The University of New South Wales

School of Computer Science and Engineering

Language Extension via Dynamically Extensible Compilers

Sean Seefried

PhD Dissertation

June 2006

Supervisor: Dr. Manuel M. T. Chakravarty
Co-supervisor: Dr. Gabriele Keller

Abstract

This dissertation provides the motivation for and evidence in favour of an approach to language extension via dynamic loading of plug-ins. There is a growing realisation that language features are often a superior choice to software libraries for implementing applications. Among the benefits are increased usability, safety and efficiency. Unfortunately, designing and implementing new languages is difficult and time consuming. Thus, reuse of language infrastructure is an attractive implementation avenue. The central question then becomes, what is the best method to extend languages?

Much research has focussed on methods of extension based on using features of the language itself such as macros or reflection. This dissertation focuses on a complementary solution: plug-in compilers. In this approach languages are extended at run-time via dynamic extensions to compilers, called plug-ins. Plug-ins can be used to extend the expressiveness, safety and efficiency of languages. However, a plug-in compiler provides other benefits. Plug-in compilers encourage modularity, lower the barrier of entry to development, and facilitate the distribution and use of experimental language extensions.

This dissertation describes how plug-in support is added, to both the front and back-end of a compiler, and demonstrates their application through a pair of case studies.

Acknowledgements

In the Scsh manual Olin Shivers wrote the prototype for a new, more honest form of acknowledgements. It finished with:

> Oh yes, the *acknowledgements*. I think not. I did it. I did it all, by myself.

Of course he's being unfair[1] but he's hit upon something that magnanimity usually prevents people from saying: ultimately it's up to the PhD candidate as to whether they finish or not.

Doing a PhD has been a rollercoaster ride. At times it was hard not to feel like I was jumping through yet another gigantic hoop, something I'd promised to swear off after my undergraduate studies. But I was lucky, I had a pair of supervisors that showed me the real academic world, actively encouraging me to publish papers, attend conferences, present talks and collaborate with researchers from around the world. I still struggled with feelings that doing a PhD was more about endurance than anything else but slowly these feelings gave way to a perception of myself as an apprentice researcher being carefully guided into the world of academia.

I'd like to offer genuine and heartfelt thanks to all those people who helped me succeed. My greatest thanks go to my supervisors, Manuel Chakravarty and Gabi Keller. Within days of beginning work on my topic I knew I had made the right decision to leave Brisbane and come to Sydney to work with them. They have both been truly inspirational mentors and supportive friends throughout my candidacy. They have also served as role models demonstrating how to achieve the correct balance between theoretical rigour and practical outcomes in research. Also their son, Leon, provided much needed moments of levity and awe inspiring cuteness.

Next I would like to thank my peers. André Pang was the first in the PLS Research Group to become interested in plug-ins in a statically typed functional programming setting. I thank him for allowing me to shamelessly rip off his ideas and extend them. He has also become one of my best friends. Don Stewart, the über-hacker of our group, was also of immeasurable assistance in thrashing out the implications of plug-ins. My work co-evolved with his popular *hs-plugins* library. Thanks also go to Stefan Wehr for his *sref*

[1] Actually he might take issue with that if he ever reads this.

tool which allowed me to include in my thesis code that was executable but also fancily formatted. To Patryk Zardarnowski I offer an apology: I'm sorry I finished writing my dissertation before you. I'd also like to thank, in no particular order, Sarah Webster, Simon Winwood, Sean "Prime" Lee[2], and Roman Leshchinskiy for all the interesting discussions that formed my days as a researcher.

The third chapter of this dissertation is closely based on a paper I published called *Optimising Embedded DSLs using Template Haskell*. Discussions with various researchers from around the world helped improved the presentation of this paper immensely. In no particular order they were Ian Lynagh, Simon Peyton Jones, John O'Donnell, Anthony Sloane, and Nicolas Magaud.

Conal Elliot deserves particular mention. During my undergraduate degree I came across his paper on Pan, a language for functional image synthesis. The sheer beauty of this language and the images one can generate using it was what got me interested in implementing domain specific languages in Haskell in the first place. This interest continued, branching into general questions of language reuse. But it was Pan that got it all started. Thank you for all the great discussions we had via email during the four years I worked on this dissertation.

In keeping with tradition I have reserved thanks for the people outside my academic circle until last. This is because you sustained me in ways different to, but in many ways just as important as my academic peers.

First, my parents. You provided me with love, support (and occasionally money!). On the brief occasions I was able to return to Brisbane I was rejuvinated by your obvious pride in what I was doing. Toby Meadows, believe it or not but you steered me back onto the path, on more than one occasion, when I'd lost all sense of direction. I'll always admire you for your commitment to truth and reason. Philippa Grahame, you journeyed with me for almost two years of my PhD. With your love and support I felt like I could achieve almost anything. I'm only sorry I put you off doing your own postgraduate degree. The giant microbe stuffed toy you gave me, William Butler Yeast, still safeguards my desk and overseas my daily activities.

[2]We couldn't have two Seans in our research group without confusion. He got the name Sean′, which in retrospect sounds a lot cooler. Anyone remember Optimus Prime?

Contents

List of Figures

Chapter 1

Introduction

While, in principle, it is possible to program any application in a Turing complete language some languages are better suited for certain tasks. That is, computational possibility is not the only concern; we are also concerned with expressiveness, safety and efficiency, things which cannot always be provided adequately through software libraries. Within languages, expressiveness is provided through concise, elegant notations while safety is provided through either static checks, which are machine checked as opposed to being done by hand, or dynamic checks, which are tedious to manually implement. Efficiency can be provided through the agency of clever optimisation or run-time support for common implementation techniques. If we accept the aforementioned benefits, we are faced with a conundrum: despite their utility, creating new languages is expensive both in time and effort.

In an effort to mitigate this reality it has been suggested that languages should be designed for extensibility. There are solid grounds for this approach. Established languages are more likely to have high quality implementations, a suite of useful libraries and good tool support such as IDEs, profilers and debuggers. There is also a higher likelihood that they benefit from a well-established programmer community that can be used as a resource to help train novices, give advanced help, and help improve and provide feedback on software.

In this dissertation I tackle the issues arising from such a language implementation approach. I provide one answer to the question of how one should design a language *and* its implementation so that it can be extended with a minimum of extraneous effort. At its heart, this approach is concerned with *reuse*.

The abstract space of solutions can be divided into halves:

- *Add language extension mechanisms to the language.* Here we focus on language features that allow us to define new constructs that have equivalent functionality to constructs that would normally be built into the language standard.

- *Design the compiler to be extensible.* There is always the option of extending a language by modifying the source code of an existing compiler. Yet for some compilers,

given their size and complexity, this can be impractical. Therefore, the basic thrust of this solution is to design extensibility into the compiler from the beginning.

The primary focus of this dissertation is on a specific instance of the latter solution: a dynamically extensible compiler implementation, which I refer to henceforth as a *plug-in compiler*. As we shall see this variant has many features to recommend it over its statically extensible counterpart.

Initially, I investigated a solution based on language extension mechanisms because, in many respects, it is more principled. The results of this research are included in this dissertation for two reasons. First, the techniques developed can be used with little modification in a plug-in compiler and second, the problems encountered during the execution of this approach are interesting in their own right and motivate the implementation of a plug-in compiler.

Plug-in compilers provide the ability to extend languages without requiring the programmer to be intimately familiar with the source code of a particular compiler; all that is necessary is a familiarity with the exposed plug-in API. Consequently, the barriers to contribution are lowered. Simultaneously, the problem of code base forking is mitigated; with plug-ins it should no longer be necessary to fork the development tree of a compiler when testing experimental language extensions, only to be faced with the tediousness of integrating them back into the trunk once they have become well established. Back-end plug-ins also provide a means by which libraries can be distributed with their own optimisations; that is as *active libraries*.

1.1 Contributions of the dissertation

The main contributions of the dissertation are:

- I show how Template Haskell, a compile-time meta-programming extension to Haskell, can be used to optimise user written code, thus allowing for domain specific optimisations to be written for embedded domain specific languages (EDSLs). Benchmarks for a specific EDSL, Pan, are provided. I also provide a comprehensive taxonomy of the problems with this approach.

- I show how back-end plug-in support can be added to a compiler, GHC in our case, and how this functionality can be used to optimise user written code. This approach overcomes many of the shortcomings of the Template Haskell approach while introducing a few of its own. One of these is the loss of portability but a potential solution to this problem is investigated.

- I provide a solution to the well-known *expression problem* in Haskell using type classes and existential types. The solution provides a way to extend data types and functions that operate on those data structures without modifying existing code while respecting separate compilation. Although a solution for Haskell has recently been proposed by Löh and Hinze my solution works well in a plug-in environment. I

also suggest syntactic sugar for extending data types and provide a formal translation from a language augmented with the new syntax to Haskell. This translation could form the basis of an implementation.

- The ability to write front-end plug-ins, which extend the syntax and semantics of a language, is added to a compiler for a Haskell-like language. The approach centres around providing plug-in points within and between each phase of the front-end of the compiler. This allows all aspects of the compiler's functionality to be extended. As a case study I present a plug-in that adds *list comprehensions* to the language.

1.2 Structure of the dissertation

The rest of the dissertation is organised as follows.

Background–Chapter 2. This chapter provides a more detailed motivation for the dissertation. It begins by covering the benefits of language features and the complexity of language implementations. I then present a survey of language extension mechanisms and the pros and cons of this approach. Most importantly the chapter provides an in depth motivation of plug-in compilers.

Optimising EDSLs using Template Haskell–Chapter 3. Pan, an embedded domain specific language, is used as a vehicle to demonstrate an approach to domain specific optimisation using a meta-programming language called Template Haskell. This chapter provides the motivation for studying plug-in compilers.

Plug-in optimisation–Chapter 4. This chapter discusses the addition of back-end plug-ins to a compiler. A plug-in optimisation that improves the quality of user-written Pan code is then presented along with benchmarks proving its efficacy. A real implementation exists for the Glasgow Haskell Compiler (GHC) and it is described. Also, the concept of plug-in DSLs is introduced. Such DSLs make writing plug-ins easier and improve the portability of a plug-in based approach.

Extensible data types–Chapter 5. In order to write front-end plug-ins it is necessary to solve the *expression problem*. A novel solution is provided for the Haskell language. It involves the use of type classes, existential types and recursive dictionaries. A translation from a language augmented with syntactic sugar for extending data types to the Haskell language is also presented. A comparison with other well-known solutions to the expression problem in other languages is then made.

Front-end plug-ins–Chapter 6. With the expression problem now solved I show how front-end plug-in support can be added to a compiler. The design centres around providing plug-in points between and within each phase of the compiler. I then present a simple, but powerful, plug-in that extends a simple functional programming language with list comprehensions. A real implementation exists in the form of

PHRaC, a small compiler written in Haskell, for a Haskell-like language. It makes use of the techniques introduced in Chapter 5 and a discussion of the issues that arose during implementation is provided.

Conclusion–Chapter 7. This chapter summarises the results of the dissertation.

1.3 Source code

The source code is available for all of the implementations described in this dissertation.

The modifications made to GHC to provide back-end plug-in support are available from the *pluggable-1-branch* fork of the GHC CVS repository. They can be browsed online at:

`http://cvs.haskell.org/cgi-bin/cvsweb.cgi/fptools/`

To download and build the pluggable GHC compiler see the instructions at:

`http://www.haskell.org/ghc/docs/latest/html/building/index.html`

The back-end plug-in for the Pan DSL, the implementation of PHRaC and the corresponding list comprehension front-end plug-in are all available as *darcs*[1] repositories at:

`http://www.cse.unsw.edu.au/~sseefried/darcs/`

The names for these projects are *plugpan*, *plug_phrac* and *listcomp_phrac_plugin* respectively.

[1]The revision control system, *darcs*, can be found at `http://abridgegame.org/darcs/`.

Chapter 2

Background

This chapter provides the motivation for choosing a plug-in architecture to build extensible compilers. In turn, they may be used to extend languages or implement domain specific languages (DSLs). The work is founded upon the assumption that more programmers would try their hand at language design were it not for the high barrier to entry. I claim, as others do [82], that writing new languages is an expensive task both in time and effort. The difficulty of language implementation motivates an approach to language construction based on *reuse*. I go on to describe two ways in which language infrastructure can be reused. The first way, via language extension features, relies on a sufficiently powerful host language within which it is possible to define terms which adequately embody new language features. The second way is to extend a compiler. The efficacy of this approach depends greatly on the manner in which the compiler is designed. I argue that a plug-in compiler, a compiler which loads and runs code at program compile time, provides a convenient and powerful mechanism for language extension.

2.1 Why extend languages?

A Turing complete language can express any computable function; thus, with adequate support for I/O, it's possible to do anything in any programming language. However, computational possibility is not the only concern. Languages are also prized for their conciseness, safety, efficiency, and reasoning properties (among other things). Sometimes libraries can only offer so much and new language features become increasingly attractive. A brief summary is provided below.

- **Conciseness**. Concise notations can remove the need to write verbose, idiomatic code. This leads to a reduction in the number of lines that need to be written to express a program.

- **Safety**. Safety relates to static or dynamic checks upon programs to ensure that they do not perform certain types of unwanted behaviour. Features such as type systems, exceptions, garbage collection and assertions all fall under the umbrella of

safety. A domain specific language can further enhance safety by providing checks tailored to a particular application area.

- **Efficiency**. Static guarantees can often be made about code and sometimes lead to optimisation opportunities. Also, support can be provided by the run-time system for efficient programming of certain applications. e.g. light-weight threads for concurrency.

- **Reasoning**. Some languages[1] have elegant algebraic properties which can be exploited by programmers wishing to prove the correctness of their programs. These properties can also be used to improve the efficiency of code.

Developing languages from scratch can be a daunting undertaking. Defining their semantics and implementing them constitute a large part of the difficulty. It can be years until a fully fledged implementation for a language is available. In this section we explain these difficulties in detail, as well as providing further evidence in favour of language reuse.

2.1.1 The complexity of languages

New languages tend to evolve from older ones, a form of intellectual reuse. In the process they may acquire features which may or may not interact well with existing features. Making these new features "play well together" with existing ones constitutes one of the main challenges of programming language design. A precise formal semantics for language features provides a powerful tool for designing and checking the consistency of languages but requires significant investment and is often not undertaken for this reason.

The complexity of languages is reflected in the fact that some in wide use today contain semantic inconsistencies, or at least an inelegant semantics. For instance, Javascript is well-known for its abundance of "gotchas" [24]. While the semantics of the language has been defined [30] and a typing system has been developed for it [92], it is clear that more thought should have been put into whether each feature was sufficiently well-defined before being added to the language.

A good indication of the semantic complexity of a language is to measure the size of the language standard. These can easily run into the hundreds of pages[2] [6, 7]. The current state of the art is that such standards are written almost exclusively in natural language. Despite extensive research nearly all popular non-trivial languages have resisted mathematical formalisation due to their sheer complexity, although it would be remiss to omit a mention of the heroic work done in partially formalising Standard ML [69]. Clearly, the sheer effort required to define a language's semantics is a strong point in favour of their reuse. Also, language design can be seen as a form of intellectual reuse, an exercise in selecting the best features from earlier languages and combining them with novel additions. As far as possible this should also be an exercise in software reuse.

[1] Cryptol [1] is an example.

[2] R5RS, the standards document for Scheme, is a notable exception running for only fifty pages. Much effort was expended in reducing it to this size.

2.1.2 The complexity of implementations

Implementing a language to a level where it can be used productively can take many months if not years. Although interpreters are, in general, easier to write than compilers, either form of implementation requires a significant amount of effort.

Both compilers and interpreters must lex, parse, and analyse source code. The first two of these phases are probably the most well understood. Lexing is usually done using a lexer generator tool which generates code that encode a deterministic finite state automata (DFA) to perform the task. Parsing, on the other hand, can be done using a number of different algorithms. Each algorithm trades speed and/or space for expressiveness or vice versa to some extent[3]. There is a lot of compiler infrastructure surrounding the parser such as the definition of the representation data structure (often an abstract syntax tree), a symbol table, and libraries for name generation, error reporting and graph manipulation.

After parsing comes analysis. This involves checking for errors in the source which cannot be done during parsing, such as checking the scope of variables, and more sophisticated analyses such as type checking, and data flow analysis.

And this is only the front-end. Interpreters must now execute the code directly while compilers must optimise and generate code. Optimisations form a large part of modern compilers and are often sophisticated. As a language develops it is this section of the compiler more than any other that is likely to grow in size and complexity. This phase of the compiler can also be divided into sub-phases in which the program is translated into one or more intermediate forms upon which certain optimisations are particularly effective or easy to specify. e.g. Static Single Assignment (SSA) form [25] or Administrative Normal Form (ANF) [23].

This leaves only code generation. There are several factors which complicate this phase of the compiler. First, there may be multiple architectures that need to be targeted. The disparate nature of machines means that it is often difficult if not impossible to share code generation routines; each platform must be targeted in isolation. Also, many hardware architectures are designed with simplicity of processor design in mind. This often means that functionality that would have been implemented in hardware in the past must now be implemented in software.

Clearly, there is strong evidence that implementing compilers is a lengthy and ongoing task. Fortunately, the need for reuse in the area of compiler implementation is well recognised and has received much attention. One way in which the effort of a compiler implementation is mitigated is through the use of compiler construction tools.

Parser generators are perhaps the most well-known instance of compiler construction tools. They save the work of writing parsers by hand and have been in use for over two decades. They process lexical tokens and specify the *order* in which they should be combined into data structures which represent the structure of the source code. But this is all most of them do. It is still up to the programmer to attach an *action* to each symbol

[3] At the fast but inexpressive end of the spectrum are recursive decent and LL parsers. At the other end are LR and Earley parsers, the latter of which can parse any context free grammar but has a cubic time complexity.

of the grammar. Such an action may, for instance, take two expressions and an operator and combine them into an aggregate expression.

Other compiler construction tools focus on different phases of the compiler. Attribute grammars can be used to generate and synthesise values in the abstract syntax tree. They can be used to write program analyses and even do code generation. Another tool, Stratego/XT [98, 97], provides a language for writing transformations on abstract syntax trees using the paradigm of rewriting strategies.

There are also tools that can create large parts or even all of a programming environment. Among these are (BEG) [35], a back-end generator that translates declarative specifications to code-generators, ELI [44], which can be used to construct an entire compiler and Centaur [18] which can create a programming language environment (editor, compiler, debugger, etc) from a formal specification of a language.

2.1.3 Programming environments and communities

A fully fledged language does not merely consist of an implementation. It is surrounded by a suite of useful tools, often called the *programming environment* and a community of programmers that use the language. The case for re-use is about more than just the effort required to write a compiler.

Among the tools that can improve the productivity of programmers are profilers, debuggers, Integrated Development Environments (IDEs), and syntax highlighting packages. There is likely to be a high degree of correlation between how widely used a language is and how well-developed such tools are[4]. Thus, in choosing to use a particular language, a programmer also benefits from the *inheritance* of its programming environment. However, It should be kept in mind that an extensible language will require extensible tools.

One should also not underestimate the usefulness of a large (and often highly skilled) programming community. They often take years to form but once present are a considerable resource which can be drawn upon for support of one's own software projects. They also provide a platform upon which the future development of the language can rest. When dealing with the complexities of language semantics it can be of enormous benefit to have a large body of interested and intelligent people check the soundness of one's ideas.

Also, such communities can only form around languages which have demonstrated their utility and are likely to continue being relevant into the foreseeable future. There is only a slim likelihood that a one-off DSL will develop a rich community around it.

2.2 What sorts of extension are we interested in?

Before going on to discuss *how* languages may be extended a moment should be taken to discuss the kinds of extension this dissertation is concerned with.

[4]Oliver Steele believes that languages tend to become either tool focused or feature focused [87]. This does not contradict the relationship I postulated above as he is only concerned with well-established languages. Even in feature focused languages there will be some tool support.

I do not use the term *extension* in the sense in which it can be used in other fields (such as language semantics); our extensions do not necessarily increase the expressiveness of the language, they may simply make them less verbose, more elegant, and/or safer to use. I regard as extensions things such as new control flow constructs, additions to the typing system, and syntactic sugar. Extensions can be divided into three broad categories: expressiveness, safety and efficiency. Expressiveness extensions increase the capabilities of a language, often allowing something that would have been idiomatic and verbose to be concisely captured in a suitable syntax. Safety extensions improve the detection of undesirable behaviour in programs or, as in the case of garbage collection method of memory management, remove the need to consider certain behaviours at all. Efficiency extensions are concerned with improving the quality of generated code.

2.3 Methods of language extension

Accepting the arguments for reuse the language designer is now faced with the problem of how to achieve it. In this section I provide a survey of the ways in which a language can be extended. There are three broad categories into which one can divide the methods of language extension: language features for extensions, henceforth known as *extension features*, extensible implementations and external extension (which is not considered in this dissertation).

These methods of language extension are *not* mutually exclusive. There is nothing stopping a language designer from using extension features *and* extending the compiler. In fact a combination of the two may well be the most prudent path.

2.3.1 External extension

This method of extension covers pre- and post-processors. Pre-processors take source code and re-write them to target an existing compiler or interpreter, while post-processors take the output from an existing compiler and transform it some way. Pre-processing is a popular way of extending languages but can end up duplicating a lot of work. The parser for the pre-processor is often only a small extension to the parser for the target language. Naturally, it is possible to find the parser for the target compiler/interpreter and extend that but this means that the programmer must keep this extension synchronised with the target language. Also, if one wishes to include static analysis done by the target compiler/interpreter this must be painstakingly recreated in the pre-processor. Usually, this is omitted which leads to incomprehensible error messages emitted by the target language compiler when errors are made in the source code to be pre-processed.

Pre-processors are powerful but brute force approach to language extension. We do not consider them further in this dissertation.

2.3.2 Extension features

At a glance familiar features such as function definitions and data type declarations do not merit the status of extension feature but a closer inspection reveals no reason why they should be excluded. The essence of an extension feature is that it allows a programmer to imbue a language entity with a semantics expressible using existing language entities. Macros, inheritance, aspects, higher order functions, reflection, and meta-programming can all be perceived in this light. However, we focus on the more powerful features: macros and meta-programming.

