error "undefined"

fst :: (α, β) → α
fst (x, y) = x

snd :: (α, β) → β
snd (x, y) = y

const :: α → β → α
const x _ = x

  -- show an item as a string
show :: Show α ⇒ α → String

  -- read an item from a string
read :: Read α ⇒ String → α

  -- read an item, returning all possible parses
reads :: Read α ⇒ String → [(α, String)]
```

```
module Data.List where

data [α] = [ ] | (:) α [α]

length :: [α] → Int
length [ ]      = 0
length (x : xs) = 1 + length xs

map :: (α → β) → [α] → [β]
map f xs = [f x | x ← xs]

foldr :: (α → β → β) → β → [α] → β
foldr f z [ ]       = z
foldr f z (x : xs)  = f x (foldr f z xs)

(++) :: [α] → [α] → [α]
xs ++ ys = foldr (:) ys xs

or, and :: [Bool] → Bool
or   = foldr (∨) False
and  = foldr (∧) True

any, all :: (α → Bool) → [α] → Bool
any f = or   ∘ map f
all  f = and  ∘ map f

null :: [α] → Bool
null [ ]      = True
null (_ : _) = False
```

```
head :: [α] → α
head (x : _) = x

tail :: [α] → [α]
tail (_ : xs) = xs

elem, notElem :: Eq α ⇒ α → [α] → Bool
elem     x = any (≡ x)
notElem x = all  (≢ x)

zipWith :: (α → β → γ) → [α] → [β] → [γ]
zipWith f (x : xs) (y : ys) = f x y : zipWith f xs ys
zipWith f _ _ = [ ]

zip :: [α] → [β] → [(α, β)]
zip = zipWith (, )

lookup :: Eq α ⇒ α → [(α, β)] → Maybe β
lookup key [ ]              = Nothing
lookup key ((x, y) : xys) | key ≡ x   = Just y
                          | otherwise = lookup key xys

iterate :: (α → α) → α → [α]
iterate f x = x : iterate f (f x)

splitAt :: Int → [α] → ([α], [α])
splitAt n xs = (take n xs, drop n xs)

take :: Int → [α] → [α]
take n _ | n ⩽ 0  = [ ]
take _ [ ]        = [ ]
take n (x : xs)   = x : take (n − 1) xs

drop :: Int → [α] → [α]
drop n xs | n ⩽ 0 =  xs
drop _ [ ]        =  [ ]
drop n (x : xs)   =  drop (n − 1) xs

span, break :: (α → Bool) → [α] → ([α], [α])
span p xs = (takeWhile p xs, dropWhile p xs)
break p = span (not ∘ p)

dropWhile :: (α → Bool) → [α] → [α]
dropWhile _ [ ]         = [ ]
dropWhile p (x : xs)
          | p x         = dropWhile p xs
          | otherwise   = x : xs


takeWhile :: (α → Bool) → [α] → [α]
takeWhile _ [ ]          = [ ]
takeWhile p (x : xs)
           | p x       = x : takeWhile p xs
           | otherwise = [ ]
```

```
replicate :: Int → α → [α]
replicate n x = take n (repeat x)
```

```
filter :: (α → Bool) → [α] → [α]
filter p xs = [x | x ← xs, p x]
```

```
concat :: [[α]] → [α]
concat = foldr (++) [ ]
```

```
concatMap :: (α → [β]) → [α] → [β]
concatMap f = concat ∘ map f
```

```
nub :: Eq α ⇒ [α] → [α]
nub [ ]      =   [ ]
nub (x : xs) =   x : nub (filter (≢ x) xs)
```

```
reverse :: [α] → [α]
reverse l = rev l [ ]
  where rev [ ] a = a
        rev (x : xs) a = rev xs (x : a)
```

```
partition :: (α → Bool) → [α] → ([α], [α])
partition p xs = (filter p xs, filter (not ∘ p) xs)
```