Macro systems have been part of LISP and Scheme for some time. They provide a powerful mechanism for defining new syntactic entities in terms of existing ones. In general, one of the major drawbacks they have is that names used inside the definition can be bound to incorrect names in the local scope. Scheme introduced the notion of *hygienic macros* [57] to solve this problem; they ensure that any free variables are looked up in the scope of the macro definition not where it was used.

Meta-programming is another compelling choice for extending languages. This programming technique was first studied by Smith [86] and then by others in the Lisp and Scheme communities [40, 104, 28, 54]. During this period notions of name generation and quasi-quotation were refined. Sheard, Taha and Pasalic [90, 91, 85, 76] introduced reflection into a strongly typed setting thus further increasing the safety of this powerful programming technique. Features such as extensible syntax and GHC's rewrite rules fall under the umbrella of meta-programming.

Sometimes extension features are entirely adequate for defining the primitive constructs of an entire language. That is, it is possible to define distinct, syntactic entities that have a one-to-one correspondence with semantics of entities in another language. Embedded domain specific languages (EDSLs) are implemented as libraries in precisely this manner. The richer the *host language* the more opportunity there is for this technique to be used[5].

Meta-programming does have its limitations. It is constrained by the degree to which one chooses safety over power. Also, the richer a language is syntactically, the more tedious it is to write transformations using meta-programming since many syntactic cases need to be considered.

Chapter 3 of this dissertation is concerned with the application of a compile-time meta-programming language, Template Haskell, to an EDSL for image synthesis known as Pan [32].

2.3.3 Extending the compiler

At its simplest this method of extension involves modifying the source code of a compiler, so-called static compiler extension. However, a compiler can also be extended at run-time via plug-ins; this is called dynamic compiler extension.

For this to be an effective method of reuse the compiler needs to be designed for

[5] Examples of EDSLs are abundant and include Pan [32], HaXML [103], Fran [31, 34, 22], FranTk [83], FAL [49], Haskore [51], and Yampa [50, 71].

extensibility. Extensibility depends on at least the following factors: the implementation language, orthogonality of the implementation, coupling between modules, and the data structures chosen. The choice of implementation language is probably the most important of all. Features such as inheritance and extensible data structures have a clear utility.

The continuous scale of extensible implementations can be partitioned into three categories: rigid, statically extensible, and dynamically extensible. Each is an evolution of the one before. In this dissertation the term *rigid* is used to describe language implementations that are hard to extend (for whatever reason). Although such implementations are often monolithic, I use the term monolithic to describe applications which are compiled to a single executable and do not load in plug-ins. A monolithic compiler could nevertheless be quite extensible if designed well.

From rigidity to dynamic extensibility

If a programmer wishes to extend a language there is nothing stopping them from modifying the source code of a compiler. As has already been stated, this could be difficult. Production quality compilers are often hundreds of thousands of lines of code, composed of modules written by many people and of differing quality. But this is not the only hindrance. As soon as the modifications have been made, and assuming your extension is not accepted by the mainstream, the implementer has locked themselves into one of two possibilities: either the new version of the compiler must be developed in isolation from the old version, never to be merged, or worse, new developments in the old version must be periodically and tediously merged into the new version. Despite these difficulties the lack of alternatives often drives people to extend languages in just this way.

A statically extensible compiler addresses the problem of badly designed, inextensible code. Examples of such compilers include Polyglot [6] [72], Pizza [108] and Cetus [64].

They are designed with extension in mind from the beginning. If designed correctly the problem of keeping variants in step with the original code base is ameliorated. However, different versions of compilers built in this way must still be separately installed and used on one's system. Only a small step in functionality separates statically extensible compilers from their dynamically extensible counterparts known as *plug-in compilers*: the ability to load and run plug-ins at run-time.

However, this small boost in functionality opens up a large array of opportunities not possible before. Describing these opportunities is the focus of the next section.

2.4 Plug-in compilers

Plug-in compilers are the central focus of this dissertation. Puzzlingly, only minimal attention [82, 13, 36, 106] has been paid to the idea of a *plug-in compiler* even though the utility of plug-ins is widely accepted in other applications. These include Firefox [2], the GIMP [9], Adobe Photoshop, Winamp [10], and Emacs [3]. The essential idea is to

[6] abc [17], a compiler for AspectJ, is an extension of this.

expose the internals of the compiler through a rigorously defined API and to provide entry points or *hooks* into the phases of the compiler. User-written plug-ins can be compiled separately, dynamically loaded at compiler run-time (i.e. program compile-time) and run as part of the compiler.

However, there are frameworks which provide functionality that overlaps to high degree with that of plug-in compilers. An example of this is the CoSy compiler framework [15, 39] which provides a principled way to write and combine stand-alone compiler units into sophisticated compilers. (It is particularly useful for writing complex optimisation suites.) Its architecture centers around *engines* which have access to a central, shared intermediate representation of the program. The engines can be composed, and ordered using an Engine Description Language (EDL). More about CoSy and its similarity to the work presented in this dissertation is discussed in Chapter 4.

2.4.1 Advantages

Plug-ins can fundamentally change the way in which compilers are developed and maintained. The next few sections explain some of the manifold benefits of this approach to compiler construction.

Complexity

One of the most daunting aspects of studying the source code for a compiler is its sheer size. A plug-in compiler encourages the development of a minimal core compiler that is then extended with functionality via plug-ins. This leads to an implementation which is *loosely coupled*, in which module boundaries more accurately reflect functional boundaries. Conversely, the current state of the art is beset by the problem of *cross-cutting* features. For instance, the implementation of syntactic sugar in a traditional compiler cuts across the lexing, parsing, analysis, type checking and sometimes even the code generation phases of the compiler. Although it is too early to say with certainty, the ability to separate the core compiler from a larger, fully-functional language implementation may lead to an application that is less complex and easier to understand. This in turn could reduce development times and improve maintainability.

Safety and Verification

Plug-ins encourage modularity. This reduces the chances that compiler bugs remain undetected by making it possible to test the compiler in layers. The core compiler can be thoroughly tested before plug-ins are even considered.

A plug-in architecture may be of some assistance in achieving the dream of *compiler verification*. Much of the difficulty associated with proving correctness is that there has simply been too much extraneous detail present in the compiler. By moving some of this detail out into plug-ins, the process of verification may be simplified.

Distribution

Plug-ins allow language extensions to be distributed as libraries; hence they can be used as a method of implementing active libraries [96, 26, 95]. Not only does this make their distribution easier, it also fundamentally changes the *marketability* of extensions. For instance, consider the development of a new optimisation technique. The current state of affairs is that an optimisation is either included in a compiler or not at all. Optimisation writers must show that their techniques are *generally* applicable, that they improve the performance of a wide range of programs. If not, they cannot go in the compiler because they may slow down existing programs. However, this misses the fact that they may be particularly effective on a specific set of programs. Peyton Jones et al. [78] eloquently phrases this:

> Compiler optimisations are like therapeutic drugs. Some, like antibiotics, are effective on many programs; such optimisations tend to be built into a compiler. Others are targeted at particular "diseases", on which they are devastatingly effective, but have no effect at all on most other programs.

Plug-ins allow such "niche market" optimisations to exist and perhaps even flourish. They can be used precisely where they are useful. A large collection of only-sometimes-used optimisations leads to the idea of *profile driven optimisation* in which aggressive profiling is used to decide which optimisations are used on a particular piece of code. This is similar to the technique used in the FFTW library [41].

A similar reasoning applies to language extensions, purely syntactic or otherwise. Features must be generally useful and more importantly non-disruptive to the entire language community if they are to go into a compiler. At the moment such non-standard extensions are baked into the compiler and enabled with command-line flags. However, having little used extensions remain inside the compiler bloats it unnecessarily. It is semantically cleaner to realise them as plug-ins.

Development

A language succeeds or fails based on whether it can acquire and sustain the attention of a large group of programmers. Plug-in compilers motivate development on a language by lowering the barrier to entry. No longer is it necessary to understand the intricate details of the inside of the entire compiler in order to write extensions.

In this way the work of maintaining a language's vibrancy is distributed more evenly among its users[7]. This can only help improve its chances of survival.

Plug-ins can also be used to help develop the compiler via a process of bootstrapping. Plug-ins can be used to create DSLs useful for writing key phases of the compiler such as parsing, type checking or code generation. These DSLs can then be used inside the core

[7] And means that the probability of a "bus error" is reduced. "Bus error" is a comic term describing the situation where an application's development is halted due to unfortunate circumstances befalling one of its key maintainers.

compiler source code. Henceforth we call such languages *plug-in DSLs*. (In Section 2.4.3 we see how plug-in DSLs can be used to mitigate a common criticism of plug-in compilers.)

Maintenance

We have already mentioned that plug-in compilers encourage stricter interfaces which, in turn, encourages modularity. This is a simple consequence of the fact that it is hard to change the exposed API. Hence, increased attention must be paid to the interface.

However, a plug-in compiler also benefits the development of and maintenance of language variants; language implementations that add a key feature such as concurrency, parallelism, or bounds checking to an existing language. Plug-in compilers should allow such variants to stay in step with each other.

This point is illustrated by considering the bounds checking extension for the C language[8]. It is implemented for the GNU Compiler Collection (GCC) as a collection of patches, one for each version of the compiler. The maintenance of this extension would be alleviated if it were possible to distribute it as a plug-in.

Flexibility

Plug-in compilers need not even be restricted to one language. A core compiler could implement a small, but rich intermediate language suitable for implementing the features of a range of different languages.

2.4.2 Classification and applications of plug-ins

In this dissertation compiler plug-ins are classified in two ways: front-end vs. back-end and lengthening vs. widening.

A front-end plug-in is primarily concerned with extending expressiveness and safety of languages while a back-end plug-in is concerned with either optimisation or code generation.

A widening plug-in is one which adds to the functionality of existing phases of the compiler. A plug-in which added new syntactic sugar would be such a plug-in. A lengthening plug-in introduces a new phase to the compiler. For instance, it may translate one intermediate language into a second, new intermediate language, perform optimisations on that representation and then translate it back into the old representation. Lengthening plug-in points exist between compiler phases; they take the output of the previous phase and produce the input for the following one. Figure 2.1 presents the classifications graphically.

Plug-ins for expressiveness and safety

Both the expressiveness and the safety features of a language are implemented within the front-end of a compiler. Accordingly these facets of a language are extended via front-end

[8]The patches are available at `http://gcc.gnu.org/extensions.html`

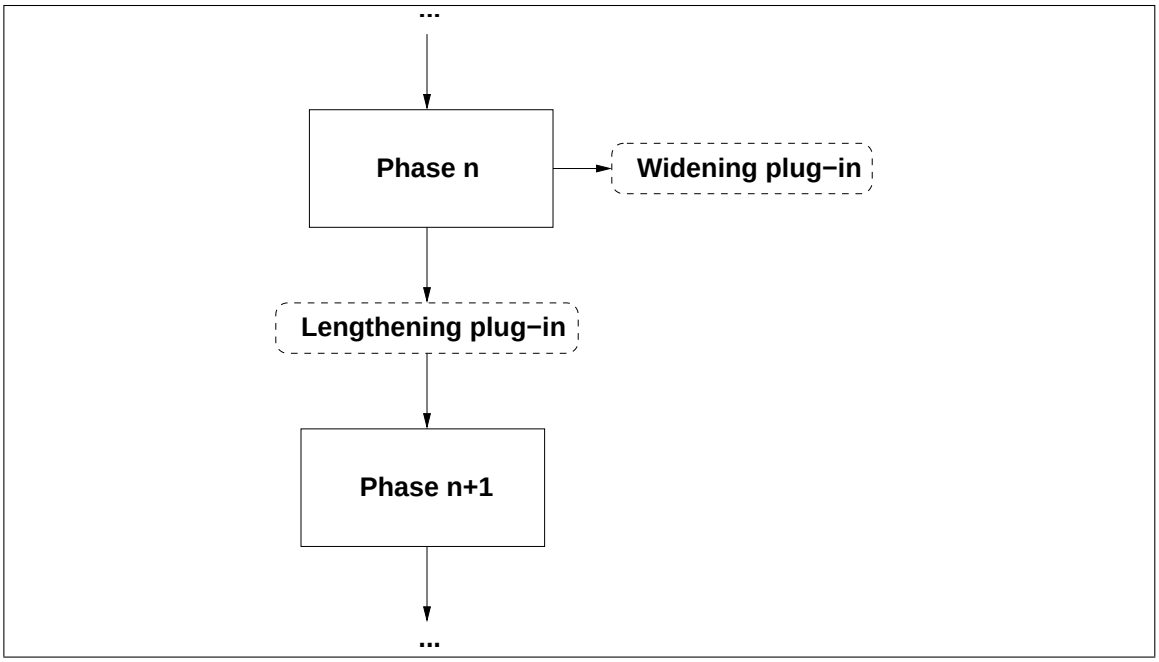

Figure 2.1: The difference between lengthening and widening plug-ins

plug-ins. The applications of these plug-ins are limited only by the language designers imagination. A sample of the sorts of things that can be done are as follows:

- **Syntactic sugar**. The term *syntactic sugar* describes additions to the syntax of a language which do not affect its expressiveness but make it "sweeter" for humans to use. Such syntax is "desugared" into existing language constructs. Such a plug-in involves extending the parser, extending the abstract syntax, and the addition of a desugaring phase. We consider just such a plug-in in Chapter 6.

- **New language features**. This plug-in is identical to the one above except that it genuinely increases the expressiveness of the language. It may need to be used in conjunction with a back-end plug-in to generate new forms of code.

- **Error messages**. This form of plug-in extends the language with custom error messages, perhaps for, but not limited to, domain specific syntactic extensions.

- **New safety checks**. These can be either static or dynamic. Front-end plug-ins can be used to implement static checks. Examples include new kinds of static analysis and checks for erroneous conditions not covered by the type system of the language being extended. Back-end plug-ins can be used to instrument generated code with run-time checks.

Plug-ins for optimisation and code-generation

Back-end plug-ins can be used to provide custom optimisations for DSLs and, in particular, EDSLs. They can also be provided for libraries making them into *active libraries* [96, 26, 95]. In Chapter 4 we consider plug-in optimisations for the Pan EDSL [32].

2.4.3 Common criticisms of plug-in compilers

In this section I respond to some common criticisms of plug-in compilers.

Portability

Robinson states in a paper [82] on the economics of compiler optimisations:

> [Writing plug-ins] requires detailed knowledge of the compiler's internal representation of a program, which is often strewn with decorations that must be maintained for the sake of other optimizations.

This valid concern can be mitigated by careful attention to the API that is exposed. However, an even more effective way is to re-introduce portability through the implementation of DSLs for writing plug-ins. Once it has become standard practice to implement extensions via these DSLs the same forces—both social and technical—that cause widely used languages to be standardised will inevitably extend themselves to these DSLs.

As an example, the CoSy compiler framework has a number of DSLs that are used to write its compiler units. EDL has already been mentioned. It can be used to specify dependencies between compiler units. There is also a structure definition language called fSDL that can be used to specify visibility, side effects and data structures used by the compiler units, that is, the interface between compiler units. This greatly clarifies the relationship between compiler units.

Language Standards

The second common criticism of plug-in compilers is one of language volatility. The fear is that the barrier to the modification of languages will be so low that language variants will flourish to such an extent that the notion of language standards will suffer.

I respond to this criticism in two ways. First, the choice of more language variants in no way diminishes any of the benefits of standardised languages. When the development of an application in a standard variant of a language is perceived to be of great enough value it can still occur.

Second, there is a perception in the programming community that some language standards have become too rigid. This perception is reinforced by examining the time between the release of versions of well-known language standards. For instance, there was a nine year gap between the C89 [5] and C99 [6] standards.

Rather than damaging the notion of standards I envision that plug-ins will induce a *hierarchy* of rigidity in standards. The most rigid standard would be the core language. Presumably after that would come a language with standard syntactic sugar added, followed by various user extensions to the language, and so on. It would be up to users of the language and its extensions to decide how rigid they required their standards to be. In turn this may mean that the success or failure of a language extension is determined by natural selection (of its users), offering a refreshing alternative to the often criticised style of development known as "design by committee".

2.4.4 Limitations of plug-in compilers

It is tempting to think that plug-in compilers allow one to extend languages in arbitrary ways. Sadly this is not the case. The utility of plug-ins depends on two things: how orthogonal the extensions are to existing features and how well the compiler is modularised.

For instance, a programmer may discover that a facet of a language is really a special case of a much more general and powerful formalism. This frequently occurs when extending the type systems of functional programming languages. This is an example of a non-orthogonal change and in this situation a plug-in compiler is not much help. The only sensible approach is to refactor the design of the compiler.

The focus in this dissertation is on extensions that can be made within the existing framework of a compiler.

2.5 Meta-programming vs. plug-in compilers

Meta-programming is a relatively principled approach to language extension. However, meta-programming languages are often not expressive enough to perform the kinds of extensions we want. On the other hand, approaching the problem using plug-ins is more powerful but not nearly as safe. Safety can be increased by carefully choosing which portions of the compiler to expose.

The two approaches are complementary. Both approaches converge on an optimum point which has both adequate expressiveness and safety. The development of meta-programming languages moves from low expressiveness but high safety towards the optimum while the development of plug-in compilers moves from a position of (perhaps excessive) expressiveness and low safety toward the optimum. However, the effort required to add features to a meta-programming language is greater than that required to expose less of a plug-in compiler via the API. Ultimately, the techniques used in plug-in compilers should be used to inform the design of meta-programming languages.

Chapter 3 is an experiment in the application of meta-programming for the optimisation of an EDSL. The chapter concludes with several reasons why the meta-programming language used, Template Haskell, is inadequate for the task. This provides the motivation to switch to the plug-in compiler solution which is described in the remaining chapters of the dissertation.

2.6 Why Haskell?

The results contained within this dissertation can be implemented in any language with sufficient expressiveness. However, almost all code that appears in the remainder will be written in Haskell. The reasons for this are:

- Haskell has excellent support for data structures. They are easy to construct and in conjunction with Haskell's support for pattern matching it is easy to de-construct and manipulate them. Memory allocation and memory layout are handled entirely

by the language so that the programmer is shielded from space leaks and complicated pointer manipulation. Since a large portion of any compiler is concerned with the handling of abstract syntax trees Haskell significantly simplifies their implementation. Languages from the ML family confer the same benefit.

- In Chapter 3, Template Haskell is used to optimise an embedded domain specific language. This language is the culmination of many years of research into functional meta-programming languages [90, 91, 84, 85]. Similar in power to C++ templates [94] it is nevertheless more principled with support for quasi-quotation, automatic handling of scoping and name generation, and the algebraic manipulation of code. Even though I abandon the meta-programming approach after Chapter 3, techniques such as the algebraic manipulation of abstract syntax and the use of monadic code to adequately handle name generation, can be used in plug-ins.

- As mentioned earlier, some domain specific languages can be *embedded* as libraries within another, albeit without the advantages of domain specific optimisations and error messages. The more expressive the host language the more applicable this technique becomes. Haskell's syntactic control features, laziness, higher order functions and rich type system make it an ideal candidate for EDSL implementation. These features will be further discussed in Section 3.1.1.

2.7 Further related work

Each chapter of this dissertation is self-contained in the sense that they draw on relatively disparate areas of programming language research. Hence, the related work shall be deferred until the end of each respective chapter.

Chapter 3

Optimising Embedded DSLs with Template Haskell

In contrast to the remainder of the dissertation this chapter attacks the problem of language reuse using a language extension feature, specifically, compile-time meta-programming. The primary purpose of this chapter is to motivate the plug-in compiler approach by highlighting the problems encountered using a state-of-the-art meta-programming language, Template Haskell, on an embedded domain specific language called Pan. Template Haskell is a sophisticated language incorporating many of the best design choices of earlier meta-programming languages—such as support for quasi-quotation, static and dynamic scoping and convenient manipulation of abstract syntax—and improving upon them with features such as staged static typing.

This approach transforms user-written code according to its syntactic structure. The key idea is to represent code as a data structure (preferably an abstract syntax tree), manipulate this data so that it represents equivalent but faster code, and finally turn this data back into code. It can be thought of as a principled or augmented form of pre-processing. Language support is provided for generation of new symbols, static and dynamic scoping, reification, and a novel staged type-checking algorithm (explained further in Section 3.1.2). Thus, much of the tediousness of writing pre-processors is removed while additional safety is introduced.

Despite Template Haskell's sophistication, it falls short of being a suitable language for library/EDSL optimisation. It turns out that an advantage of meta-programming over plug-ins, namely its compiler independence, is also a problem. Meta-programs have fairly limited access to internal information, which can lead to poor error messages and difficulty in implementing safety features. Of course, we could design meta-programming languages that expose more information, but then we raise similar questions to those raised when designing plug-in compilers. Conversely, this indicates that meta-programming can inform the design of APIs for plug-in compilers and maybe DSLs for compiler plug-ins, something we will come back to in Chapter 4.

The chapter is organised as follows. We begin with a discussion of why Haskell is

a good host language for EDSLs and follow this with a short introduction to Template Haskell describing how it is used and what features make it state-of-the-art. We then introduce the Pan EDSL—which is considered in both this chapter and the subsequent one on plug-in optimisations—and my implementation of it, PanTHeon. We then describe the implementation of three different source level transformations that were implemented in PanTHeon. This is followed by two sets of benchmarks. One proves their efficacy by comparing optimised code against unoptimised code while the other compares PanTHeon against an earlier implementation of Pan by its designers.

An in depth discussion of my experience with Template Haskell follows, describing areas where it falls short of what is required. This provides the background to motivate the plug-in compiler approach.

In summary the contributions of the chapter are:

- The efficacy of compile-time meta-programming is demonstrated on a realistically sized domain specific language, Pan. Three different optimisation techniques are introduced and demonstrated. Benchmarks are provided.
- Drawbacks of the compile-time meta-programming approach are identified an explained.
- This provides the motivation for a plug-in compiler based approach to providing domain specific optimisations.

3.1 Haskell for EDSLs and meta-programming

Before introducing the Pan DSL we focus upon the features of Haskell that make it suitable for implementing EDSLs and then introduce Template Haskell, a compile-time meta-programming extension implemented in the Glasgow Haskell Compiler (GHC).

3.1.1 Haskell: a good host language for EDSLs

Haskell has a number of features that make it ideal for implementing EDSLs.

Laziness

Laziness can be used to provide the same behaviour as control flow constructs in other languages. For instance consider what would happen in the following code if the *if-then-else* construct was strict: $f\ x\ =\ \mathbf{if}\ x\ ==\ 0\ \mathbf{then}\ 1\ \mathbf{else}\ f\ (x-1)$. Using laziness, one can define a function, *ifF*, in Haskell that provides the required behaviour.

$$
\begin{aligned}
&\mathit{ifF}\ ::\ \mathit{Bool}\ \rightarrow\ a\ \rightarrow\ a\ \rightarrow\ a\\
&\mathit{ifF}\ \mathit{True}\ \mathit{thenPart}\ \_ \qquad\qquad = \mathit{thenPart}\\
&\mathit{ifF}\ \mathit{False}\ \_\ \mathit{elsePart} \qquad\qquad = \mathit{elsePart}
\end{aligned}
$$

Syntax control

Haskell has declarations that can change the precedence, fixity and associativity of user defined functions. This allows the syntax of the EDSL to more closely resemble that of a real language. For instance Haskell defines + and ∗ to be left associative with the former having a lower precedence than the latter. This is done with the following declaration in the Prelude.

infixl 7 ∗
infixl 6 +, −

This means that expressions such as $2 * 3 + 7$ parse and evaluate to the correct value, 13. Unlike other languages this is defined in Haskell's libraries, not its compiler.

Higher order functions

Higher order functions allow powerful *combining forms* to be defined that offer the same power as built-in operators in other languages.

Rich type system

Haskell has a rich type system that allows new, mutually recursive algebraic data types to be defined. Used carefully one can disallow certain expressions in the EDSL simply by making sure they lead to type errors in Haskell.

For instance, in a linear algebra setting we can make sure that only vectors can be added to points by defining types and functions as follows.

newtype $Vector = Vec\,(Int,\ Int)$
newtype $Point = Pt\,(Int,\ Int)$
$plus :: Vec \to Pt \to Pt$
$(Vec\,(dx, dy))\ `plus`\ (Pt\,(x, y)) = Pt\,(x + dx,\ y + dy)$

3.1.2 Compile-time meta-programming using Template Haskell

This section introduces Template Haskell. We start by showing how a useful algebraic transformation can be implemented. Since the optimisations of PanTHeon are treated more than adequately in the rest of the chapter we focus on a useful transformation in a well-known domain: linear algebra.

A basic result of linear algebra is that an $n \times n$ matrix, M, multiplied by its inverse, M^{-1}, is equal to the identity matrix, I. This is just the sort of property that we cannot expect most compilers to optimise away, due to the domain-specific knowledge that is required to perform such an optimisation. Consider an expression $m * inverse\ m$ where m is a matrix. (The precise details of how matrices are implemented is immaterial.) In order to simplify this expression it must first be converted from code into a data structure via a process known as *reification* [40].

```
rmMatByInverse (InfixE (Just 'm) 'GHC.Num.* (Just (AppE 'inverse 'm))) =
  VarE (mkName "identity")
rmMatByInverse (LamE pats exp) = LamE pats (rmMatByInverse exp)
rmMatByInverse (AppE exp exp') =
  AppE (rmMatByInverse exp) (rmMatByInverse exp')
rmMatByInverse exp = exp
```

Figure 3.1: An example of an arithmetic transformation.

Once we have verified that this data structure matches the pattern

$$\langle \text{matrix} \rangle * \mathit{inverse}\ \langle \text{matrix} \rangle$$

we can replace it with the data structure that represents *identity*. We then need to convert the data structure back into code, via a process known as *reflection* [86]. This is also known, particularly in Template Haskell, as *splicing*.

The expression, $m * \mathit{inverse}\ m * n$ is reified into an algebraic data type. The transformation of this expression is then simple. A new data structure is created which represents $\mathit{identity} * n$ and spliced. We now show how this is achieved using Template Haskell on a (rather contrived) lambda expression with body equal to $m * \mathit{inverse}\ m * n$.

$$\mathit{exp\_mat} = [|\ \lambda\, m\ n \rightarrow m * \mathit{inverse}\ m * n\ |]$$

exp_mat makes use of the quasi-quote notation of Template Haskell, denoted by the [| and |] brackets. These brackets reify code within.

Figure 3.1 presents the function *rmMatByInverse* which removes the redundancy in the reified expression. The first case does the real work; it matches against infix expressions of the form $\langle \text{matrix} \rangle * \mathit{inverse}\ \langle \text{matrix} \rangle$ and returns *identity*, while the second and third (after matching against expressions of the form $\mathit{lambda}\ p \rightarrow e$ and $f\ a$ respectively) recursively apply to sub-expressions. (Note that we have only presented the cases necessary to transform *exp_mat*.)

Template Haskell's splicing operator, $\$(\ldots)$, runs meta-programs and converts the resulting data structure back to code. In our case the expression $\$(\mathit{rmMatByInverse}\ \mathit{exp\_mat})$ evaluates to the code $\lambda\, m\ n \rightarrow n$ at compile-time. This is a key aspect of our approach; by using a language which is restricted to compile-time meta-computation we guarantee that there is no run-time overhead in the code generated.

Features of Template Haskell

Template Haskell has a number of features that make it particularly well suited to generating and manipulating programs.

First, it respects static scoping. The original paper on Template Haskell presents an example of a meta-program that mimics the behaviour of C's *printf* function by generating

a function which takes the required arguments (of required types) as parameters. It is defined as follows:

```
data Format = D | S | L String

printf :: String → Q Exp
printf s = gen (parse s) [| "" |]

gen :: [Format] → Q Exp → Q Exp
gen [] x = x
gen (D : xs) x = [| λn → $(gen xs [| $x ++ show n |]) |]
gen (S : xs) x = [| λs → $(gen xs [| $x ++ s |]) |]
gen (L s : xs) x = gen xs [| $x ++ $(lift s) |]
```

The n and s bound by lambdas in the third and fourth equations of *gen* are used within the nested quasi quotes. Had a different variable name been used a scoping error would have been signalled by the compiler.

Second, Template Haskell maintains type safety through staged typing. Traditionally, a program is type checked, compiled and then executed. However, we now have execution of code generators occurring at compile-time. In contrast to MetaML, which type-checks all programs before code generation, Template Haskell chooses to interleave type checking with generation. First, the meta-program is type checked at the abstract syntax level. This is, it is determined whether the value produced will be of the type of a *representation* of a declaration, expression, etc. Whether the program represented type checks or not is another question. The meta-program is then executed and the final program type checked as if the programmer had written it in the first place.

Third, quasi-quotation and nested splicing provide a convenient way to generate programs.

Finally, programs can be reified and manipulated using ordinary algebraic data types. Without this capability transformations of the kind demonstrated above could not be done. Also, there are times when the quasi-quote notation is not adequate to generate expressions. For instance, this occurs when the programmer wishes to generate tuples of variable length based on a numeric parameter. In this case algebraic data types can be used to *generate* new expressions.

3.2 Introduction to the Pan DSL

Pan is a domain specific language founded upon the concept of modelling an *image* as a function from continuous Cartesian coordinates to colour values. Images modelled have a number of interesting features. They are:

- *Continuous.* Images are defined as mappings from Cartesian points to colours. This means that they are *resolution independent.* For instance, a circle can be modelled

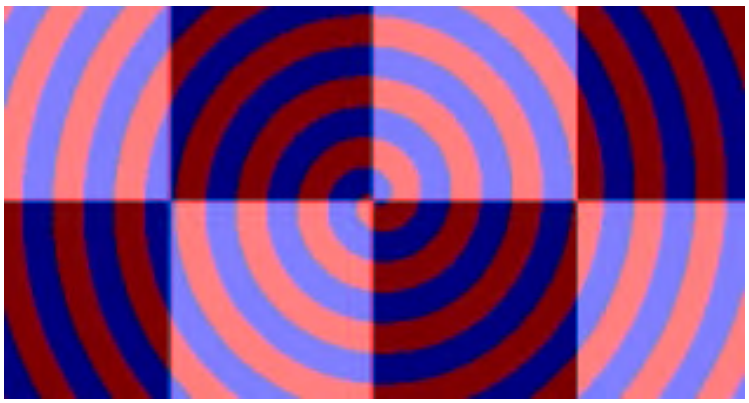

Figure 3.2: Checker board imposed over swirled vertical blue and red stripes

ideally; the circumference will appear smooth at all scales, not jagged as one zooms in.

- *Infinite.* Since the Cartesian domain is effectively infinite (up to the limits of floating point arithmetic) it is easy to define images that are unbounded in extent.
- *Interactive.* We can choose what part of the image to look at and at what scale with no difficulty. Images will not lose resolution as we zoom in, nor are there memory management issues to contend with.
- *Composable.* Images are functions and functions can be composed. Hence it is possible to combine images in arbitrary ways.

3.2.1 A simple image in Pan

Figure 3.2 presents a simple Pan effect that will be used as a running example throughout the rest of the chapter. It is self-contained with respect to Appendix A.

$$
\begin{aligned}
&\mathit{checker}\ (x, y)\ =\ \textbf{if}\ \mathit{even}\ e\ \textbf{then}\ \mathit{blackH}\ \textbf{else}\ \mathit{whiteH}\\
&\quad \textbf{where}\\
&\qquad e\ =\ \mathit{floor}\ x\ +\ \mathit{floor}\ y\\
&\mathit{stripes}\ (x, y)\ =\\
&\quad \mid \mathit{even}\ (\mathit{floor}\ x) = \mathit{blue}\\
&\quad \mid \mathit{otherwise}\ \ \ = \mathit{red}\\
&\mathit{checker\_on\_stripes}\ =\ \mathit{checker}\ `\mathit{over}`\ (\mathit{empty}\ `\mathit{over}`\ \mathit{swirl}\ \mathit{stripes})
\end{aligned}
$$

Colours are represented as four-tuples containing red, green, blue and alpha (transparency) components in the range $[0, 1]$. Functions *whiteH* and *blackH* are 50% transparent. The checker board (*checker*) is defined as a function which takes a coordinate (x, y) and returns *blackH* if $\lfloor x \rfloor + \lfloor y \rfloor$ is even and *whiteH* otherwise. The *stripes* function is even easier to define. Here we simply check that that $\lfloor x \rfloor$ is even and colour it blue if so, red if not.

In *checker_on_stripes* we see the use of the *image overlay* combinator, *over*. This function combines two images pointwise. Depending on the transparency of the top image

a portion of the underlying image will show through. The Pan primitive *swirl* warps an image by rotating points a distance proportional to their distance from the origin. The *empty* image is completely transparent. See Appendix A for their implementation.

3.2.2 The original Pan implementation

The original implementers of Pan decided to implement it as an *embedded compiler* [33] for performance reasons. In this approach the primitives of the DSL were defined as functions over an abstract syntax tree (AST) representation. The ASTs generated by programs written in the host language were then optimised and fed to a code generator which produced efficient code in another language, C. The effort involved was equivalent to writing a compiler back-end and although it was considerably more effort than my implementation, the cost of writing the components of a compiler front-end (such as a lexical analyser, parser, and type checker) was saved. The motivation in choosing to re-implement Pan was to see if equivalent performance could be realised with a direct implementation augmented with compile-time transformations. The results are discussed in Section 3.4.2.

The main disadvantage of embedding a compiler is that access to the (often extensive) general optimisations of the host language compiler are lost. Furthermore, if there is disparity between the host language and the target language, generated programs may not be able to use features of the host language. Finally, even though one ostensibly writes programs in the host language it may not be possible to use language constructs which require a base type of the language. (For instance, for *if-then-else* expressions to be valid they typically require an expression of the *built-in* boolean type.) Unfortunately, the representation of expressions as ASTs requires the use of user-defined types, which precludes the use of such language constructs.

3.3 PanTHeon

PanTHeon is the name given to my implementation of the Pan DSL: a direct implementation augmented with compile-time optimisations. There are three main classes of optimisation: the unboxing of arithmetic expressions, aggressive inlining and algebraic transformations. The unboxing optimisation ensures that arithmetic expressions are evaluated using unboxed arithmetic which provides significant performance gains. Inlining is an effective optimisation for Pan programs which GHC usually fails to do aggressively enough; this is rectified by the aggressive inlining optimisation. Finally, the algebraic transformations attempt to improve performance via algebraic simplification.

In the subsections below I describe why each is particularly applicable to our domain, and its realisation in general meta-programming terms, without recourse to Template Haskell specifics.

3.3.1 Unboxing Arithmetic

Motivation and abstract approach

PanTHeon is a numerically intensive application, almost exclusively using floating-point arithmetic. Hence unboxing can yield significant improvements in speed[1]. Unboxed code also yields better memory locality as the arguments and results do not require an indirection to a heap allocated object. In fact, it is possible that the arguments are placed directly into registers.

Most compilers optimise away as much unnecessary boxing as is feasible, but as implementers of an EDSL we have more knowledge than the compiler does and can consequently do better. We can be certain of the validity of unboxing assuming that every function in PanTHeon is also monomorphic. We lose a bit of Pan's flexibility this way; a much nicer solution would be to specialise each invocation of a polymorphic function based on the type information gleaned from the context in which it is invoked. This is discussed further in the next subsection.

This begs the question, why do we not simply define all the functions in terms of unboxed arithmetic in the first place? Apart from the fact that the syntax of unboxed arithmetic is ugly and cumbersome to use, there is a more important issue: abstraction. When a colour is displayed, each of its component values is converted to an integral value between 0 and 255 and combined into a single 32-bit integer that is placed into video memory. Efficiency can be gained by converting the functions that operate on colours to their integer arithmetic equivalents behind the scenes, while the user retains their view of the current abstraction (i.e. floats of [0,1]). (Although early experiments indicated that this arithmetic conversion measurably improved performance, there were technical reasons that prevented it. The reasons are discussed in the section on implementation details.)

In general terms this optimisation requires that we traverse the representation of each top level function replacing all boxed arithmetic operators and constants with their unboxed equivalents. The unboxing of arithmetic is an interesting transformation as it changes the semantics of the program. Each type in the resulting program corresponds exactly to a type in the original, but it is clear that the validity of this correspondence relies upon our knowledge of the domain.

Implementation in Template Haskell

The process of replacing all boxed operators and constants with their unboxed equivalents is generally a straightforward process in Template Haskell, although we run into difficulty in the context of polymorphic data structures. Most cases written for the family of unboxing functions merely call *unbox* recursively on sub-objects (be they declarations, types, bodies, expressions, etc). There are only a few interesting cases:

[1] Without unboxing, each arithmetic function must first unbox its arguments, perform a primitive arithmetic operation upon these values, and re-box the result.

1. Transforming type signatures. It is clear that any type signatures or type annotations that existed in the original declarations will no longer be valid. For each type synonym and data type declared for the boxed declarations we declare an unboxed version. For ease of recognition the name of such types have a *_UB* suffix appended.

2. Replacing arithmetic operators with unboxed equivalents. This code assumes that all operators will be changed to their unboxed floating point equivalents. We recognise this as an oversimplification and we discuss this further in this chapter.

3. Replacing tuples with stricter versions. We declared two new data types to express points and colours to increase strictness. Unlike built-in tuples, one can add strictness annotations to the arguments of the constructor.

An example covering all three of these cases is the following:

$$
\begin{aligned}
&\mathit{checker} :: \mathit{ImageC}\\
&\mathit{checker}\ (x, y) = \textbf{if}\ \mathit{even}\ e\ \textbf{then}\ \mathit{blackH}\ \textbf{else}\ \mathit{whiteH}\\
&\quad \textbf{where}\\
&\qquad e = \mathit{floor}\ x + \mathit{floor}\ y
\end{aligned}
$$

becomes

$$
\begin{aligned}
&\mathit{checker} :: \mathit{ImageC\_UB}\\
&\mathit{checker}\ (\mathit{Point\_UB}\ x\ y) = \mathit{if}\ \mathit{evenInt\#}\ e\ \mathit{then}\ \mathit{blackH}\ \mathit{else}\ \mathit{whiteH}\\
&\quad \textbf{where}\\
&\qquad e = \mathit{float2Int\#}\ x +\# \mathit{float2Int\#}\ y
\end{aligned}
$$

The main problem with the implementation of the unboxing pass is the lack of easily accessible typing information. It is problematic in three ways.

- as Lynagh [67] has already observed it is impossible to know what the type of a literal is. Fortunately, nearly all literals in the definition of Pan functions are instances of *Fractional*, so we could unbox them to primitive floating point numbers. However there were a few instances where this was not true and special cases had to be written for them.

- In the previous section we discussed converting the components of colours to the range $[0, 255]$. Unfortunately, this would have necessitated a relatively complex transformation on all functions which manipulated colours.

  For instance consider the definition of *cOver* (a key component of the definition of image overlay.)

$$
\begin{aligned}
&\mathit{cOver}\ (r1, g1, b1, a1)\ (r2, g2, b2, a2) = (h\ r1\ r2,\ h\ g1\ g2,\ h\ b1\ b2,\ h\ a1 a2)\\
&\quad \textbf{where}\\
&\qquad h\ x1\ x2 = a1 * x1 + (1 - a1) * x2
\end{aligned}
$$

  Under our proposed transformation it would become

$$cOver\ (Colour\_UB\ r1\ g1\ b1\ a1)\ (Colour\_UB\ r2\ g2\ b2\ a2)\ =$$
$$(Colour\_UB\ (h\ r1\ r2)\ (h\ g1\ g2)\ (h\ b1\ b2)\ (h\ a1\ a2))$$

where

$$h\ x1\ x2\ =\ (a1\ *\#\ x1\ +\#\ (255\#\ -\#\ a1)\ *\#\ x2)\ `divInt\#`\ 255\#$$

Such a transformation is only feasible when one has knowledge of the type of each variable. For instance, in the example above it is necessary to know that $a1$ is of type *ColourBit_UB* (i.e. in range $[0, 255]$).

- I have had to define all PanTHeon functions that contain arithmetic operations monomorphically. A restriction that GHC imposes is that a function containing unboxed operations cannot operate on polymorphic data types. With type information we could specialise such polymorphic functions at each call site.

Without the ability to reify the type of a fragment of an expression, some transformations simply cannot be written for the general case, and until a satisfactory solution has been found, I regard this as one of the principle shortcomings of the implementation. This issue is further discussed in section 3.5.2.

3.3.2 Inlining

Motivation and abstract approach

The style of embedding used in the original implementation of Pan has the effect of inlining all definitions and β-reducing the resulting function applications before any further simplification occurs. This greatly increases the opportunities for algebraic transformation but has the drawback of introducing the possibility of code replication. Fortunately, the effect of code replication can be mitigated by applying a common subexpression elimination (CSE) pass after inlining. Based on the success Elliott, Finne, and de Moor [33] had with it I investigated this approach to code improvement.

However, since GHC has its own passes for performing beta-reduction and CSE, it was decided to leave these passes unimplemented and see how well the compiler performed. The results of the experiment are encouraging and a concrete example is provided in next section.

In general terms the ability to inline code relies upon two meta-programming facilities: the ability to reify, transform and splice code, and the ability to look up the definition of a top-level function declaration. Unfortunately, Template Haskell does not (yet) support this second facility. In Section 3.5.1 I explain a solution to this problem which involves the manual creation of a look-up table.

With this infrastructure in place, the inlining process is relatively straightforward. We take as input the final animation or image function that PanTHeon will display and traverse its definition. Each time we encounter the use of a function that has been defined in PanTHeon[2] we look up its definition, create an equivalent lambda expression and

[2]We do not inline functions that are part of other Haskell libraries.

substitute it at that location. This is done recursively.

Clearly, this leads to non-termination in the presence of recursion. While we could refuse to inline recursive function definitions, determining whether a function is recursive is an involved process requiring the construction of a call graph and the determination of strongly connected components, and in any case GHC already does this. Unfortunately we do not have access to this information. (Perhaps Template Haskell should provide it.) Instead we have chosen to limit the inlining process to a fixed depth which roughly corresponds to loop unrolling in the context of recursion.

Implementation in Template Haskell

Most definitions in the inlining transformation are concerned with traversing the components of a declaration. The function that actually does the real work is *mkInlineExp*. Its implementation is quite cluttered with Template Haskell specifics so we discuss a stepwise example of its effect on our running example (introduced in Section 3.2.1) rather than its definition.