```
module Data.Char where

  -- is a character a space
isSpace :: Char → Bool
```

```
module Data.Maybe where

data Maybe α = Nothing | Just α

maybe :: β → (α → β) → Maybe α → β
maybe nothing just Nothing  = nothing
maybe nothing just (Just x) = just x

fromMaybe :: α → Maybe α → α
fromMaybe x = maybe x id

fromJust :: Maybe α → α
fromJust (Just x) = x

isNothing :: Maybe α → Bool
isNothing Nothing = True
isNothing _       = False
```

```
module Control.Monad where

class Monad m where
  (>>=) :: m α → (α → m β) → m β
  (>>) :: m α → m β → m β
  return :: α → m α

class Functor f where
  fmap :: (a → b) → f a → f b

(=<<) :: Monad m ⇒ (α → m β) → m α → m β
(=<<) = flip (>>=)

liftM :: Monad m ⇒ (α → β) → m α → m β
liftM f x = x >>= (return ∘ f)

mapM :: Monad m ⇒ (α → m β) → [α] → m [β]
mapM f = sequence ∘ map f

mapM_ :: Monad m ⇒ (α → m β) → [α] → m ()
mapM_ f = sequence_ ∘ map f

sequence :: Monad m ⇒ [m α] → m [α]
sequence ms = foldr k (return []) ms
  where k m ms = m >>= λx → ms >>= λxs → return (x : xs)

sequence_ :: Monad m ⇒ [m α] → m ()
sequence_ ms = foldr (>>) (return ()) ms


module Control.Monad.State where

newtype State s α = State{runState :: s → (α, s)}
instance Monad (State s)

evalState :: State s α → s → α
evalState s = fst ∘ runState s

get :: State s s
put :: s → State s ()


module Data.Function where

on :: (β → β → γ) → (α → β) → α → α → γ
on g f x y = g (f x) (f y)