The inlining pass traverses the declaration of *checker_on_stripes* until it comes to the *variable* sub-expression *checker*. At this point a look-up is performed upon its name and the declaration for *checker* is retrieved. We then convert this definition to an equivalent lambda expression. Note that *where* declarations are converted to *let* declarations. Also, without typing information we cannot inline functions that have been overloaded using Haskell's type classes.

$$\begin{aligned}&\lambda\,(x, y)\ \rightarrow\\ &\quad \textbf{let}\ e\ =\ \mathit{floor}\ x\ +\ \mathit{floor}\ y\\ &\quad \textbf{in if}\ \mathit{even}\ e\ \textbf{then}\ \mathit{blackH}\ \textbf{else}\ \mathit{whiteH}\end{aligned}$$

This expression is then substituted in place of the variable. Function definitions that contain guards are also handled. This occurs during the inlining of *stripes*. It is replaced with

$$\lambda\,(x, y)\ \rightarrow\ \textbf{if}\ \mathit{even}\ (\mathit{floor}\ x)\ \textbf{then}\ \mathit{red}\ \textbf{else}\ \mathit{blue}$$

In the previous section I promised a concrete example of the effect of GHC's common subexpression elimination on inlined code. A fitting example to consider comes from the original paper on the implementation of Pan [33]:.

$$\mathit{swirlP}\ r\ =\ \lambda\, p\ \rightarrow\ \mathit{rotate}\ (\mathit{distO}\ p\ *\ (2 * pi/r))\ p$$

The result of inlining clearly contains much redundancy:

$$\begin{aligned}&\lambda\,(x, y)\ \rightarrow\\ &\quad (x\ *\ \mathit{cos}\ (\mathit{sqrt}\ (x * x\ +\ y * y)\ *\ (2 * pi/r))\\ &\quad -\ y\ *\ \mathit{sin}\ (\mathit{sqrt}\ (x * x\ +\ y * y)\ *\ (2 * pi/r)),\\ &\quad y\ *\ \mathit{cos}\ (\mathit{sqrt}\ (x * x\ +\ y * y)\ *\ (2 * pi/r))\\ &\quad +\ x\ *\ \mathit{sin}\ (\mathit{sqrt}\ (x * x\ +\ y * y)\ *\ (2 * pi/r)))\end{aligned}$$

$$
\begin{aligned}
&\mathit{empty}\ `\mathit{over}`\ \mathit{image} = \mathit{image}\\
&\mathit{image}\ `\mathit{over}`\ \mathit{image} = \mathit{image}\\
&\mathit{translate}\ (\mathit{x1}, \mathit{y1})\ (\mathit{translate}\ (\mathit{x2}, \mathit{y2})\ \mathit{im}) = \mathit{translate}\ (\mathit{x1} + \mathit{x2}, \mathit{y1} + \mathit{y2})\ \mathit{im}\\
&\mathit{rotate}\ a\ \mathit{im} = \mathit{rotate}\ (a - n * 2 * \mathit{pi})\ \mathit{im}\ (\textbf{where}\ n = a\ `\mathit{div}`\ 2 * \mathit{pi})\\
&\mathit{rotate}\ \mathit{a1}\ (\mathit{rotate}\ \mathit{a2}\ \mathit{im}) = \mathit{rotate}\ (\mathit{a1} + \mathit{a2})\ \mathit{im}\\
&\mathit{scale}\ (\mathit{x1}, \mathit{y1})\ (\mathit{scale}\ (\mathit{x2}, \mathit{y2})\ \mathit{im}) = \mathit{scale}\ (\mathit{x1} * \mathit{x2}, \mathit{y1} * \mathit{y2})\ \mathit{im}\\
&\mathit{fromPolar}\ (\mathit{toPolar}\ f) = f
\end{aligned}
$$

Figure 3.3: Some algebraic properties of Pan primitives

The following dump of the Core[3] code produced shows that it is capable of removing much of the redundancy.

```
\w_se6i ww_se6l ww_se6m ->
let { a'334 = <core equivalent of x*x + y*y * (2*pi/r)>
} in
 (# (GHC.Prim.minusFloat#
   (GHC.Prim.timesFloat# ww_se6l (GHC.Prim.cosFloat# a'334))
   (GHC.Prim.timesFloat# ww_se6m (GHC.Prim.sinFloat# a'334))),
  (GHC.Prim.plusFloat#
   (GHC.Prim.timesFloat# ww_se6m (GHC.Prim.cosFloat# a'334))
   (GHC.Prim.timesFloat# ww_se6l (GHC.Prim.sinFloat# a'334)))
 #)
```

GHC also performs β-reduction and constant folding (e.g. 2π is replaced with the constant $6.283\ldots$) which saves us yet more implementation effort.

3.3.3 Algebraic Transformation

Motivation and abstract approach

The principle behind algebraic transformation as an optimisation technique is simple: expressions are substituted for semantically equivalent expressions which compile to faster code, be it universally or on average. Considering our running example again, we can see that overlaying the entirely transparent *empty* image on top of *swirl stripes* will have no effect. (This is proved by examining the definition of *over*.)

$$\mathit{checker\_on\_stripes} = \mathit{checker}\ `\mathit{over}`\ (\mathit{empty}\ `\mathit{over}`\ \mathit{swirl}\ \mathit{stripes})$$

The sub-expression may simply be replaced with *swirl stripes*. That is, the following algebraic identity holds: *empty* ‘*over*‘ ⟨image⟩ = ⟨image⟩. (For more examples of algebraic properties of Pan see Figure 3.3.)

What is interesting about our use of this technique in PanTHeon (and in the general context of EDSLs) is that we are using it in a fairly novel context: outside the compiler.

[3] An intermediate representation used by GHC. Adding the flag -dverbose-core2core to the command line will dump the code to standard output.

A key advantage over an embedded compiler is that we only need to implement transformations specific to our EDSL, *extending* rather than overriding the optimisations of the compiler.

In general terms algebraic transformations are easy to implement. For a given expression we attempt to match it against our known algebraic identities. When successful we replace it with the equivalent optimised expression. To ensure that sub-expressions are also optimised we recursively apply to the sub-expressions left unchanged by the original transformation. This is also done when no algebraic transformation is applicable.

Implementation in Template Haskell

Template Haskell's reification of code as algebraic data types in combination with its pattern matching features make algebraic transformations easy to write (and this has been noted by others [27]). Earlier I showed that the expression *empty* ‘*over*‘ ⟨image⟩ can be replaced with ⟨image⟩. This particular case is implemented via the following code.

```
algTrans (AppE (AppE (VarE 'over) (VarE 'empty)) image) =
  algTrans image
```

One of the side effects of the rich syntax offered by modern programming languages, including Haskell, is that there is often more than one way to write essentially the same expression. This is very useful for program *generation* but in the context of program *transformation* means that separate cases must be written to transform equivalent expressions. In order to reduce the number of patterns to be matched against, code needed to be written that translated expressions to a canonical form. For instance, the example above matches on the canonical (prefix) form, *over empty*⟨image⟩, of the algebraic identity presented earlier.

Another tedious aspect of all transformations is the recursive cases. Since we wish our transformations to be applicable not just to expressions but sub-expressions also, we must have cases which recursively call on them. These cases are numerous and easily outnumber the cases that actually do interesting work. However, a well-known paper [60] presents a method by which the such boiler-plate code can be “scrapped”; that is the traversal can be done in a handful of lines of code. These techniques have been used in the source code.

An example of the code reductions are shown below. The function *inlineExp* checks whether an expression is a variable and inlines the appropriate function if so and returns the expression unchanged if not.

The function *inline* is defined using the *everywhereM* combinator. It can be used on any code representation data structure and will transform any component of such data structure that contains an expression, no matter how deeply nested. The function *mkInlinedExp* (not shown here) creates a lambda expression equivalent to the looked up function definition.

```
inlineExp :: [(String, FunDecl)]
          -> ([(String, FunDecl)] -> (forall a. Data a => a -> Q a))
```

```
                → Exp → Q Exp


inlineExp tbl inline e@(VarE nm) =
  case lookup (nameBase nm) tbl of
    (Just (funDec, _)) →mkInlinedExp (inline tbl) funDec
    Nothing            →return e
inlineExp _ _ exp = return exp


inline :: [(String, FunDecl)] → (forall a. Data a ⇒ a → Q a)
inline tbl = everywhereM (mkM (inlineExp tbl inline))
```

3.3.4 Architecture of PanTHeon

PanTHeon consists of three main parts—the language implementation, the optimisation modules and a client for displaying effects. As mentioned above, the language implementation is a direct implementation of the combinators in the original paper on Pan [32]. Users write effects in Pan (which is really just Haskell) which can then be loaded directly into the client via conventional file menu widgets. The user written code is imported into an automatically generated module which transforms their code via functions present in the optimisations modules. This file is then compiled in GHC and dynamically loaded using Don Stewart's *hs-plugins* library [75].

The client program is responsible for displaying images *interactively*. The client chooses a bounding box for the images and then samples the image repeatedly from left to right and then down, writing the resulting colour values to the screen. Using the mouse or keyboard the user of the Pan client is able to pan the image in two dimensions and zoom in or out. The manner in which images are sampled and displayed is discussed further in Section 4.2.2.

3.4 Benchmarks

Performance testing on PanTHeon has been conducted in two ways—optimised effects have been compared with their unoptimised counterparts as well as against the original Pan implementation.

3.4.1 PanTHeon vs. itself

Figure 3.4 compares the frame rate of an effect for which different combinations of optimisations have been applied. When both unboxing and inlining are applied the effects run at least twice as fast, and for one particular example the optimisations led to a nine-fold speed-up.

Effect	Base	Inlined	Unboxed	Unboxed & Inlined
checker_swirl	**8.86 f/s**	1.309x	2.190x	2.258x
circle	**11.241 f/s**	1.324x	2.126x	2.083x
checker_on_stripes	**1.302 f/s**	1.027x	8.361x	9.003x
four_squares	**2.512 f/s**	1.366x	4.184x	4.152x
triball	**1.244 f/s**	1.914x	2.578x	2.707x
tunnel_view	**4.62 f/s**	1.223x	2.042x	2.661x

Figure 3.4: Effect of optimisations on frame rate at resolution of 320x200.

The effects were run on a 1Ghz Apple Powerbook G4 with 512MB of RAM. We have left out the effect of algebraic transformations only because our sample size is so small. Naturally, we could contrive an effect with much computational redundancy which would show off its effectiveness, but this would not tell us much. Only by collecting a large number of effects can anything be said about its effectiveness.

3.4.2 PanTHeon vs. Pan

How well does the performance of PanTHeon compare with that of Pan? Unfortunately, this is difficult to compare because of platform disparity. PanTHeon has been implemented for *nix[4] platforms while Pan only runs on Microsoft Windows. Nevertheless, we performed measurements on PanTHeon and Pan on the same machine: a 733 MHz Pentium III, 384 MB RAM, at 400x300 resolution.

Pan still outperforms PanTHeon. The *checker_swirl* effect has performance that compares favourably; at a resolution of 400x300 it runs at 4.78 *frames/s* in PanTHeon and at 10.61 *frames/s* in Pan. Other effects such as *triball* and *four_squares* perform far better in Pan (> 18 *frames/s*) and very slowly in PanTHeon (1.68 *frames/s* and 6.54 *frames/s* respectively). Both these effects makes substantial use of the *over* primitive for layering images on top of each other. Pan seems to do substantial unrolling of expressions, which is an optimisation we have not yet implemented in PanTHeon. It is suspected that it will significantly improve effects' performance.

However, I have shown that the issue does not lie with any inherent deficiencies in the quality of the code that GHC produces. I hand-coded (and optimised) a very simple effect which ran at a speed comparable to the same effect in Pan (24 million pixels/s).

Another aspect of the Template Haskell implementation that has hindered further tuning or creation of optimisations is the lack of support for profiling of programs which contain splicing.

3.4.3 Relative code base sizes

Finally, the amount of code needed to implement Pan and PanTHeon is compared as a crude means of comparing implementation effort. Both Pan and PanTHeon have two components – a language definition and a display client. The PanTHeon library, plus optimisations totals at about 3000 lines of code. The client is implemented in under 1000

[4]It has been successfully built on Debian GNU/Linux and Mac OS X.

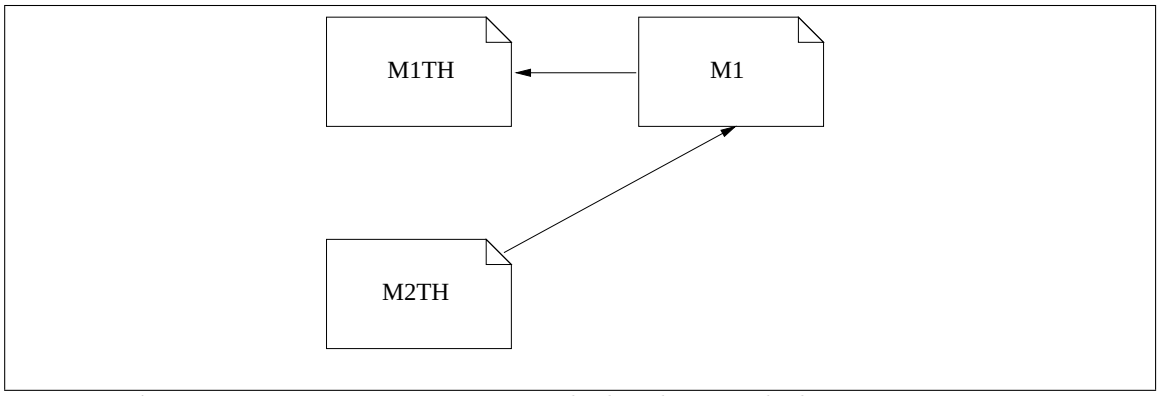

Figure 3.5: *M2TH* imports *M1* which splices in declarations in *M1TH*.

lines of code. Pan, in its entirety, exceeds 13000 lines of code. The inheritance of a code generator and host language optimisations by PanTHeon is the primary reason for this difference.

3.5 Template Haskell specifics

Template Haskell was an excellent language for transformational meta-programming. Its quasi-quote notation, its novel approach to typing and its ability to represent and inspect code made the task of writing elegant compiler-like optimisations possible. However, there are ways in which the language could be improved further and in this section we review difficulties with the current Template Haskell implementation and any solutions that were devised.

3.5.1 Reification of top-level functions

Both unboxing of arithmetic and inlining, require the ability to reify top-level function declarations. Currently, such reification is unsupported in Template Haskell. There is, however, a relatively simple work-around to this problem. We can create a look up table in two steps. First, we place the entire module in declaration reification brackets and call the resulting data structure something appropriate, such as *moduleImageFuns*. We can then create the look-up table for this module by applying a function which creates a list of pairs matching names to function declarations.

An interesting dilemma arises when one wishes to write a module, say *M2TH*, which refers to functions defined in module *M1TH*. The functions in *M1TH* are not in scope and will only become so if they are spliced in another module. So we create a module *M1* which splices in the reified functions from this module and then import *M1*, not *M1TH*, inside module *M2TH*. The basic idea is summarised in Figure 3.5.

There is just one more tiny problem. We wish to transform both the functions in *M1TH* and *M2TH* before bringing them into scope for display in the PanTHeon client, but the solution outlined above causes the functions in *M2TH* to refer to the functions in scope in *M1*. The reified form of such functions contain *original names* which are of the form $M : f$ (where M and f are module and function names respectively). In order

to refer to whatever is in scope at post-transformation splice time we must remove the module prefix from the original names.

The addition to Template Haskell of a native means to deal with the reification of top-level function declarations would greatly simplify the implementation of PanTHeon and similar programs, and would be less error prone.

3.5.2 Lack of type information

In Section 3.3.1 I mentioned that lack of type information prevented satisfactory implementation of the unboxing transformation. This is for three main reasons:

1. We require the type of literals in order to choose the correct primitive unboxed arithmetic functions.

2. Knowing the type that an invocation of a polymorphic function would be instantiated to is also necessary to choose the correct primitive unboxed arithmetic functions.

3. Polymorphic data structures cannot contain unboxed values. Therefore, specialised data structures are required. Again, types are needed. The next subsection discusses this further.

Template Haskell does provide some ability to reify type information by requesting information for a particular variable. Unfortunately, this is only possible if the variable name in question was brought into scope for one of the following reasons: it came from another module, it was not generated by a splice and appears somewhere in the current module, or it was generated by a top-level splice occurring earlier in the current module.

In fact, this is the only type information that *can* be available without splicing the declaration in which the variable appears, for in general it is undecidable as to whether an arbitrary meta-program, once run, will produce correctly typed code. (This was the motivation behind the design of Template Haskell's typing system which defers type checking until meta-programs have been run.)

However, in the special case that the reified code was *closed*, in the sense that it contained no further splices and all variable names were in scope, it is possible (in principle) to type the code. This is precisely the sort of declaration in PanTHeon that we wish to glean type information from.

3.5.3 Unboxing in the context of polymorphic data structures

Section 3.3.1 hinted at a problem with unboxed values in the context of polymorphic data structures. One of the restrictions on unboxed values is that they may not be stored in polymorphic data structures. This necessitates the specialisation of polymorphic data structures to monomorphic counterparts. Disregarding the difficulty of doing this in the absence of typing information there is an additional difficulty. While it is possible to reify data type declarations in other modules (using the Template Haskell primitive *reifyDecl*)

it is not possible to reify the definitions of functions in those modules. The following example illustrates some of the difficulty arising from this.

$$\begin{aligned} &\mathit{weird} :: \mathit{Point} \\ &\mathit{weird} = \mathit{head}\ (\mathit{zipWith}\ (,)\ [1]\ [0.5]) \end{aligned}$$

Without the ability to reify the definitions of *zipWith* and *head*, and specialise them to work on a monomorphic version of the list data type the only other solution is to marshal data to and from unboxed/monomorphic representations at key points within the function definition, which to be feasible also requires access to type information. At present, it is not clear whether the ability to reify entire modules or functions in other modules will be added to Template Haskell or not. The latter solution will be necessary in case it does not.

3.6 Related Work

In his seminal paper Smith [86] introduced a well-defined concept of computational reflection. (Henceforth, this term is used as a synonym for meta-programming.) Reflection was a novel and poorly understood concept when Smith began work on it, even though Lisp had some support for it already. To put reflection on a sound foundation he defined his own dialect called 3-Lisp. 3-Lisp modelled an infinite tower of interpreters; each level of the interpreter was able to scrutinise, manipulate and generate expressions at the level below it. Similar functionality is realised by Template Haskell's support for nested splicing and quasi-quotation.

A host of papers in the Lisp and Scheme communities followed in which simpler descriptions of the tower were presented [40, 28, 54] and a continuation semantics proposed [105]. During this period the notions of name generation and quasi-quotation were refined.

The work of Sheard, Taha and Pasalic [90, 91, 85, 76] introduced meta-programming into a strongly typed setting thus further increasing the safety of this powerful programming technique. The languages they developed, MetaML and then MetaOCaml are quite different to Template Haskell. First, they support run-time meta-programming as well as compile-time. However, for the purposes of optimising libraries and EDSLs we actually prefer the restriction to compile-time. Second, they do not support a notion of reification which means that transformations of the kind presented in this chapter are not possible. However, a technique known as *abstract interpretation* can be used to achieve a similar effect. Lastly, these two languages have powerful but restrictive typing systems. The type of a generated program is reflected in the type of the meta-program that generates it. Unfortunately the type system is so restrictive that it disallows many interesting meta-programs.

MetaML and MetaOCaml are often used to implement DSLs as *staged interpreters* but in quite a different way. Here the essence of the approach is to remove "tagging". Staged interpreters use meta-programming annotations to traverse the representation of the interpreted program before the essence of the program is executed at run-time and amounts to

a form of domain specific partial evaluation. It is an approach based on *generation* rather than *transformation*. This approach also inherits the optimisations of the host language (as they will be applied to the generated code) but introduces the expense of having to implement the front-end of a compiler. If this merely involved the implementation of a simple lexer and parser this expense would be acceptable. Unfortunately real languages often require significant front-end infrastructure such as a symbol table, a complicated abstract syntax tree representation and analysis phases. Figure 3.6 compares the three approaches to implementing DSLs mentioned in this chapter.

Approach	**Inherit front-end**	**Inherit back-end**	**Optimise via**
Embedded compiler (Elliott et al)	yes	no	traditional compiler opts.
Staged interpreter (MetaML)	no	yes	generation (through *delayed expressions*)
Compile-time meta-programming (Template Haskell)	yes	yes	transformation

Figure 3.6: A comparison of three approaches to implementing DSLs

Finally we mention Metaborg. Metaborg [20] is a framework for defining new syntactic forms in a language independent manner. Using this framework one writes transformations using a term rewriting language called Stratego/XT. However, it may be necessary to encode information about the language being transformed in the rewrite rules. This effect is mitigated in Template Haskell which natively supports notions such as syntactic correctness and scoping.

3.7 Towards plug-in compilers

In this chapter I have presented a case study in optimising an EDSL using compile-time meta-programming. Although Template Haskell has performed an admirable job it has not been without its difficulties. The following criticisms apply to all comparable compile-time meta-programming systems.

3.7.1 Problems with compile-time meta-programming

Optimisations occur before compiler's

There is a subtle problem with the interaction between GHC's optimisations and those presented in this dissertation: our optimisations are performed before any of GHC's. Unfortunately the order in which optimisations are done is often important; some increase the effectiveness of others if performed at the correct time.

The only solution to this problem seems to be to provide an API for the optimisation phases. Without a way to name optimisations, either to use an existing one, or add a

new one, one cannot hope to script the optimisation pipeline. However, this seems a little outside the scope of a meta-programming language.

User-level syntax

There are two things one usually wishes to do with meta-programming – generation and transformation. Generation is best done on the user-level syntax of the programming language since syntactic sugar is beneficial for writing concise program generators, not a hindrance. However, transformation is better performed upon an intermediate representation which has few constructs. Otherwise many cases need to be considered in any transformation.

Thus, although sophisticated meta-programming languages provide useful features such as name generation, reification and quasi quotation, it appears that user-level syntax is not always the best level to use them at. The question arises, is transformation of an intermediate representation really meta-programming?

Typing

Another problem arises because of the conflicting goals between program generation and transformation. When *generating* a program one simply cannot always know the type of the final program – it is undecidable in general. Either one restricts the programs that can be generated, like MetaML does, or one implements staged type checking as Template Haskell does.

On the other hand, when *transforming* code one already has the complete program. There is no reason why the full types cannot be provided. Indeed, as mentioned in Section 3.3.1, types are sometimes necessary in order to transform expressions correctly. It would probably be possible to provide two forms of typing to a meta-programming language such as Template Haskell, traditional type inference for fully generated expressions and staged type inference for everything else. However, the question must be asked once again, is this discord in type checking goals indicative of a "fault line" upon which the meta-programming language should be separated? Do we really wish to be transforming user-level code?

3.7.2 The case for plug-in compilers

One could imagine that the first problem above, the lack of type information, could be solved by extending the language. However it is not easy to imagine how a) the notion of mixing optimisations written in Template Haskell with those that reside inside the compiler and b) having those optimisations operate on the intermediate representation of the compiler could be realised within the language. In fact, the natural solution appears to be to abandon the meta-programming approach and to allow a solution based on writing optimisation plug-ins. This approach is explored in the next chapter.

Chapter 4

Plug-in optimisations

In a seminal paper [82] on the economics of compiler optimisations Robinson notes:

> Compile-time program optimizations are similar to poetry: more are written than are actually published in commercial compilers. Hard economic reality is that many interesting optimizations have too narrow an audience to justify their cost in a general-purpose compiler, and custom compilers are too expensive to write.

The results of this chapter provide a means by which this problem can be redressed: back-end plug-ins. They empower programmers to write optimisations without having to recompile the compiler or know everything about its internals. More importantly, they enable custom optimisations to be written for and distributed with libraries, elevating them to the level of *active libraries* [96, 26, 95]. Finally, along with the ability to insert an optimisation at any point within the pipeline of existing ones comes the ability to *script* the optimisation pipeline, providing a level of control over the compiler only approximated by command line flags.

The chapter begins by describing three requirements of any back-end plug-in enabled compiler, followed by a more detailed survey of the design choices available. From among these choices I have chosen an approach based on dynamic loading and linking of plug-ins and retrofitted it into GHC. How this is accomplished and the manner in which the aforementioned requirements are satisfied is then discussed. We continue with a case study involving a custom optimisation for the Pan EDSL. This is done, in part, to compare the techniques of this chapter with the meta-programming approach taken in the previous chapter. Care has been taken to keep this discussion as high-level as possible in order to high-light general lessons learned.

Following the high level presentation of the optimisation more details of the implementation are presented. I describe the implementation of back-end plug-in support GHC in more detail, how the API was implemented, and present key functions of the optimisation in full. I also briefly discuss the implementation of a *plug-in DSL* that can make the writing of optimisations less tedious and error prone. The chapter concludes with some

benchmarks demonstrating the efficacy of the optimisation and a survey of related work.

4.1 Requirements and design alternatives for back-end plug-ins

In this section we describe three important requirements for a back-end plug-in architecture followed by a discussion of design alternatives that satisfy these requirements.

The three requirements are:

- At least one small, elegant and well-understood *intermediate representation* exposed via an API.
- A *compositional scripting* framework for the optimisation pipeline.
- A *communication* and *control flow* framework; a means for information to be passed between plug-ins and from plug-ins to the other phases of the compiler.

4.1.1 Intermediate representations

A good intermediate representation can be difficult to design but once accomplished simplifies the writing of optimisations considerably. This is already true in a monolithic compiler but becomes more important in a plug-in setting since the idea is to lower the barrier of entry for programmers, who may not necessarily be language design experts. A small intermediate representation means there is less to learn. Ease of reasoning is another desirable trait. Two examples of intermediate representations that are small, elegant and easy to reason about are Static Single Assignment (SSA) (notably used in the Low-Level Virtual Machine (LLVM) framework [62]) and the various lambda calculi such as System F and Administrative Normal Form (ANF) [23]. The intermediate representation should also be designed in such a way that it can be annotated with arbitrary information so that, for instance, this can be used by later phases to aid a particular optimisation. In a static setting this means that parametric polymorphism becomes a desirable. (In a dynamic type setting there is no hindrance.)

As an example Figure 4.1 presents an API suitable for compiler written in a functional language. The object language is a typed lambda calculus with support for let expressions, data types and pattern matching. The API is monadic [70, 99, 100] due to the need for fresh name generation. The simplicity of the intermediate representation makes transformations easier. Representations in which it is possible to express the same programs in many different ways are tedious to write optimisations for because of the multitude of different cases that need to be considered.

The most important functionality provided by the API is the ability to look up existing names (both in this module and others), generate new names and manipulate abstract syntax. Some support for generic programming either through libraries [60] or through query and transformation languages such as XPath [11] and XSLT [12].

Abstract syntax representation	**data** *Expr* = *Var Id* | *Abs Id Expr* | *App Expr Expr* | *Con Id* | *Let Id Expr Expr* ...
New names	*newId* :: *String* → *Type* → *M Id*
Look-up body of a binding	*lookupBind* :: *String* → *M* (*Maybe Expr*)
Look-up name	*lookupId* :: *String* → *M Id*
Free variables of expression	*exprFreeVars* :: *Type* → [*Id*]
Representation of types	**data** *Type* = *TyVarTy Name* | *AppTy Type Type* | *TyConApp TyCon* [*Type*] ...
Utilities on types such as	— type of Id *idType* :: *Id* → *Type* — type of expression *exprType* :: *Expr* → *Type* — are two types equal? *eqType* :: *Type* → *Type* → *Bool*

Name is a data type for names. *TyCon* represents type constructors. *M* is a monad with support for name generation; a value of type *M* α represents a computation returning a value of type α.

Figure 4.1: A monadic API for a typed lambda calculus with data types, pattern matching and let-expressions.

4.1.2 Compositional framework for scripting optimisations

Developing an optimisation phase that works well can be difficult. This section covers a number of desirable traits that a framework for scripting optimisations should have. However, before beginning we divide optimisations into three categories.

- *Transformers.* These optimisations do the real work. They transform one program into another that is more desirable, either because it is faster, uses less space, etc.
- *Representation changers.* These plug-ins change a program in one intermediate representation to another intermediate representation. Some representations are better for certain kinds of optimisations hence the need for them.
- *Analysers.* These plug-ins annotate a program with information useful to other plug-in optimisations.

In designing an optimisation pipeline there are many subtle factors that need to be taken into account.

- The order in which optimisations are run can have great effect.
- Some optimisations are beneficial if run several times at different points in the pipeline.
- The efficacy of optimisations varies between architectures. Generate-and-test or automatic profiling techniques may be beneficial.
- There are often strict dependencies between optimisations. Analysers provide information for other optimisations while representation changers must appear before other optimisations that work on the target representation of the changer.

Thus, it is desirable that such a framework allows a back-end plug-in programmer to place existing and newly written optimisations in a pipeline wherever they are allowed to be placed and as many times as necessary. It should not be possible to script the pipeline in a such a way as to violate the dependencies between them. Above all the framework should be *compositional*. It should be possible for optimisations to be combined with others and, in turn, for these amalgamated optimisations to be composed still further. In the extreme, it should not even be necessary for the order of the pipeline to be fixed at plug-in link time. It would be nice if it were possible to write a pipeline that dynamically responded to the software and hardware architecture of the system the compiler is run on.

4.1.3 Communication and control flow framework

Since, by definition, back-end plug-ins are written *after* the compiler there is the issue of how the compiler controls and communicates with the optimisation phase. There are two main possibilities; either the compiler is capable of dynamically loading, linking and running the plug-ins or plug-in optimisations are standalone processes that communicate with the compiler. Figure 4.2 shows the two alternatives.

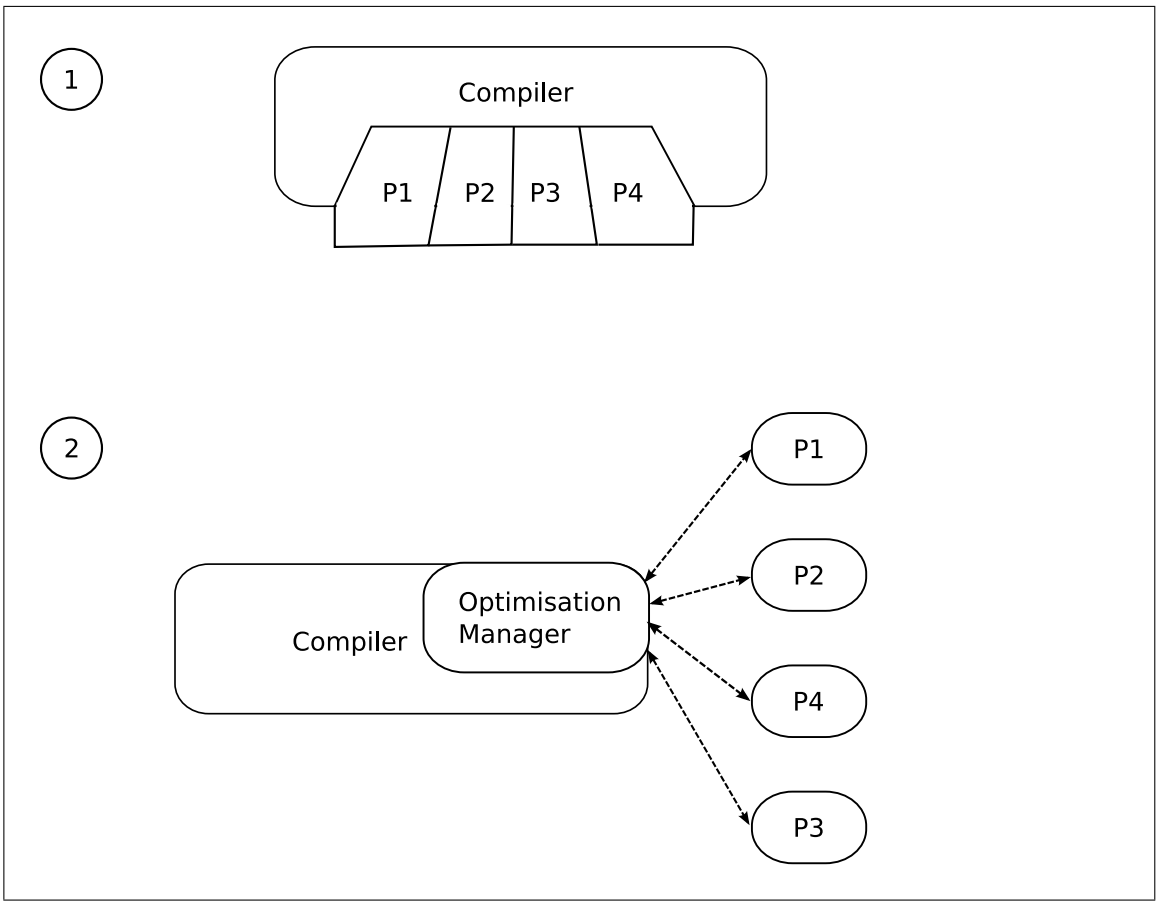

Figure 4.2: Two possible architectures for back-end plugins. 1) Dynamically loaded plug-ins 2) Plug-ins as stand-alone processes.

Dynamic loading and linking

The ability for a program to load and remove object code at run-time is known as *dynamic linking* and is well described in a paper [48] by Wilson and Olsson. A short summary is presented here.

An *object module* is the machine code equivalent of a source module. It commonly contains both a *text* segment, containing symbol names and machine code for functions corresponding to those names, and a *data* segment containing constants and other static data. *Linking*, is normally done statically, is the process of resolving references between modules and *relocating* object modules into a final executable module. Relocation is the act of changing memory addresses so that the executable can correctly reference symbols in the object module.

Unlike static linking, dynamic linking allows a process to add, remove, replace or relocate object modules within its address space during its execution. Adding an object module involves allocating space in the *heap* and then resolving references. Removal is the reverse process; all symbols that become unresolved are marked as undefined.

It is important to note that there is a one-time performance penalty in dynamic linking as the object module needs to be read from the disk. When multiple plug-ins are loaded this penalty can be substantial. This is exacerbated by the fact that when building a program consisting of many modules the conventional practice is to invoke the compiler once for each file. Good operating systems can mitigate this problem by temporarily caching program code in memory after termination. Upon reloading the program time can be saved by simply relocating the cached code.

Should this not suffice one could implement the compiler using a client/server model. In this scenario the compiler and its plug-ins are loaded once and for all and then remain resident in memory. Compilation of a source file is then achieved by sending a message to the compiler.

Stand-alone processes

In the stand-alone process architecture each plug-in optimisation is compiled to a separate binary executable. The processes then communicate with a component of the compiler which I have called the *optimisation manager*. It is responsible for controlling the communication between stand-alone processes, effectively directing the control flow of the optimisation pipeline. The processes themselves can communicate with the manager either by serialising data structures or using shared memory. How the optimisation manager becomes aware of the stand-alone processes is an implementation choice. Some possibilities are that they could register themselves, that they could be registered in a database or that they could be registered via command line arguments to the compiler. One advantage of stand-alone processes is the loose-coupling between them, but this comes at the cost of increased communication between them. The processes could even communicate over a network with the use of CORBA [8].

The infrastructure that needs to be developed to support stand-alone processes would

probably be a little more heavy-weight than in the dynamic loading and linking setting. We cannot necessarily take advantage of safety features of the host language such as static typing. (A notable exception is Java's RMI technology.) Nor can we treat the plug-ins as first class objects. It will be necessary to mediate their control using the operating system's support for inter-process communication. Conversely, this means the control and communication manager could provide capabilities beyond that of the host language of the compiler.

The CoSy compiler framework [15, 39] essentially takes this approach. Although explicitly focused on allowing end users to write custom optimisations, their framework could easily be adapted for this use. More is said about this in Section 4.8.

4.2 Back-end plug-ins using dynamic loading and linking in GHC

Now that we have surveyed requirements and design decisions for a back-end plug-in architecture a specific case study is presented. This section describes the design and implementation of a dynamic loading and linking based back-end plug-in architecture for the Glasgow Haskell Compiler. This is followed by the design and implementation of a domain specific optimisation for the Pan EDSL. In Section 4.6 the efficacy of this optimisation is demonstrated.

4.2.1 Retrofitting GHC with back-end plug-ins

As it turned out GHC already had many features which made it easier retrofit it with back-end plug-in support.The implementers of GHC chose a design based on the idea of correctness preserving transformations. The principles of this approach are:

- Their intermediate language is Core — a simple, well-understood variant of the lambda calculus.
- Each optimisation is just a transformation from the intermediate representation to itself.
- Annotations that are added to the representation are not *necessary* for other optimisations. Annotations may help improve the quality of code produced in later passes but the lack of them will not cause the optimisation pipeline to stall.

In a paper [77] of the implementers' experiences they state that a transformational approach is attractive for two reasons. First, it means that each transformation can be implemented, verified and tested separately. This is not only attractive but a necessity for plug-in writers. Second, it makes it easy to "plug and play" by re-ordering transformations, applying them more than once, or trading compilation time for code quality by omitting some. They state that this "allows a late commitment to phase ordering."

The fact that transformations have no side-effects and that annotations are only optional for later phases means that there are no hard dependencies between plug-in optimisations. This design, while perhaps not as powerful as one that handled dependencies more thoroughly, is certainly safe to use in a plug-in environment. More is said about the implementation in Section 4.4.2.

4.2.2 An image lifting optimisation for the Pan EDSL

This section draws upon the description of the Pan EDSL in Section 3.2. In this chapter the name Pan is used to refer to *our* implementation of the EDSL as opposed to Elliot et al.'s. Also, new types have defined to represent points and colours; tuples are no longer used. This was mainly for performance reasons but it also provides additional type safety.

The *image lifting* optimisation exploits the fact that some images do not depend on one of the Cartesian axes. An example of this is presented below: clearly, an image of horizontal stripes only depends on the y axis.

$$stripes\ (Pt\ x\ y)\ =\ if\ even\ (floor\ y)\ then\ black\ else\ white$$

The manner in which Pan displays images is simple: starting at the bottom left hand corner of the area to be displayed, the image is sampled repeatedly from left to right, and then up; once for each pixel. The result of each sample is written to video memory. Other point sampling orders are possible, but for efficiency reasons few others make sense. A simplified version of the display function appears below; the real version is optimised for speed, and hence less clear.

$$\begin{aligned}
&blitLoop\ 0\ 0\ =\ return\ ()\\
&blitLoop\ 0\ n\ =\ blitLoop\ \langle\text{screen width}\rangle\ (n-1)\\
&blitLoop\ m\ n\ =\ \textbf{do}\\
&\qquad\qquad \textbf{let}\ (i,j)\ =\ \langle\text{transform } (m,n) \text{ to image co-ordinates}\rangle\\
&\qquad\qquad \textbf{let}\ col\ =\ image\ (Pt\ i\ j)\\
&\qquad\qquad \langle\text{write } col \text{ to video memory at position } (m,n)\rangle\\
&\qquad\qquad blitLoop\ (m\ -\ 1)\ n
\end{aligned}$$

An optimisation opportunity immediately presents itself. This occurs when portions of an image are known to be the same as others. In such a case it makes sense to sample once and write many times. A special case of this general optimisation technique occurs when an image only depends on its y axis, that is, only the y coordinate is used to calculate the colour displayed at a given point. One need only sample the point once for each row and then repeatedly write that value to video memory.

The best way to do this is to lift images that are not dependent on their x axis to one level of recursion higher in the display function. That is, in our display function, consisting of a recursion across columns inside a recursion up the rows, we should lift images from the former level to the latter. From this point we call such images *liftable*.

The display function now works as follows: the lifted image is applied to the first point of each new row and the resulting colour is fed to the inner-most recursion, where it is

repeatedly written to video memory. Large gains in efficiency will result if the colour value of the point is expensive to calculate.

4.2.3 High-level description

As it is described above the image lifting optimisation would hardly ever be applicable since top-level liftable images are quite rare in practice. However, it is quite common for images to be composed of other images that *are* liftable. We should lift out the sub-images and leave behind a modified image function that, when given a list of colours as input, yields the same image as before: this is called the *partial inner image*. The colour value passed to the partial inner image are produced by applying the lifted images to the first point[1] in each row. An image with liftable sub-images appears below:

$$circleOnStripes \;=\; circle\;\text{`}over\text{`}\;stripes$$

Applying the aforementioned transformation should lift out stripes and yield the following *partial inner image*.

$$circleOnStripes \;=\; \lambda\,[c1] \;\rightarrow\; circle\;\text{`}over\text{`}\;(\lambda\_ \;\rightarrow\; c1)$$

However, even this formulation has problems; lifting sub-images is complicated by issues of scope.

Dealing with issues of scope

Some images, themselves part of the top-level image, may depend on variables that are brought into scope elsewhere. For example, consider the expression below.

$$\begin{aligned}&\textbf{let}\; w \;=\; 20\;\textbf{in}\; circle\;\text{`}over\text{`}\;\langle\text{imageGen}\rangle\;w\\ &\qquad\text{where}\;\langle\text{imageGen}\rangle\;::\;Int\;\rightarrow\;ImageC\;{}^{2}.\end{aligned}$$

The image, $\langle\text{imageGen}\rangle\;w$, cannot be lifted out since w would no longer be in scope. However, what we can do is η-expand $\langle\text{imageGen}\rangle$ yielding the following:

$$\begin{aligned}&\text{Lifted expression}\quad \lambda pt\;w \;\rightarrow\; \langle\text{imageGen}\rangle\;w\;pt\\ &\text{Partial inner image}\;\lambda\beta \;\rightarrow\; \textbf{let}\; w \;=\; 20\;\textbf{in}\; circle\;\text{`}over\text{`}\;(\lambda\_ \;\rightarrow\; \beta\;w)\end{aligned}$$

From this point on we refer to the result of lifting as a *lifted expression*, not a lifted image, since it is only an image in the case where there are no variables that would become out of scope. However, such expressions always have a target type of *Colour*.

The algorithm

With the last subtlety of lifting sub-images having been exposed (and dealt with) we are ready to discuss the image lifting algorithm. The first thing to do is identify the

[1] It doesn't actually matter which point in the row we apply lifted images to; the first is just convenient.

images that can be lifted by examining the structure of the top-level image to see if it has any sub-expressions that are also images; this is done by checking whether the type of the expressions is the same as that for images: *ImageC*. Since images may contain sub-images of their own, it is necessary to carry out this process recursively until a *primitive image* is found: an image that contains no sub-images.

Having found such a candidate image it must now be inspected to see if it is liftable. We do this by:

- looking for a pattern match of the form $(Pt\ \langle \mathrm{x} \rangle\ \langle \mathrm{y} \rangle)$
- retrieving the identifiers of $\langle \mathrm{x} \rangle$ and $\langle \mathrm{y} \rangle$. These have type *Id* and contain a unique identifier to aid in comparison with other identifiers
- looking for uses of $\langle \mathrm{x} \rangle$ and $\langle \mathrm{y} \rangle$ within the image. If $\langle \mathrm{x} \rangle$ is not used anywhere within the body of the alternative[3] then the image is liftable.

The lifting optimisation is conservative; it may not identify all liftable images but it will never erroneously lift one that is fully dependent on both the x and y axes.

Each liftable image needs be replaced with a so-called *constant image*: an image that ignores its point argument and returns a colour. This is done by first finding the variables that would no longer be in scope, creating a fresh variable which stands in place of the lifted expression, applying the variable to the out-of-scope variables and abstracting on a dummy variable.

Once the liftable images have been replaced with constant images, the modified image is no longer closed, since it contains fresh variables. Back at the top-level two things happen:

1. The fresh variables, $\beta_1, \ldots, \beta_n$, are used to create a lambda expression of the form: $\lambda\, \beta_1,\ \ldots,\ \beta_n\ \rightarrow\ \langle \text{modified image} \rangle$. This is the *partial inner image* referred to earlier and the expression is now closed.
2. A function known as the *row initialiser* is created. Its purpose is, before recursion on a row begins, to apply each lifted expression to the first point and then pass the resulting values to the partial inner image.

These two expressions become the arguments to the *augmented display function* of Figure 4.3. It applies the first point of each row to the row initialiser and passes the resulting image to the inner-most recursion: *innerBlitLoop*.

This concludes the overview of the lifting process. In the next few sub-sections we go into more detail, providing code examples and a formal account of the creation of constant images and lifted expressions.

[3] Each pattern match and associated body in a case expression is known as an *alternative*.

```
blitLoop 0 = return ()
blitLoop n = innerBlitLoop w

  where
    let j = ⟨transform n to image co-ordinate j⟩
    innerImage = ⟨row initialiser⟩ (Pt 0 j)
    innerBlitLoop 0 = blitLoop (n − 1)
    innerBlitLoop m = do
                        let col = innerImage pt
                        ⟨write col to video memory at position (m, n)⟩
                        innerBlitLoop (m − 1)
```

Figure 4.3: An augmented display function that incorporates lifted images

4.2.4 Formal account

A formal account of the construction of the partial inner image and lifted expressions is provided in Figures 4.4 and 4.5. However, before these can be understood some terms must be defined. The *precursor* is an intermediate, unclosed expression that is constructed while the partial inner image is being built. The *lift context* is a pair that is constructed for each liftable image. It consists of an identifier and a lifted expression. Each identifier corresponds precisely to one of the free variables in the precursor.

The remaining discussion closely follows the figures. The algorithm of Figure 4.4 is responsible for creating the partial inner image. This is done, in turn, by repeatedly following the algorithm in Figure 4.5. As mentioned earlier, it may well be the case that once an expression is lifted outside of the context in which it was defined that some variables may no longer be in scope. The free variables of a liftable image will be a superset of those of the top-level image. In order to find those free variables that were brought into scope *by* the top-level image it is only necessary to find the set difference of these two sets. It is then possible to construct both the constant image and the lifted expression.