module System.IO where

  -- write out a character to the output
putChar :: Char → IO ()
```

putStr :: String $\rightarrow$ IO ()
putStr = mapM_ putChar

putStrLn :: String $\rightarrow$ IO ()
putStrLn x = putStr x $\gg$ putChar `'\n'`

print :: Show $\alpha \Rightarrow \alpha \rightarrow$ IO ()
print = putStrLn $\circ$ show

-- get the command line arguments
getArgs :: IO [String]

-- get the input stream
getContents :: IO String

Bibliography

Stephen Adams. Efficient sets – a balancing act. *JFP*, 3(4):553–561, 1993.

Alex Aiken and Brian Murphy. Static Type Inference in a Dynamically Typed Language. In *Proc. POPL '91*, pages 279–290. ACM Press, 1991.

Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

Adam Bakewell and Colin Runciman. A space semantics for core Haskell. In *Proc. Haskell Workshop 2000*, September 2000.

Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In *Proc. ICFP '97*, pages 25–37. ACM, 1997.

Urban Boquist and Thomas Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In *Proc IFL '96*, volume 1268 of *LNCS*, pages 58–84. Springer-Verlag, 1996.

Björn Bringert and Aarne Ranta. A pattern for almost compositional functions. In *Proc. ICFP '06*, pages 216–226. ACM Press, 2006.

Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *Proc. ESOP '00*, volume 1782 of *LNCS*, pages 56–71. Springer–Verlang, 2000.

Wei-Ngan Chin and John Darlington. A higher-order removal method. *Lisp Symb. Comput.*, 9(4):287–322, 1996.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. ICFP '00*, pages 268–279. ACM Press, 2000.

John Horton Conway. *Regular Algebra and Finite Machines.* London Chapman and Hall, 1971.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proc ICFP '07*, pages 315–326. ACM Press, October 2007a.

Duncan Coutts, Don Stewart, and Roman Leshchinskiy. Rewriting Haskell strings. In *Proc PADL 2007*, pages 50–64. Springer-Verlag, January 2007b.

Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proc. PPDP '01*, pages 162–174. ACM, 2001.

Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. In *Proc. TCS '02*, pages 448–460, Deventer, The Netherlands, 2002.

Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR '06)*, volume 4130 of *LNCS*, pages 281–286. Springer–Verlag, 2006a.

J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated termination analysis for Haskell: From term rewriting to programming languages. In *Proceedings of the 17th International Conference on Rewriting Techniques and Applications (RTA-06)*, volume 4098 of *LNCS*, pages 297–312. Springer–Verlag, 2006b.

Andrew Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In *Proc FPCA '93*, pages 223–232. ACM Press, June 1993.

Dimitry Golubovsky, Neil Mitchell, and Matthew Naylor. Yhc.Core – from Haskell to Core. *The Monad.Reader*, 1(7):45–61, April 2007.

M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A truly functional-logic language. In *ILPS'95 Post Conference Workshop on Declarative Languages for the Future.* Portland State University and ALP, Melbourne University, 1995.

Ralf Hinze. Generics for the masses. In *Proc. ICFP '04*, pages 236–243. ACM Press, 2004. ISBN 1-58113-905-5.

Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. In *Summer School on Generic Programming*, volume 2793 of *LNCS*, pages 1–56. Springer-Verlang, 2003.

Gerard Huet. Functional pearl: The Zipper. *JFP*, 7(5):549–554, September 1997.

John Hughes. A novel representation of lists and its application to the function "reverse". *Inf. Process. Lett.*, 22(3):141–144, 1986.

Graham Hutton and Joel Wright. Calculating an Exceptional Machine. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming volume 5*. Intellect, February 2006. Selected papers from the Fifth Symposium on Trends in Functional Programming, Munich, November 2004.

S. C. Johnson. Lint, a C program checker. Technical Report 65, Bell Laboratories, 1978.

Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proc. FPCA '85*, pages 190–203. Springer-Verlag New York, Inc., 1985.

Mark P. Jones. Dictionary-free Overloading by Partial Evaluation. In *Proc. PEPM '94*, pages 107–117. ACM Press, June 1994.

Mark P. Jones. Type classes with functional dependencies. In *Proc ESOP '00*, volume 1782 of *LNCS*, pages 230–244. Springer-Verlang, 2000.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993.

Peter A. Jonsson and Johan Nordlander. Positive Supercompilation for a higher order call-by-value language. In *Proc. IFL 2007*, September 2007.

J B Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95(2):210–255, 1960.