The latter is constructed via η-expansion. The fresh variable, pt, becomes the first argument of the lifted expression. This is done so that the row initialiser can partially apply the lifted expression to the first point of each row during image display. The constant image and the lift context are then returned.

Having repeatedly performed the algorithm of Figure 4.5 the constant images are used to create the precursor. Then the free variables returned in the lift contexts, $\beta_1 \ldots \beta_m$, are used to create the partial inner image. In turn, the partial inner image and lifted expressions, $\langle \text{lifted} \rangle_1 \ \ldots \ \langle \text{lifted} \rangle_m$ (also returned in the lift contexts) are used to create the *row initialiser*. The row initialiser is the final product of the image lifting optimisation and is passed to the augmented display function of Figure 4.3.

In order to see the connection between the lifted expression and the constant image it is only necessary to consider what happens when the row initialiser partially applies the lifted expression to the first point on the row and then passes the resulting expression to the partial inner image. Once the resulting term has been substituted into the partial

1. Take the original image and find the liftable images.

   $\langle \text{image} \rangle \equiv \ldots \langle \text{liftable} \rangle_1 \ldots \langle \text{liftable} \rangle_m \ldots$

2. Replace them with constant images generated by using steps from Figure 4.5, yielding the *precursor* to the partial inner image.

   $\langle \text{precursor} \rangle \equiv \ldots \langle \text{const. image} \rangle_1 \ldots \langle \text{const. image} \rangle_m \ldots$

   and collect the lift contexts: $(\beta_i, \langle \text{lifted} \rangle_i)$ for $i = 1, \ldots, m$.

3. Close the precursor using the fresh variables returned in the lift contexts.

   $\langle \text{partial inner image} \rangle \equiv \lambda\, \beta_1 \ldots \beta_m \rightarrow \ldots \langle \text{const. image} \rangle_1 \ldots \langle \text{const. image} \rangle_m \ldots$

4. Create the *row initialiser* using the partial inner image and the lifted expressions returned in the lift contexts.

   $\langle \text{row initialiser} \rangle \equiv \lambda\, pt \rightarrow \langle \text{partial inner image} \rangle\, (\langle \text{lifted} \rangle_1\, pt) \ldots (\langle \text{lifted} \rangle_m\, pt)$

Figure 4.4: Creating the partial inner image

inner image we have the following:

$$\ldots (\lambda\, a_1 \ldots a_n \rightarrow \langle \text{subst. point value for } pt \quad \text{in} \quad \langle \text{liftedBody} \rangle \rangle)\, v_1 \ldots v_n \ldots$$

Further evaluation will clearly yield the correct result: each value v_i is substituted for the variable a_i. That is, the values of the variables that would have been out of scope have been restored.

4.3 Implementing the image lifting pass

We have yet to see how to implement the image lifting pass using the API provided in Figure 4.1. Instead of merely presenting pseudocode I have included code from my implementation for a plug-in enabled GHC. The code presented makes use of functions with very similar names to those of the API given before; we leave discussion of how the API was implemented in GHC until section 4.4.2. However, this code makes frequent use of *generic traversals* implemented in Lämmel and Peyton Jones' *Scrap Your Boilerplate* (SYB) library [60].

Generic traversals simplify the implementation of transformations that operate on rich, mutually recursive data structures. A generic traversal is constructed by defining one or more *type specific* transformations—these are transformations that are only defined for a specific type which resides somewhere within a rich, mutually recursive data structure—and then using one of SYB's traversal combinators to apply the type specific traversals in

1. Find a primitive image: $\langle \text{liftable} \rangle$
2. Find the η-*set* $= FV(\langle \text{liftable} \rangle) - FV(\langle \text{top-level image} \rangle) = \{v_1, \ldots, v_n\}$
3. Create a fresh identifier, β and assign it the type $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \mathit{Colour}$, where $\mathit{typeof}(v_i) = \tau_i$.
4. Create the constant image: $\lambda \_ \rightarrow \beta\ v_1\ \ldots\ v_n$
5. Replace each occurrence of v_i in $\langle \text{liftable} \rangle$ with a fresh variable a_i and assign it type τ_i (i.e. same type as v_i). Call this $\langle \text{liftedBody} \rangle$.
6. Create a fresh variable pt of type *Point*
7. Create the lifted expression: $\langle \text{lifted} \rangle \equiv \lambda\ pt\ a_1\ \ldots\ a_n \rightarrow \langle \text{liftedBody} \rangle\ pt$.
8. Create the lift context: $(\beta, \langle \text{lifted} \rangle)$

Figure 4.5: Creating a constant image and lift context

a certain manner. For instance, one may choose to apply the type specific transformation wherever possible and in a top-down manner, or one may choose to apply it at most once in a bottom-up manner.

In the following code I make use of the following generic traversals:

- *somewhere*. This combinator takes a type specific transformation which has the additional property that it may fail. It then tries to apply the transformation on a data structure. If it fails at the root term it then recursively calls the transformation on subterms. It stops when it has succeeded on a subterm or if it has failed everywhere.
- *everything*. This combinator takes a type specific query—queries can return arbitrary values but are most often used to return boolean values based on whether a term satisfies a predicate—and applies it everywhere. The results of the query are combined using a user supplied function. Among other things, it is useful for checking that a particular property holds everywhere within a data structure.

The correspondence between the API presented earlier and the one used in the forthcoming code is so close that it almost doesn't require explanation. Functions such as *newId*, *lookupType*, etc now have a *cp* prefix (e.g. *cpNewId*). Also, the monad in which the image lifting pass is written is called *LiftImageM*.

4.3.1 Isolating primitive images

Isolating primitive images is no more difficult than explained in the Section 4.2.3. The function *isPrimImage* returns *True* if and only if an expression is of image type and it does not contain any sub-images.

isPrimImage :: *CoreExpr* → *LiftImageM Bool*
isPrimImage coreExpr = **do**

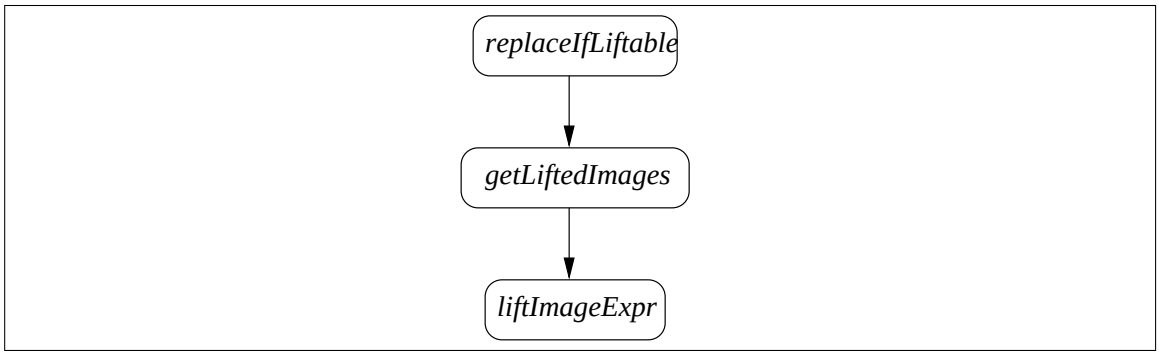

Figure 4.6: The call graph of *replaceIfLiftable*

```
isIm ← isImage coreExpr
contains ← containsImage coreExpr
let result = isIm && not contains
return result
```

As can be seen *isPrimImage* dispatches to *isImage* and *containsImage*. The former merely checks if an expression has the same type as an image. The latter performs a generic traversal to see if there are any sub-images within the current expression. It makes use of a custom generic traversal (not found in the SYB library) that works like *everything* except that it only traverses sub-expressions, not the top-level one. The code for these functions appears below.

```
isImage :: CoreExpr → LiftImageM Bool
isImage coreExpr
  | Type _ ← coreExpr = return False
  | otherwise = do
    imageCType ← cpLookupType panImageModule "imageCType"
    return (exprType coreExpr `tcEqType` imageCType)

containsImage :: Data a ⇒ a → LiftImageM Bool
containsImage = everythingButThis (∗||∗)
                                  (return False)
                                  (mkQ (return False) isImage)
```

4.3.2 Lifting images

I now present code that implements the algorithm presented in Figure 4.4. In order to guide the following discussion I present a call graph (Figure 4.6) of the important functions that make up the image lifting pass.

The function *replaceIfLiftable* does two things:

- It dispatches to *getLiftableImages* which is responsible for returning the *precursor* to the partial inner image and a list of *lift contexts*.

```
getLiftedImages :: CoreExpr → LiftImageM (CoreExpr, [LiftContext])
getLiftedImages coreExpr =
  (do guardM (isPrimImage coreExpr *&&* notM (fullDepImage coreExpr))
    (newExpr, lifted) ← liftImageExpr coreExpr
    return (newExpr, [lifted]))
  `mplus`
  (do recursivelyGetLiftedImages coreExpr)
```

Figure 4.7: *getLiftedImages*

- It creates the *partial inner image* using the lift contexts, and then the *row initialiser*.

The function, *getLiftedImages*, yields the precursor and lift contexts by repeatedly dispatching to *liftImageExpr* and gathering the results together. The "work horse" of the image lifting pass is *liftImageExpr* and it is only applied to liftable images. It yields a pair consisting of a *constant image* and a *lift context*. *getLiftedImages* uses the constant images to replace the liftable images in the top-level image, and simply gathers the lift contexts images together into a list which is passed back to *replaceIfLiftable*. Figures 4.7 and 4.8 present the complete definition of these functions.

The most salient points to note are as follows. The code for *getLiftedImages* (Figure 4.7) uses the *mplus* combinator to combine two monadic effects. The first checks whether the current expression is a primitive image and uses *liftImageExpr* to attempt to lift it. If the current expression is not a primitive image or the image is not liftable then the second monadic effect will be run. This one recursively tries to apply *getLiftedImages* to sub-expressions and combines the results. Its code is omitted.

The code for *liftImageExpr* (Figure 4.8) closely follows the algorithm of Figure 4.5.

4.4 Adding plug-in support and implementing the API in GHC

Plug-in support was added by modifying the GHC compiler. The API was largely implemented by exposing the internals of GHC to plug-in writers. However, effort went into simplifying the interface as much as possible.

4.4.1 Adding plug-in support

For some time GHC has been able to run in an interactive mode known as GHCi. It differs from a regular interpreter in that it can load both source files and pre-compiled object files; for the latter to be loaded interface files must also be present. This requires that GHC include support for dynamic loading and linking of object into its run-time address space in the manner of Wilson and Olsson's *dld* [48]. This infrastructure was leveraged by Pang to create a run-time loader and then further extended by Stewart to create the popular *hs-plugins* library [75].

```
liftImageExpr :: CoreExpr → LiftImageM (CoreExpr, LiftContext)
liftImageExpr coreExpr = do
  topCoreExpr ← getCoreExpr
  imageCType ← cpLookupType panImageModule "imageCType"
  v ← cpNewId "v" imageCType
  let topFVs  = exprFreeVars topCoreExpr
      expFVs  = exprFreeVars coreExpr
      fvs     = varSetElems (expFVs `minusVarSet` topFVs)
  debug $ "Top free vars: " ++ showPpr topFVs
  debug $ "Exp FVs: " ++ showPpr expFVs
      --
      -- fvs = variables that are in the top level expression but not
      -- in the current one. i.e. they are in scope only because
      -- they are defined somewhere else in the top-level expression.
      --
  fvVarPairs ← mapM varForFV fvs
  colourType ← cpLookupType panImageModule "colourType"
  pointType  ← cpLookupType panImageModule "pointType"
  let newExpr = replaceVars fvVarPairs coreExpr
      newVars = map snd fvVarPairs
        --
        -- Creates type of form: a1 → ... an → Colour, where
        -- a1...an are the types of the free variables.
        --
      colFunType = mkPiTypes fvs colourType
  colFunId  ← cpNewId "colFun" colFunType
  dummy     ← cpNewId "dummy" pointType
  pointType ← cpLookupType panImageModule "pointType"
  point     ← cpNewId "pt" pointType
  let colFun    = Var colFunId
      liftedExp = Lam point (foldr Lam (App newExpr (Var point)) newVars)
  return (Lam dummy (foldl App colFun (map Var fvs))
          ,(colFunId, liftedExp))
  where
      --
      -- Returns a pair consisting of the old free variable and a newly
      -- generated replacement for it.
      --
    varForFV :: Id → LiftImageM (Id, Id)
    varForFV fv = do
      newVar ← cpNewId "fv" (idType fv)
      return (fv, newVar)
```

Figure 4.8: *liftImageExpr*

Only a few relatively minor changes were necessary to introduce plug-in support to the Glasgow Haskell Compiler. Angelov and Marlow have integrated GHC with Microsoft's Visual Studio IDE [16]. They provided the internals of GHC as an API to the IDE by building it as a package[4], henceforth referred to as *package ghc*.

Combining the dynamic linking capabilities provided by the GHCi infrastructure with *package ghc* allowed plug-in support to be provided with a minimal implementation cost. The basic idea is as follows:

- A plug-in point is defined within the source code. It loads the plug-in and expects a function of a particular type.
- Users write a plug-in that imports *package ghc* and provides a function of the required type.
- When the compiler is run it dynamically loads the user-written plug-in using the infrastructure provided by GHCi . It is then run.

Writing and running optimisation plug-ins

Optimisation plug-ins are written as packages and loaded on the command line via a new command line flag: `-fcore-plugins`. This can be done, either by registering it centrally, using the *ghc-pkg* utility, or specifying the path to the package configuration file on the command line using the flag `-package-conf`.

This package must export a single (nullary) function with signature:

$$corePasses :: [CorePass]$$

This list specifies the order in which the passes are to be run. The *core-plugins* package is compiled against *package ghc* so that existing optimisations can be re-used. The user is free to insert their own passes, at any point, and as many times as they wish, within the list of passes. In this way the optimisation phases can be *scripted*. This is a key benefit of using back-end plug-ins.

4.4.2 Implementing the API

GHC performs its optimisations upon an intermediate language known as Core: a typed lambda calculus with support for pattern matching and data types. Its type system is closely based on System F and hence, is more powerful than Haskell's[5]. As might be expected a substantial library of functions already exists to manipulate Core expressions. However, the functionality required for the API was scattered among several modules and

[4] *Packages* are GHC's mechanism for providing libraries. They consist of library binaries coupled together with interface files and dependency information and can be included on the command line using the `-package` option.

[5] Incidentally this has turned out to be an excellent design choice. On more than one occasion when considering a type extension to Haskell a System F encoding has been found to exist.

```
type CPM s a = StateT (s, CPState) IO a
data CPState =
  CPState{ cpHscEnv          :: HscEnv
         , cpDynFlags        :: DynFlags
         , cpUniqSupply      :: UniqSupply
         , cpOrigNameCache   :: OrigNameCache
         }
runCPM :: HscEnv → UniqSupply → s → CPM s a → IO a
runCPM hscEnv uniq st cpM = do
  nameCache ← readIORef (hsc_NC hscEnv)
  let dynFlags = hsc_dflags hscEnv
      origNameCache = nsNames nameCache
  evalStateT cpM (st, CPState{ cpHscEnv          = hscEnv
                             , cpDynFlags        = dynFlags
                             , cpUniqSupply      = uniq
                             , cpOrigNameCache   = origNameCache})
```

Figure 4.9: The *CPM* monad

implemented inside different monads. For instance, the monad in which existing Core-to-Core passes are written, *SimplM* provided functions for name generation but not for name look-up.

It was decided that a new monad should be defined in order that all functionality required could be provided in a single place. It is called the *core pass monad* (and is abbreviated *CPM* in the code).

An important part of the domain specific optimisations we wish to write is the look-up of names. In order to gain access to them it is necessary to look up the name cache stored inside the global environment. Since the name cache is stored inside an *IORef*, it is necessary to run our pass *inside* the *IO* monad. Hence *CPM* is defined as a stateful extension of the *IO* monad.

The definition of the monad appears in Figure 4.9.

4.5 Reusing existing Core-to-Core passes

One of the key benefits of plug-in optimisations is that the programmer can script the optimisation pipeline as they see fit; they can place their optimisations wherever they wish and as many times as they want within it. In this way the optimisation infrastructure of a compiler can be reused for maximum benefit. While writing this optimisation I found that two existing Core-to-Core passes increased the applicability of image lifting transformation: inlining and dead code elimination. This section covers the reasons.

4.5.1 Inlining

The astute reader may have noticed that unless a primitive image is inlined within the body of the top-level image it will not be lifted. This is because the lifting pass only looks

at the expression available; it does not look up function definitions when it encounters a variable. So, in fact, the *stripes* example from before would not be inlined unless it was defined as follows:

$$circleOnStripes \;=\; circle\;`over`\;(\lambda\,(Pt\;x\;y)\;\rightarrow\;\mathbf{if}\,even\;y\,\mathbf{then}\,black\,\mathbf{else}\,white)$$

Inlining is not the only solution. Since the rest of the module is able to be scrutinised by the image lifting pass we could, in principle, examine the body of a declaration to see if its use in the top-level image could be lifted. However, whether we implement this solution or inline, some form of recursion is required. For instance, consider the situation where only a sub-image of the declaration was liftable. Unless the body of stripes is also looked up an optimisation opportunity is missed. This is demonstrated below.

$$squareOnStripes \;=\; square\;`over`\;stripes$$
$$image \;=\; circle\;`over`\;squareOnStripes$$

Even though it appears these approaches are very similar I have chosen to aggressively inline. Not only is the process of looking up and examining the bodies of declarations tedious it also complicates the lifting optimisation somewhat by forcing us to lift images that are not the top-level image. To see why consider how *stripes* would be lifted in the code above. This would require the construction of a partial inner image from *squareOnStripes*. Inlining aggressively at the beginning simplifies the algorithm.

Although GHC has its own passes for inlining it was necessary to write a new one because the existing pass changed the code in such a way that we could no longer find the top-level effect expression. As part of the process of inlining it would introduce new top-level bindings. Fortunately, the inlining pass benefited from existing compiler infrastructure. The simplifier already creates so-called unfoldings for bind identifiers. An *unfolding* is the associated body of the top-level binding with that identifier. GHC decides whether or not to use its inlining phase. Also, it will *not* create an unfolding for a recursive function. By using unfolding information in our inlining pass we guarantee that it will terminate.

Generic programming made this pass particularly easy to write. I defined a new generic combinator that applies a transformation in a top-down manner. The transformation itself inlined any use of what was dubbed an *image generator*—a function which had as target type *Colour*. An excerpt of the code appears in Figure 4.10

Interestingly, inlining by itself may not be enough to catch all liftable images.

$$stripesWeird\;(Pt\;x\;y)\;=\;stripes\;(Pt\;x\;y)$$

Although we can determine that *stripes* is liftable, *stripesWeird* would not be considered liftable since we pattern match on x and y and use them in the body of the definition; the optimisation is conservative. A combination of η-contraction and inlining would solve the problem in this case. There are doubtless other opportunities for maximising the liftability of images. This is left as future work.

```
everywhereM' :: Monad m ⇒ GenericM m → GenericM m
everywhereM' f x = do x' ← f x
  gmapM (everywhereM' f) x'

inlineImageGenerators :: CoreExpr → InlineImageM CoreExpr
inlineImageGenerators = everywhereM' (mkM inlineImageGenerator)
  where
    inlineImageGenerator :: CoreExpr → InlineImageM CoreExpr
    inlineImageGenerator exp =
      (do
        isImageGeneratorBool ← isImageGenerator exp
        (Var id) ← return exp
        guard (isImageGeneratorBool && coreBndrHasUnfolding id)
        debug $ "Unfolding expression: " ++ showPpr exp
        let unfolding = getCoreBndrUnfoldExpr id
        debug $ "The unfolding is: " ++ showPpr unfolding
        return unfolding)
      `mplus`
      (return exp)
```

Figure 4.10: The heart of the custom inlining pass

4.5.2 Dead code elimination

Dead code elimination is necessary in order for the detection of non-dependence on the x-axis to work correctly. The reason for this is most clearly presented by looking at a dump of the Core code generated for the *stripes* effect (Figure 4.11)

For performance reasons I defined the fields of the *Point* data structure to be strict. GHC recognises that it can unbox the values. Of course, this means that they must be re-boxed in order for them to be used, which GHC does, declaring two new identifiers inside let-expressions. Now the problem is clear: the x identifier is *used* inside the let-expression and then never again. Unfortunately, the lift images pass will detect this as dependence on the x-axis.

Fortunately, running the pass that performs dead code elimination *before* the image lifting pass removes the declaration in which the x value is boxed. This is a clear demonstration of the benefit of reusing existing Core-to-Core passes.

4.5.3 Beta reduction

It was also found that timely beta-reduction could lead to important opportunities for code improvement. I now sketch an example. Consider a liftable effect of the form:

$$imageCombinator\ v_1\ \ldots\ v_n$$

where $v_1, \ldots, v_n$ are expressions that are constant with respect to the evaluation of the image; they only need be calculated once before the image is displayed. Now, say the inlining of *imageCombinator* is a lambda expression of the form $\lambda a_1\ \ldots\ a_n\ pt\ \rightarrow\ \langle$a colour$\rangle$.

```
(\ (ds_d28R :: PanziImage.Point) ->
 case PanziImage.Colour ds_d28R of wild_B1 { PanziImage.Pt rb_d28T rb_d28U ->
 let {
   x_a1u8 :: PanziImage.Frac
   []
   x_a1u8 = GHCziFloat.Fzh rb_d28T } in
 let {
   y_a1u9 :: PanziImage.Frac
   []
   y_a1u9 = GHCziFloat.Fzh rb_d28U } in
 __letrec {
 } in
   case PanziImage.Colour even_a28O (floor_a28P y_a1u9) of wild_B1 {
     GHCziBase.False -> PanziImage.white; GHCziBase.True -> PanziImage.black
   }
 })
```

Figure 4.11: A dump of the Core code generated for *stripes*.

```
corePasses :: [CorePass]
corePasses = [   -- Performs, among other things, dead code elimination
                 coreDoSimplify SimplGently [NoCaseOfCase, maxIterations]
               , inlineImagesPass
                    -- Does some important beta reduction after inlining
               , coreDoSimplify (SimplPhase 2) [maxIterations]
               , liftImagesPass
                    -- other passes
               ...
               ]
```

Figure 4.12: How the optimisation pipeline is scripted

Without beta reduction the values of the expressions $v_1, \ldots v_n$ are not substituted for the variables $a_1, \ldots a_n$ in ⟨a colour⟩.

Without beta reduction the image lifting pass will detect a primitive image inside this expression: $\lambda\, pt \;\rightarrow\;$ ⟨a colour⟩. It will be lifted out yielding the following partial inner image.

$$\lambda\, f \;\rightarrow\; \lambda\, a_1 \;\ldots\; a_n \;\rightarrow\; \lambda\, \_ \;\rightarrow\; f\; v_1 \;\ldots\; v_n$$

Unfortunately, this means that the expression $f\; v_1 \;\ldots\; v_n$ is evaluated in *every* iteration of the inner loop of the display function. Since these values need only be calculated once this is wasteful.

By inserting a beta reduction pass between the inlining and image lifting pass this scenario can be avoided.

4.5.4 Scripting the optimisations

As mentioned in Section 4.4.1 the optimisation pipeline is scripted using the *corePasses* value. For the image lifting optimisations it is scripted as per Figure 4.12.

Number	Effect name
1	*WhiteOnRedOnBlack*
2	*Stripes*
3	*ColouredStripes*
4	*StripesOnStripes*
5	*StripesOfWidth*
6	*CircleOnStripes*

Performance	1	2	3	4	5	6
with lifting (frames/s)	11.61	28.70	27.39	15.91	27.91	11.98
without lifting (frames/s)	12.25	20.45	2.30	1.31	3.68	3.76
speed up	0.95x	1.40x	11.92x	12.13x	7.58x	3.19x

Figure 4.13: Results for six effects containing liftable images

As can be seen we place the *inlineImagesPass* and *liftImagesPass* early in the pipeline, so that the code has been minimally transformed. A special purpose optimisation such as image lifting can only be applied to very specific forms of expressions. The more optimisations that are run over the code, the less likely the image lifting pass will correctly identify these patterns.

Before inlining we insert a call to the *coreDoSimplify* pass. This is responsible for dead code elimination. The second call to this pass, this time with different arguments, perform beta reduction among other things.

4.6 Benchmarks

The performance of the effects when optimised and un-optimised appear in Figure 4.13. All examples contain liftable images. Collectively they cover cases such as liftable images contained within others, mixtures of non-liftable and liftable images, and compositions of liftable images. In all cases except *WhiteOnRedOnBlack* there are significant gains in performance. The reason that *WhiteOnRedOnBlack* suffers a small performance hit is that plain colour images are literal values. The overhead of calculating their value and then passing them as function arguments to the inner loop of the display function reduces performance.

4.7 Future work: a DSL for writing optimisations

As we mentioned in Section 2.4.3 one of the criticisms of plug-in compilers is that portability is lost due to the fact that the one is tied to the internals of a particular compiler. One way to mitigate this problem is to define *plug-in DSLs*: small languages that are well suited to particular phases of a compiler. GHC's rewrite rules [78] can be seen in this light; they provide a small language for writing limited source-to-source transformations. This section covers some ideas on useful features that could be of aid in writing Core-to-Core passes.

Features found in Template Haskell [85], notably quasi-quotation, would be beneficial. Using a Template Haskell-like quasi-quotation notation a programmer could write expressions such as

```
do
  point ← newIdPanM "pt" pointType
  return (App body point)
```

using the much more natural syntax

```
[| λ pt → $(body) pt |]
```

Such a domain specific language extension would be implemented in manner very similar to that of Template Haskell.

Also, we could add generic traversal as a native feature into the DSL. Not only would this provide a more convenient syntax it would also mean that the performance of the traversals could be improved. One of the problems with the SYB library at the moment is that no static analysis is (or can be) done on a data type in order to determine whether it is even worth descending recursively into a field of a data type for a particular traversal. For example, consider the following data type.

```
data Foo = Foo Bar Baz
data Bar = Bar String
data Baz = Nil
         | Baz Name Baz
```

Now consider the type specific function, *transBar* and the generic traversal *transFoo* created from it.

```
transBar (Bar str) = str ++ "_suffix"
transFoo = everywhere (mkT transBar)
```

When applying *transFoo* to a value of type *Foo* there is no point whatsoever in descending into the *Baz* field. Unfortunately, SYB can't know this and will attempt to do so anyway. A simple static analysis should be able to detect this.

4.7.1 A note on implementing plug-in DSLs

Without presenting the techniques of Chapter 5 and 6, which allow a programmer to extend the syntax and semantics of a language using front-end plug-ins, a full discussion of the implementation of plug-in DSLs is not possible. However, we do note the following important points now. Plug-in DSLs are implemented via a process of bootstrapping; the DSL is to be used within the source code of the plug-in compiler, yet the compiler is required to implement them in the first place. Thus, bootstrapping requires that the compiler is self-hosting (i.e it must be written in the language it compiles). Then the domain specific extensions are added using front-end plug-ins. Once these plug-ins have been compiled to object code, it then becomes possible to write modules of the compiler in the DSL.

4.8 Related work

Although there has been a lot of research into extensible compiler frameworks [72, 108, 64], little attention has been paid to plug-in compilers.

The SUIF2 Compiler Infrastructure [13] was designed to support collaborative research and development of compilation techniques. Its design allows the dynamic loading of modules which can be applied to programs in memory. Despite not using the term themselves this is the essence of plug-ins. Much like ours, the architecture can be used to write both optimisations and script the order in which they are run. The authors also mention that analysis phases can be written. This is also true of our plug-in compiler although generally it is considered more preferable to do this on the abstract syntax of the full language not just on the intermediate representation.

Engler [36] has also implemented a plug-in compiler called MAGIK. In this author's opinion he is one of the few researchers to have fully grasped the far reaching implications of a plug-in based compiler.

> It is the hope of the author that elevating [...] interfaces [...] to first class citizenship (where they are optimized and checked easily and well by compilers) will change programming practice in a non-trivial way. For decades, there has been a clamor for higher and higher-level languages. But, in fact, these languages are already prevalent, as a simple perusal of header files and module definitions will show. There apparent absence is merely due to lack of compiler support.

He notes that plug-in compilers allow domain specific checks, analysis and transformation to be done on libraries and, similar to Veldhuizen [96], states that the line between languages and libraries becomes blurry once this is possible: "library design [becomes] language design". In a short discussion on the limitations of MAGIK he also discusses the addition of syntactic sugar to make the construction of expressions both easier and safer. In fact, he suggests using 'C ("tick-C") [37] a language that provides very similar quasi-quotation capabilities to those of Template Haskell. He also notes, just as we have, that one should choose a simple intermediate representation in order to keep the number of cases small for transformation writers.

The benefits of plug-in based architectures were also advocated in a Wilson's [106] keynote address at the OOPSLA'98 conference. He states that next generation programming systems can accomplish extensibility via tools that support plug-ins, languages that allow extension of syntax, and, interestingly, the storage of source code in rich heterogeneous formats.

The CoSy compiler framework [15, 39], although never listing the ability for end users to write plug-in as a design objective, would be eminently suitable for implementing a back-end plug-in architecture. The basic idea is centered around "engines" which perform analyses, transformations and code generation on a common (and extensible) intermediate representation. Engines are run as stand-alone processes in parallel. The framework

defines a few DSLs the most notable being the Engine Description Language (EDL) which is used to describe the control flow and interaction of engines. Most importantly, the engines can be typed and dependencies between them can be defined.

The LLVM (Low-level Virtual Machine) Compiler infrastructure [62] was designed to enable effective program optimisation across the entire lifetime of a program. Although only statically extensible, it is an excellent example of a compiler infrastructure that provides a well-defined API for writing optimisation passes. The intermediate representation is static single assignment ([25]) based and in conjunction with a number of sophisticated analyses can be used to write global and inter-procedural optimisations.

4.9 Summary

The chapter began with a discussion of the requirements of a back-end plug-in architecture and design choices to be made when implementing them. It then explained the architecture we chose for our proof of concept: support for plug-ins in the Glasgow Haskell Compiler. A domain specific optimisation for the Pan EDSL was then introduced and how it was realised as a plug-in was demonstrated. Its effectiveness has been shown through benchmarking. This optimisation benefits from pre-existing optimisations such as inlining and dead code elimination. For the image lifting optimisation to benefit from these passes it must be possible to place it after they have run. The ease with which this can be done can be seen clearly in Figure 4.12. Indeed, the optimisation pipeline is completely scriptable by plug-in writers; they can remove and add existing and newly written passes at will. They can even schedule a pass to run multiple times.

However, without language support writing optimisations can be tedious and this is true whether they are plug-ins or pre-existing. I have sketched a solution to this problem in which the plug-in compiler itself is bootstrapped to provide a domain specific language for writing optimisations. This provides a layer of abstraction between the internals of the compiler and the plug-in writer, increasing the portability of the code. Standardisation of such DSLs is a recommended practice.

Chapter 5

Extensible data types in Haskell

5.1 Introduction

In the last chapter we demonstrated the benefits of a compiler instrumented with plug-in optimisations. However, for plug-in compilers to reach their full potential they must allow programmers to extend the language they compile via new syntactic forms, new semantics and domain specific error messages. In a traditional compiler this might be done by modifying the source of the parser, desugaring phase and type checker and extending the data structures and functions of these phases in the process.

When the source code is available modifying each function which operates on an extended data structure is merely time consuming, error prone and tedious. But it is impossible when all that is available is object code. Plug-ins, by themselves, are of no help in solving this problem as they can only add new declarations. They cannot modify or override existing code[1]. To go back to modifying source code is an intolerable option; to do so is to immediately lose the benefit of the abstraction provided by a plug-in compiler. What is required is a mechanism with which data types can be *extended* via new declarations in new modules. This problem has been well-studied. It was coined the *expression problem* by Wadler [101] on the Java-Genericity mailing list. Although it originally described a specific problem—extending a program that processes terms of a simple programming language—it has come to represent the general problem of extensible data types. Solutions in several languages have been proposed but few are entirely satisfactory. Zenger and Odersky [109] provide a good definition of the problem and a list of attendant criteria that a solution should satisfy. It is presented here essentially verbatim.

Suppose we have a data type which is defined by a set of cases and we have functions which operate on this data type. There are primarily two directions along which we can extend such a system:

- The extension of the data type with new data variants.
- The addition of new functions.

[1] A clever technique suggested by Stewart and Chakravarty[88] allows one to override function definitions if the compiler is set up for it. Nevertheless data declarations cannot be overridden.

Solving the expression problem requires an implementation technique which satisfies the following four requirements. (They defined a fifth criterion which is described in Section 5.8.)

- *Extensibility in both dimensions:* It should be possible to add new data variants and adapt existing operations accordingly. Furthermore, it should be possible to introduce new functions.
- *Strong static type safety:* It should be impossible to apply a function to a data variant which it cannot handle.
- *No modification or duplication.* It should not be necessary to change existing code, nor should it be necessary to re-implement functionality when extending since this effectively amounts to duplication.
- *Separate compilation:* Compiling data type extensions or adding new processors should not encompass re-type-checking the original data type or existing processors.

A key observation made by Reynolds [81] and later echoed by others ([107], [58]) was that object-oriented and functional languages can be seen as complementary approaches to data abstraction. In object-oriented languages variants of a data type are modelled using *classes*; usually each variant is defined as a subclass of an abstract superclass. Thus it is easy to add new variants. Unfortunately, the addition of new functionality on those variants is difficult; the only way to add new methods to a class is by sub-classing and it must be done for each variant. This quickly becomes unwieldy. In functional languages the converse is true: it is easy to add new functionality by defining new functions on a data type, but is difficult to add new variants. Another approach in object-oriented languages is to use the *visitor pattern* which makes it easy to add new functionality. However, as is the case with functional languages, adding new variants becomes difficult. Each of these approaches solves one half of the problem space but not the other.

The expression problem involves many subtleties. Despite informal solutions being proposed for the Haskell language (e.g. [59]) these only turn out to work on the small examples the proposed solution was demonstrated upon. Recently a complete solution was proposed by Löh and Hinze [66]. However, it falls short of our solution in two key ways. First, it does not provide true separate compilation. Second, it relies on features that have not yet been implemented in any Haskell compiler. This is discussed further in Section 5.8.

Our solution, while reliant on extensions to Haskell 98, works *as is* and provides true separate compilation. A solution satisfies this property if it is only necessary to recompile the modules in which changes are made whether they be on ordinary declarations or extensible ones. The solution is presented as a translation from a simple syntactic extension to Haskell to existing Haskell syntax. However, the translation can, and should be, viewed from more than one angle. First, the translation forms the basis for the implementation of a pre-processor. However, the target of the translation can also be seen as a *programming*

idiom which can be readily used by developers to implement extensible data types *by hand*. In Chapter 6 a compiler called PHRAC is presented which can load front-end plug-ins. In implementing it I successfully used the results of this chapter in just such an idiomatic way. No automatic translation was used.

The solution, henceforth known as *open abstract types*, uses several experimental features of Haskell: multi-parameter type classes, scoped type variables, kind annotations, zero constructor data types and recursive dictionaries. All of these features have been present from GHC 6.4 onwards.

Open abstract types will be presented as follows. First, syntactic sugar is introduced for declaring extensible data types. Next, a running example is introduced, demonstrating the new syntax in action. At this point it is necessary to cover a (relatively complex) technique that is instrumental in the translation. In Section 5.4 the concept of *retrospective superclassing* is introduced. Without presenting the formalisation of the translation, Section 5.5, shows us the result of applying the translation and the most salient points of the code are discussed. Section 5.6 introduces the formal translation, which can be used as the basis for the implementation of a pre-processor. Throughout its description the correspondence between the general rules and our running example are highlighted.

The chapter concludes with a comparison of solutions to the expression problem in other languages. In addition, the only other solution to the expression problem in Haskell is outlined and its shortcomings in a plug-in setting explained.

5.2 Syntactic sugar for open abstract types

Although the majority of this chapter is concerned with demonstrating an encoding of extensible data type support in Haskell we are ultimately interested in introducing syntactic sugar to reduce its syntactic burden. In this section I present two new data declaration forms as a means of motivating the rest of the chapter. In Section 5.6 an austere Haskell-like language augmented with these declarations becomes the source language in a formal translation to the encoding we are about to develop.

The two syntactic forms are *open data* and *extend data* declarations. A new extensible data type is introduced with the **open data** keywords.

```
module F0 where

open data Exp    = Var String
                 | Lam String Exp
                 | App Exp Exp
```

Functions can be defined upon these data types just like they can on ordinary algebraic data types.

```
alpha :: Exp → (String, String) → String
alpha (Var v) = ...
```

In another module we can then extend the data type using the **extend data** keywords as follows:

module $F1$ **where**

extend data $Exp\ =\ LetE\ String\ Exp\ Exp$

As usual it is possible to define new functions on the data type in this new module but in this case they can also be defined on the new *Let* variant.

$$
\begin{aligned}
&eval\ ::\ Exp\ \rightarrow\ Env\ \rightarrow\ Exp\\
&eval\ (Var\ name)\ =\ \ldots\\
&eval\ (Lam\ name\ body)\ =\ \ldots\\
&eval\ (App\ f\ x)\ =\ \ldots\\
&eval\ (LetE\ name\ body\ exp)\ =\ \ldots
\end{aligned}
$$

Unlike regular Haskell, new equations for the functions defined in the first module can be defined. However, this can only be done for the new variants introduced. In this case we would be limited to a new equation on the *Let* variant.

$$alpha\ (Let\ name\ body\ exp)\ =\ \ldots$$

The semantics of pattern matching is slightly different than usual. Since new equations can be introduced on existing functions whenever an *extend data* declaration the meaning of the wild card pattern becomes ambiguous. Consider the situation where the wild card pattern is used both in module *F0* and *F1*. Which one should be used? Does the new one equation override the old one? In order to simply the presentation of this chapter we have opted to disallow the wild-card pattern altogether. However, the *best-fit left-to-right* pattern matching solution devised by Löh and Hinze [66] could be implemented without too much trouble.

There are a few more restrictions on the new syntax. An *open data* and *extend data* declaration cannot appear in the same module. For a particular extensible data type there is at most one *extend data* declaration per module. It was stated earlier that new equations on existing functions *could* be defined. In fact, they *must* be; to omit them is an error.

5.3 A running example: the lambda calculus

As a running example we implement a data type representing the lambda calculus and two operations: alpha conversion and evaluation. At its simplest the lambda calculus consists of three core concepts: variables, abstraction and application.

We define two modules, an initial and one than extends the previous. The initial module appears in Figure 5.1 and defines the *alpha* function on a data type that represents just the core concepts of the lambda calculus.

```
module F0_Alpha
where

data Exp = Var String
         | Lam String Exp
         | App Exp Exp
alpha :: Exp → (String, String) → Exp
alpha (Var v       :: Exp) =
  λ(s :: (String, String)) → Var (swap s v)
alpha (Lam v body :: Exp) =
  λ(s :: (String, String)) → Lam (swap s v) (alpha body s)
alpha (App a b     :: Exp) =
  λ(s :: (String, String)) → App (alpha a s) (alpha b s)
swap :: (String, String) → String → String
swap ((a, b) :: (String, String)) = λ(o :: String) → if a == o then b else o
```

Figure 5.1: The initial module. It defines the data structure to represent the simple lambda calculus and an alpha conversion function.

We then extend the module in Figures 5.2 and 5.3. We add a new variant to the lambda calculus, *let expressions*. We then add a new equation for this variant to the *alpha* function and define two new functions, *eval* and *apply*.

The reader may notice that the functions are not defined as they usually would be. There is at most one pattern match for each function and in each case the pattern match is flat (i.e. not nested). Also, the right-hand side of each function is a lambda expression which while legal Haskell is not standard idiom. (Usually the parameters would appear on the left hand side of the equations.) In addition, readers may wonder why there is an *apply* function at all when this could easily be defined as a case expression inside *eval*.

The translation presented later in this chapter is complicated by many of the syntactically friendly features of Haskell such as where clauses, nested pattern matches, etc. To simplify the presentation the translation is assumed to be performed on an austere Haskell which includes the syntactic sugar introduced in Section 5.2. This is discussed in more detail in Section 5.6. By presenting our running example in this austere Haskell it is hoped that the correspondence between the rules of the translation and the result of applying them to Figures 5.1, 5.2, and 5.3 is much more readily apparent.

5.4 Läufer's method and *retrospective superclassing*

In Section 5.5 a complete translation of the program in Figures 5.1 and 5.2 is presented. The solution is based on an extension to the work of Läufer [63] and involves a technique that I have dubbed *retrospective superclassing*. This section will outline Läufer's work, show a gap in the solution the expression problem and present retrospective superclassing as a means of closing that gap. We also show why *recursive dictionaries*, a recent extension to Haskell, are necessary in order for retrospective superclassing to work.

```
module F1_Eval
where

import F0_Pretty
extend data Exp = Let String Exp Exp

alpha (LetE name body exp :: Exp) =
  λ(s :: (String, String)) →
    LetE (swap s name) (alpha body s) (alpha exp s)
eval :: Exp → Env → Exp
eval (Var name :: Exp) = λ(env :: Env) → lookupEnv env name
eval (Lam name body :: Exp) = λ(env :: Env) → Lam name body
eval (App f x :: Exp)    = λ(env :: Env) → apply x env (eval f env)
eval (LetE name body exp :: Exp) =
    λ(env :: Env) → eval (App (Lam name exp) body) env
apply :: Exp → Env → Exp → Exp
apply (Var name :: Exp) =
    λ(env :: Env) (x :: Exp) → error "Function expected"
apply (Lam name body :: Exp) =
    λ(env :: Env) (x :: Exp) → eval body (extEnv env (name, eval x env))
apply (App f x :: Exp)    =
    λ(env :: Env) (x :: Exp) → error "Function expected"
apply (LetE name body exp :: Exp) =
    λ(env :: Env) (x :: Exp) → error "Function expected"
```

Figure 5.2: The extension module. It extends the earlier data structure to represent let expression, defines an extra equation on the alpha conversion function and defines a new evaluation function.

```
type Env = [(String, Exp)]
lookupEnv :: Env → String → Exp
lookupEnv ([] :: Env) =
  λ(name :: String) → error $ "lookupEnv: Variable " ++
    show name ++ " not found"
lookupEnv (hd : tl :: Env) =
  λ(name′ :: String) → lookupEnvAux hd tl name′
lookupEnvAux :: (String, Exp) → Env → String → Exp
lookupEnvAux ((name, term) :: (String, Exp)) =
  λ(rest :: Env) (name′ :: String) →
    if name == name′ then term else lookupEnv rest name′
extEnv :: Env → (String, Exp) → Env
extEnv = λ(env :: Env) (x :: (String, Exp)) → x : env
```

Figure 5.3: Some helper functions that are also present in the extension module.

In Haskell, type classes are the only candidate for emphencoding extensible data types since they are the only *open* declarations. Most declarations in Haskell are *closed*: their meaning is fully determined once and for all in the module they are written in. Their very nature precludes them from being used to encode extensible data types. However, *instance declarations*, which define the functionality of class methods for a given type, are open. They can be defined in a module that is not the same as the *class declaration* as long as they do not *overlap*[2] with an existing instance.