R. Lämmel and J. Visser. A Strafunski Application Letter. In *Proc. PADL'03*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.

Ralf Lämmel. The sketch of a polymorphic symphony. In *Proc. of International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002)*, volume 70 of *ENTCS*. Elsevier Science, 2002.

Ralf Lämmel. Typed generic traversal with term rewriting strategies. *Journal of Logic and Algebraic Programming*, 54:1–64, 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proc. TLDI '03*, pages 26–37. ACM Press, March 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proc. ICFP '04*, pages 244–255. ACM Press, 2004.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proc. ICFP '05*, pages 204–215. ACM Press, September 2005.

C. Lee. Representation of switching circuits by binary decision diagrams. *Bell System Technical Journal*, 38:985–999, 1959.

Michael Leuschel. Homeomorphic embedding for online termination of symbolic methods. In *The essence of computation: complexity, analysis, transformation*, pages 379–403. Springer-Verlag, 2002.

Tobias Lindahl and Konstantinos Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In *Proc. APLAS '04, LNCS 3302*, pages 91–106. Springer, November 2004.

Fredrik Lindblad, Matthew Naylor, and Colin Runciman. Lazy SmallCheck - project home page. `http://www.cs.york.ac.uk/~mfn/lazysmallcheck/`, October 2007.

Luc Maranget. Warnings for pattern matching. *JFP*, 17(3):1–35, May 2007.

Simon Marlow. *Deforestation for Higher-Order Functional Programs.* PhD thesis, University of Glasgow, 1996.

Simon Marlow and Simon Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *JFP*, 16(4–5):415–449, July 2006.

Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. Faster laziness using dynamic pointer tagging. In *Proc. ICFP '07*, pages 277–288. ACM Press, October 2007.

Conor McBride and James McKinna. The view from the left. *JFP*, 14(1): 69–111, 2004.

Conor McBride and Ross Paterson. Applicative programming with effects. *JFP*, 17(5):1–13, 2007.

John Meacham. jhc: John's haskell compiler. `http://repetae.net/john/computer/jhc/`, 2008.

Lambert G. L. T. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. The MIT Press, May 1997.

Neil Mitchell and Stefan O'Rear. Derive - project home page. `http://www.cs.york.ac.uk/~ndm/derive/`, March 2007.

Neil Mitchell and Colin Runciman. Not all patterns, but enough – an automatic verifier for partial but sufficient pattern matching. In *Proc. Haskell '08*, 2008a.

Neil Mitchell and Colin Runciman. A static checker for safe pattern matching in Haskell. In *Trends in Functional Programming (2005 Symposium)*, volume 6, pages 15–30. Intellect, 2007a.

Neil Mitchell and Colin Runciman. Unfailing Haskell: A static checker for pattern matching. In *Proceedings of the Sixth Symposium on Trends in Functional Programming*, pages 313–328, September 2005.

Neil Mitchell and Colin Runciman. Supercompilation for core Haskell. In *Selected Papers from the Proceedings of IFL 2007*, 2008b. To appear.

Neil Mitchell and Colin Runciman. Supero: Making Haskell faster. In *Proc. IFL 2007*, September 2007b.

Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Proc. Haskell '07*, pages 49–60. ACM, 2007c.

Markus Mohnen. Context patterns in Haskell. In *Implementation of Functional Languages*, pages 41–57. Springer-Verlag, 1996.

Paliath Narendran and Jonathan Stillman. On the complexity of homeomorphic embeddings. Technical Report 87–8, State University of New York at Albany, Albany, NY, USA, March 1987.

Matthew Naylor and Colin Runciman. Finding inputs that reach a target expression. In *Proc. SCAM '07*, pages 133–142. IEEE Computer Society, September 2007.

Matthew Naylor and Colin Runciman. The Reduceron: Widening the von Neumann bottleneck for graph reduction using an FPGA. In *Selected Papers from the Proceedings of IFL 2007*, 2008. To appear.

George Nelan. *Firstification.* PhD thesis, Arizona State University, December 1991.

Will Partain et al. The `nofib` Benchmark Suite of Haskell Programs. `http://darcs.haskell.org/nofib/`, 2008.

Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.

Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, 2002.

Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

Simon Peyton Jones. Call-pattern specialisation for Haskell programs. In *Proc. ICFP '07*, pages 327–337. ACM Press, October 2007.

Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *JFP*, 2(2):127–202, 1992.

Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Proc FPCA '91*, volume 523 of *LNCS*, pages 636–666, Cambridge, Massachussets, USA, August 1991. Springer-Verlag.

Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *JFP*, 12:393–434, July 2002.

Simon Peyton Jones and Andrés Santos. Compilation by transformation in the Glasgow Haskell Compiler. In *Functional Programming Workshops in Computing*, pages 184–204. Springer-Verlag, 1994.

Simon Peyton Jones, Will Partain, and Andre Santos. Let-floating: Moving bindings to give faster programs. In *Proc. ICFP '96*, pages 1–12. ACM Press, 1996.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Proc. Haskell '01*, pages 203–233. ACM Press, 2001.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proc. ICFP '06*, pages 50–61. ACM Press, 2006.

François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. In *Proc. POPL '04*, pages 89–98. ACM Press, 2004.

Deling Ren and Martin Erwig. A generic recursion toolbox for Haskell or: scrap your boilerplate systematically. In *Proc. Haskell '06*, pages 13–24. ACM Press, 2006.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. ACM '72*, pages 717–740. ACM Press, 1972.

Niklas Röjemo. Highlights from nhc – a space-efficient Haskell compiler. In *Proc. FPCA '95*, pages 282–292. ACM Press, 1995.

David Roundy. Darcs: distributed version management in Haskell. In *Proc. Haskell '05*, pages 1–4. ACM Press, 2005.

Jens Peter Secher and Morten Heine B. Sørensen. On perfect supercompilation. In D. Bjørner, M. Broy, and A. Zamulin, editors, *Proceedings of Perspectives of System Informatics*, volume 1755 of *LNCS*, pages 113–127. Springer-Verlag, 2000.

Damien Sereni. Termination analysis and call graph construction for higher-order functional programs. In *Proc. ICFP '07*, pages 71–84. ACM, 2007.

Tim Sheard. Languages of the future. In *Proc. OOPSLA '04*, pages 116–119. ACM Press, 2004.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proc. Haskell Workshop '02*, pages 1–16. ACM Press, 2002.

M. Heine Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479. MIT Press, 1995.

Don Stewart and Spencer Sjanssen. XMonad. In *Proc. Haskell '07*, pages 119–119. ACM Press, 2007.

Jonathan Stillman. *Computational problems in equational theorem proving.* PhD thesis, State University of New York at Albany, Albany, NY, USA, 1989.

The GHC Team. The GHC compiler, version 6.8.2. `http://www.haskell.org/ghc/`, December 2007.

The Yhc Team. The York Haskell Compiler – user manual. `http://www.haskell.org/haskellwiki/Yhc`, February 2007.

Andrew Tolmach. An External Representation for the GHC Core Language. `http://www.haskell.org/ghc/docs/papers/core.ps.gz`, September 2001.

Akihiko Tozawa. Towards Static Type Checking for XSLT. In *Proc. DocEng '01*, pages 18–27. ACM Press, 2001.

V. F. Turchin. *Refal-5, Programming Guide & Reference Manual.* New England Publishing Co., Holyoke, MA, 1989.

V F Turchin. The algorithm of generalization in the supercompiler. In *Partial Evaluation and Mixed Copmutation*, pages 341–353. North-Holland, 1988.

Valentin F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.

Valentin F. Turchin, Robert M. Nirenberg, and Dimitri V. Turchin. Experiments with a supercompiler. In *Proc. LFP '82*, pages 47–55. ACM, 1982. ISBN 0-89791-082-6. doi: http://doi.acm.org/10.1145/800068.802134.

Alan Mathinson Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937.

David Turner. Total Functional Programming. *Journal of Universal Computer Science*, 10(7):751–768, July 2004.

Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, second edition, 1996.

Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 216–238. Spinger-Verlag, June 2004.

Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc ESOP '88*, volume 300 of *LNCS*, pages 344–358. Berlin: Springer-Verlag, 1988.

Philip Wadler. List comprehensions. In Simon Peyton Jones, editor, *Implementation of Functional Programming Languages*. Prentice Hall, 1987.

Philip Wadler. Theorems for free! In *Proc. FPCA '89*, pages 347–359. ACM Press, 1989.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL '89*, pages 60–76. ACM Press, 1989.

Stephanie Weirich. RepLib: a library for derivable type classes. In *Proc. Haskell '06*, pages 1–12. ACM Press, 2006.

Noel Winstanley. Reflections on instance derivation. In *1997 Glasgow Workshop on Functional Programming*. BCS Workshops in Computer Science, September 1997.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proc. POPL '99*, pages 214–227. ACM Press, 1999.

Dana N. Xu. Extended static checking for Haskell. In *Proc. Haskell '06*, pages 48–59. ACM Press, 2006.

Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for Haskell. In *Proc. IFL 2007*, pages 382–399, 2007.