Läufer [63] introduced a technique similar to the dynamic dispatch mechanism of object-oriented languages which can be used as the basis for a solution to the expression problem. The key idea is to treat a class declaration as the *interface* to an abstract data type. Existential types are then used "wrap" specific implementations of the abstract data type so that the only way to perform operations on the data type is through class methods. These methods are available because the class context is "wrapped up" inside the existential type. The technique is demonstrated on our running example. Below we introduce a class for the *alpha* function and an existential type *Exp* wraps up differing value behind the *MkExp* constructor. It shall be called the *wrapper type* from now on.

```
class Alpha a where
  alpha :: a → (String, String) → Exp

data Exp = forall a. Alpha a ⇒ MkExp a
```

Methods can then be defined on various data types but with the aid of an *unwrapping instance* can be applied to values of *Exp* and have the correct behaviour. The unwrapping instance provides us with a function of type $Exp \rightarrow Exp$ as required. Its definition is quite simple.

```
instance Alpha Exp where
  alpha (MkExp e) s = alpha e s
```

We now define *component types* and corresponding instances of the *Alpha* class to represent the core lambda calculus and the let expression extension. The component types are called *Exp_0* and *Exp_1* respectively. Note that where we used to have recursive occurrences of the data type we now refer to the wrapper type.

```
data Exp_0 = Var String
           | Lam String Exp
           | App Exp Exp

instance Alpha Exp_0 where ...
```

Exp_1 can be defined along with its instance in an entirely new module. Instances are open declarations.

[2] An overlap occurs when a given instance can be unified via substitution to another. e.g. $C(a, Int)$ overlaps with $C(Bool, b)$.

data *Exp_1* = *LetE String Exp Exp*

instance *Alpha Exp_1* **where** ...

5.4.1 The version problem

Let us now consider extending the functionality of the *Exp* data type by defining an interpreter on it. This will require a new class, *Eval*, to be defined. Using the inheritance mechanism of type classes we can require that *Alpha* is a superclass of *Eval*.

class *Alpha a* ⇒ *Eval a* **where** ...

Unfortunately, this requires that we introduce a new type, say *EExp*, to wrap up this new class, since *Exp* only wraps up the *Alpha* class.

data *EExp* = *forall a.Eval a* ⇒ *MkEExp a*

Without going any further we can see that there is going to be a problem. Once we have correctly defined instances on the component types and declared an unwrapping instance we will have a data type for which *eval* and *alpha* are both methods. However, while the type of *eval* is *EExp* → *Env* → *EExp* the type of *alpha* is *EExp* → (*String*, *String*) → *Exp*. The return type is the original type. Unfortunately, this means the following expression would not type check: *eval* (*alpha* (*MkExp* (*Var* "*a*")) ("*a*", "*c*")) [].

5.4.2 Retrospective superclassing

Let us look more closely at why this problem occurs. When a value of type *Exp* is unwrapped the value extracted has access to all of class *Alpha*'s methods and those of its superclasses, *and no more*. At present there is no way that we can define the function *alpha* to return values which will have access to methods that a programmer may write in the future.

The first hint of a solution becomes evident when we restate the methods a value of type *Exp* has access to, putting the emphasis in a different place this time: it has access to all of *Alpha*'s methods *and those of its superclasses*, and no more. If it were somehow possible to define *Eval* in such a way that it was a superclass of *Alpha* then values of type *Exp* would have access to these methods. This would be a kind of *retrospective superclassing*.

In fact, retrospective superclassing is possible using a technique due to Hughes [52] and elaborated upon by Lämmel and Peyton Jones [61] which allows abstraction over type classes. Hughes' suggestion was to allow declarations like the following:

class *cxt a* ⇒ *Alpha cxt a* **where**
  alpha :: *a* → (*String*, *String*) → *Exp cxt*

data *Exp cxt* = *forall a. Alpha cxt a* ⇒ *MkExp a*

This is not valid Haskell since the second parameter, *cxt*, of the *Alpha* class stands for a *class*, not a type or type constructor. However, let us assume for the moment that such declarations are legal. Now type *Exp* has an extra parameter, *cxt*, which abstracts over a class. Since this very same class is declared to be a superclass of *Alpha* we see that method *alpha* now returns values which have access to the methods in any class that *cxt* is instantiated to.

Fortunately, Hughes was successful in encoding just such an abstraction over classes and the technique is now demonstrated. First, we define a class *Sat* with a single method *dict*. This class is used to return an *explicit dictionary* whose values are taken directly from the implicit one associated with a given class.

```
class Sat a where
  dict :: a
```

Now, whenever the programmer defines a new class they also define a corresponding data type that represents explicitly the implicit dictionary of the class. The programmer also needs to define an instance that equates the methods of the explicit dictionary with those classes we wish to abstract over. The following self-contained example demonstrates this.

```
type Env = [(String, Exp EvalD)]
class Sat (cxt a) ⇒ Alpha cxt a where
  alpha :: a → (String, String) → Exp cxt
class Alpha EvalD a ⇒ Eval a where
  eval :: a → Env → Exp EvalD
data EvalD a = EvalD{ eval' :: a → Env → Exp EvalD }
instance Eval a ⇒ Sat (EvalD a) where
  dict = EvalD{ eval' = eval }
```

Here is a quick summary of the salient points:

- The class head, *class cxt a ⇒ Alpha a* , has become *class Sat (cxt a) ⇒ Alpha a*.
- *EvalD* is the explicit analogue of the implicit dictionary that is associated with the *Eval* class.
- The instance equates the methods of *Eval* with the explicit dictionary *EvalD*.

There is one remaining caveat – calls to extension methods must be done through explicit dictionaries. The following expression will not type check since method *eval* is not a member of any superclass of *Alpha*.

```
case alpha exp ("a", "b") of MkExp exp' → eval exp' []
```

However, *dict* is a method of *Alpha*'s superclass, *Sat*. All that is required is to replace *eval exp'* [] with *eval' dict exp'*[] which only imposes minor syntactic inconvenience.

```
module F0_Alpha
where

data P d
class Sat a where
  dict :: a
data Exp (cxt :: * → *) =
  forall b. Alpha cxt b ⇒ MkExp b
```

Figure 5.4a: Preliminaries: the proxy type, Sat class and wrapper type

```
class Sat (cxt b) ⇒ Alpha cxt b where
  alpha :: P cxt → b → (String, String) → Exp cxt
data Exp_0 cxt = Var String
               | Lam String (Exp cxt)
               | App (Exp cxt) (Exp cxt)
```

Figure 5.4b: Initial component type and the base functionality class

Retrospective superclassing relies on *recursive dictionaries*, a recently[3] implemented feature of GHC. These dictionaries allow cycles to occur while resolving the constraints introduced by class and instance declarations. We defer an in depth discussion of this to Section 5.5.3 but refer the reader to Lämmel and Peyton Jones' paper [61] on extensible generic functions where the technique was first described.

Notice that the explicit dictionary of the *Sat* instances "ties the knot" of constraint resolution. Interestingly, this brings the functionality introduced by each class—in this case *Alpha* and *Eval*— to the same semantic level. Obviously, the bodies of extension methods can contain uses of existing methods but the opposite is also true. In Section 5.6.4 we will see that it is possible to call extension functions from new equations on existing functions.

5.5 Translation of the running example

We are now ready to discuss the translation of the initial module (Figure 5.1) and the extension module (Figures 5.2 and 5.3) of Section 5.3.

In order not to overwhelm the reader the translation has been broken up into several sub-figures. The translation of the initial module appears in Figures 5.4a through 5.4g and the translated extension module in Figures 5.5a through 5.5h.

5.5.1 Initial module

Figure 5.4a introduces the *Sat* class and the wrapper type which, this time, contains a kind annotation. Although not strictly necessary in this case it is required when the open data type has parameters. We also introduce a *proxy type*, *P*. An argument of the proxy

[3] Recursive dictionaries are available from GHC 6.4 onwards.

```
instance (Sat (cxt (Exp cxt))
         , Sat (cxt (Exp_0 cxt))
         ) ⇒ Alpha cxt (Exp_0 cxt) where
  alpha (_ :: P cxt) (Var v :: Exp_0 cxt) =
    λ(s :: (String, String)) → var (u :: P cxt) (swap s v)
  alpha (_ :: P cxt) (Lam v body :: Exp_0 cxt) =
    λ(s :: (String, String)) →
      lam (u :: P cxt) (swap s v)
                       (alpha (u :: P cxt) body s)
  alpha (_ :: P cxt) (App a b :: Exp_0 cxt) =
    λ(s :: (String, String)) →
      app (u :: P cxt)
        (alpha (u :: P cxt) a s) (alpha (u :: P cxt) b s)
```

Figure 5.4c: Functionality instance

```
instance Sat (cxt (Exp cxt))
         ⇒ Alpha cxt (Exp cxt) where
  alpha (_ :: P cxt) (MkExp e :: Exp cxt) =
    λ(s :: (String, String)) → alpha (u :: P cxt) e s
```

Figure 5.4d: Unwrapping instance

```
data AlphaEnd b
class Alpha AlphaEnd b ⇒ AlphaCap b
instance AlphaCap (Exp_0 AlphaEnd)
instance AlphaCap (Exp AlphaEnd)
instance AlphaCap b ⇒ Sat (AlphaEnd b) where
  dict = error "Capped at Alpha"
```

Figure 5.4e: Capping classes, capping types and capping instances

```
var :: forall cxt. (Sat (cxt (Exp cxt))
                   , Sat (cxt (Exp_0 cxt))) ⇒
       P cxt → String → Exp cxt
var (_ :: P cxt) =
  λ(x1 :: String) → MkExp (Var x1 :: Exp_0 cxt)
lam :: forall cxt. (Sat (cxt (Exp cxt))
                   , Sat (cxt (Exp_0 cxt))) ⇒
       P cxt → String → Exp cxt → Exp cxt
lam (_ :: P cxt) = λ(x1 :: String) (x2 :: Exp cxt) →
                     MkExp (Lam x1 x2 :: Exp_0 cxt)
app :: forall cxt. (Sat (cxt (Exp cxt))
                   , Sat (cxt (Exp_0 cxt))) ⇒
       P cxt → Exp cxt → Exp cxt → Exp cxt
app (_ :: P cxt) = λ(x1 :: Exp cxt) (x2 :: Exp cxt) →
                     MkExp (App x1 x2 :: Exp_0 cxt)
```

Figure 5.4f: Smart constructors

```
swap :: (String, String) → String → String
swap ((a, b) :: (String, String)) =
  λ(o :: String) → if a == o then b else o
```

Figure 5.4g: Regular declarations

```
module F1_Eval
where
import F0_Alpha
data Exp_1 (cxt :: ∗ → ∗) = LetE String (Exp cxt) (Exp cxt)
```

Figure 5.5a: Module header and new component type

type is required[4] whenever the type signature of a method does not contain an occurrence of the *cxt* parameter. It is required for the correct unification of types. This is described in Section 5.6.4.

Figure 5.4b defines the initial functionality class, *Alpha* and the initial component type *Exp*_0. The functionality instance of Figure 5.4c defines the three equations of the *alpha* method on the *Var*, *Lam* and *App* variants of type *Exp*_0. There are two important things to note. First, there are two *Sat* constraints in the instance head, one on the initial component type and one on the wrapper type. The one for the wrapper type is necessary since *alpha* returns a value of type *Exp cxt*. Second, use is made of the smart constructors *var*, *lam* and *app* defined in Figure 5.4f. These simplify the presentation considerably and are also useful when constructing concrete values of type *Exp* τ (for some type τ).

We call the *swap* function in Figure 5.4g a *regular declaration* since it is not defined directly upon the open data type. Although it is unchanged in this translation this will not always be the case. If a function uses one of the instance methods it will need to be translated to a function containing a proxy argument necessitating the augmentation of its type. More is said about this in Section 5.6.

The only remaining figure to explain is Figure 5.4e. A *capping class* is a null extension that allows a programmer to use the EDT in its current state. A capping class is always accompanied by a *Sat* instance featuring the capping class in its context. (In this case the capping class is *AlphaCap*.)

5.5.2 Extension module

The first thing to notice about Figures 5.5a through 5.5h is that the type variable *cxt* has been replaced almost wholesale by *EvalD cxt*. *EvalD* is the name of the explicit dictionary defined in Figure 5.5c and its occurrence in the type *Exp* (*EvalD cxt*) gives a visual indication that evaluation is defined upon it. Although we present no more functionality for the *Exp* EDT it is readily extensible. As more functionality is added the *cxt* type variable is replaced with further explicit dictionaries, e.g. *Exp* (*EvalD* (*Pretty cxt*)) and so on.

[4]The proxy type is not strictly required for this example either.

```
instance (Sat (EvalD cxt (Exp (EvalD cxt)))
         , Sat (EvalD cxt (Exp_0 (EvalD cxt)))
         , Sat (EvalD cxt (Exp_1 (EvalD cxt)))
         ) ⇒ Alpha (EvalD cxt) (Exp_1 (EvalD cxt)) where
  alpha (_ :: P (EvalD cxt)) (LetE name body exp :: Exp_1 (EvalD cxt)) =
    λ(s :: (String, String)) →
      letE (u :: P (EvalD cxt)) (swap s name)
           (alpha (u :: P (EvalD cxt)) body s)
           (alpha (u :: P (EvalD cxt)) exp s)
```

Figure 5.5b: Instances for new equations on existing functions

```
class (Sat (EvalD cxt b), Alpha (EvalD cxt) b) ⇒ Eval cxt b where
  eval  :: P (EvalD cxt) → b → Env (EvalD cxt) →
           Exp (EvalD cxt)
  apply :: P (EvalD cxt) → b → Env (EvalD cxt) →
           Exp (EvalD cxt) → Exp (EvalD cxt)
data EvalD cxt b =
  EvalD{ eval'  :: P (EvalD cxt) → b → Env (EvalD cxt) →
                   Exp (EvalD cxt)
       , apply' :: P (EvalD cxt) → b → Env (EvalD cxt) →
                   Exp (EvalD cxt) → Exp (EvalD cxt)
       , evalExt :: cxt b}
```

Figure 5.5c: Functionality classes and explicit dictionary

```
instance Sat (EvalD cxt (Exp (EvalD cxt))) ⇒
  Eval cxt (Exp (EvalD cxt)) where
  eval (_ :: P (EvalD cxt)) (MkExp e :: Exp (EvalD cxt)) =
    λ(x1 :: Env (EvalD cxt)) → eval' dict (u :: P (EvalD cxt)) e x1
  apply (_ :: P (EvalD cxt)) (MkExp e :: Exp (EvalD cxt)) =
    λ(x1 :: Env (EvalD cxt)) (x2 :: Exp (EvalD cxt)) →
      apply' dict (u :: P (EvalD cxt)) e x1 x2
```

Figure 5.5d: Unwrapping instance

```
instance (Sat (EvalD cxt (Exp (EvalD cxt)))
          , Sat (EvalD cxt (Exp_0 (EvalD cxt)))
          , Sat (EvalD cxt (Exp_1 (EvalD cxt)))
          ) ⇒ Eval cxt (Exp_0 (EvalD cxt)) where
  eval (_ :: P (EvalD cxt)) (Var name :: Exp_0 (EvalD cxt)) =
    λ(env :: Env (EvalD cxt)) → lookupEnv env name
  eval (_ :: P (EvalD cxt)) (Lam name body :: Exp_0 (EvalD cxt)) =
    λ(env :: Env (EvalD cxt)) → lam (u :: P (EvalD cxt)) name body
  eval (_ :: P (EvalD cxt)) (App f x :: Exp_0 (EvalD cxt)) =
    λ(env :: Env (EvalD cxt)) →
      apply′ dict (u :: P (EvalD cxt)) x env
        (eval′ dict (u :: P (EvalD cxt)) f env)
  apply (_ :: P (EvalD cxt)) (Var v :: Exp_0 (EvalD cxt)) =
    λ(env :: Env (EvalD cxt)) (x :: Exp (EvalD cxt)) →
      error "Function expected"
  apply (_ :: P (EvalD cxt)) (Lam name body :: Exp_0 (EvalD cxt)) =
    λ(env :: Env (EvalD cxt)) (x :: Exp (EvalD cxt)) →
      eval′ dict (u :: P (EvalD cxt))
        body (extEnv env (name, eval′ dict (u :: P (EvalD cxt)) x env))
  apply (_ :: P (EvalD cxt)) (App f x :: Exp_0 (EvalD cxt)) =
    λ(env :: Env (EvalD cxt)) (x :: Exp (EvalD cxt)) →
      error "Function expected"
instance (Sat (EvalD cxt (Exp (EvalD cxt)))
          , Sat (EvalD cxt (Exp_0 (EvalD cxt)))
          , Sat (EvalD cxt (Exp_1 (EvalD cxt)))
          ) ⇒ Eval cxt (Exp_1 (EvalD cxt)) where
  eval (_ :: P (EvalD cxt)) (LetE name body exp :: Exp_1 (EvalD cxt)) =
    λ(env :: Env (EvalD cxt)) →
      eval′ dict (u :: P (EvalD cxt))
        (app (u :: P (EvalD cxt))
          (lam (u :: P (EvalD cxt)) name exp) body)
        env
  apply (_ :: P (EvalD cxt)) (LetE name body exp :: Exp_1 (EvalD cxt)) =
    λ(env :: Env (EvalD cxt)) (x :: Exp (EvalD cxt)) →
      error "Function expected"
```

Figure 5.5e: Instances for new functions on all component types

```
data EvalEnd b
class Eval EvalEnd b ⇒ EvalCap b
instance EvalCap (Exp (EvalD EvalEnd))
instance EvalCap (Exp_0 (EvalD EvalEnd))
instance EvalCap (Exp_1 (EvalD EvalEnd))
instance EvalCap b ⇒ Sat (EvalD EvalEnd b) where
  dict = EvalD{ eval′  = eval
              , apply′ = apply
              , evalExt = error "Capped at Eval"}
```

Figure 5.5f: Capping class, capping type and capping instances

```
letE :: forall cxt. (Sat (EvalD cxt (Exp (EvalD cxt)))
                     , Sat (EvalD cxt (Exp_0 (EvalD cxt)))
                     , Sat (EvalD cxt (Exp_1 (EvalD cxt)))) ⇒
  P (EvalD cxt) → String → Exp (EvalD cxt) →
  Exp (EvalD cxt) → Exp (EvalD cxt)
letE (_ :: P (EvalD cxt)) =
  λ(x1 :: String)
    (x2 :: Exp (EvalD cxt))
    (x3 :: Exp (EvalD cxt)) → MkExp (LetE x1 x2 x3 :: Exp_1 (EvalD cxt))
```

Figure 5.5g: Smart constructors

```
type Env cxt = [(String, Exp cxt)]
lookupEnv :: Env (EvalD cxt) → String → Exp (EvalD cxt)
lookupEnv ([] :: Env (EvalD cxt)) =
  λ(name :: String) → error ("lookupEnv : Variable " ++ show name ++
                             " not found")
lookupEnv (hd : tl :: Env (EvalD cxt)) =
  λ(name′ :: String) → lookupEnvAux hd tl name′
lookupEnvAux :: (String, Exp (EvalD cxt)) →
  Env (EvalD cxt) → String →
  Exp (EvalD cxt)
lookupEnvAux ((name, term) :: (String, Exp (EvalD cxt))) =
  λ(rest :: Env (EvalD cxt)) (name′ :: String) →
    if name == name′ then term else lookupEnv rest name′
extEnv :: Env (EvalD cxt) → (String, Exp (EvalD cxt)) →
          Env (EvalD cxt)
extEnv = λ(env :: Env (EvalD cxt))
  (x :: (String, Exp (EvalD cxt))) → x : env
```

Figure 5.5h: Regular declarations

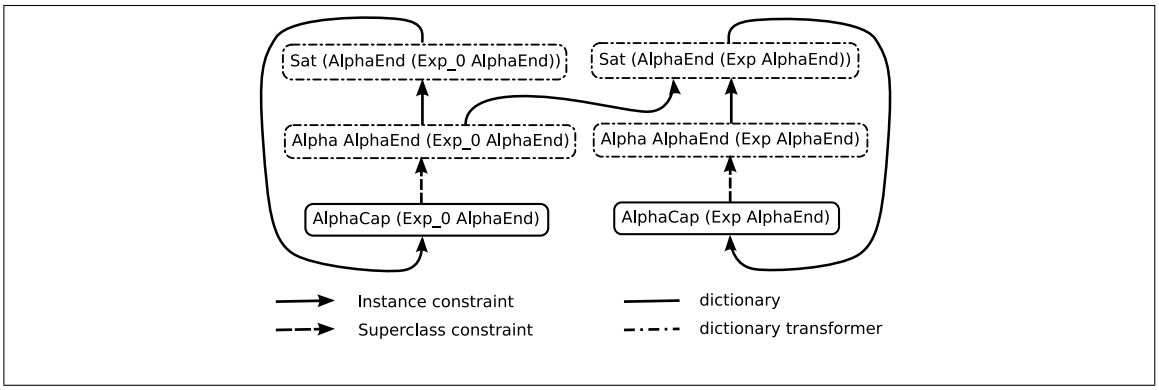

Figure 5.6h: A diagram of two recursive dictionaries produced by *AlphaCap* instances on *Exp* and *Exp_0*.

The *extension functionality class* is shown in Figure 5.5c. In general there will be one of these present in the translation whenever a new function is defined on the EDT.

Figure 5.5e, while much larger than the corresponding code in Figure 5.2 is a relatively straightforward translation of what is present there. One key difference is that an uses of *eval* and *apply* on the right hand sides of the equations have been replaced with calls to $eval'\ dict$ and $apply'\ dict$ respectively. This occurs in any instances on extension functionality classes.

Figure 5.5f introduces the capping classes, types and instances. Note that this time the methods of class *Eval*, *eval* and *apply* are equated with the selector methods of *EvalD*, $eval'$ and $apply'$. The selector method *evalExt* is equated with an error, much like *dict* was in Figure 5.4e. As more functionality is added to the *Exp* EDT the *dict* method of the *Sat* instance will come to consist of nested explicit dictionaries. See Section 5.6 for more detail.

The *regular declarations* of Figure 5.5h have changed in the translation. The *Env* type now has a *cxt* parameter because it references the *Exp* type. Similarly the types of *lookupEnv*, *lookupEnvAux* and *extEnv* have changed.

5.5.3 Recursive dictionaries

In conjunction with *capping instances* the "knot" of class constraint dependency is "tied" via the *Sat* instance. Also, the capping type—in this case *AlphaEnd*—allows concrete values of the EDT to be created.

A recursive dictionary is created for (and only for) each instance of the capping class. Figure 5.6h graphically represents the structure of the two recursive dictionaries created for the *Exp_0* and *Exp* types. (Interestingly, one of the dictionaries contains the other.) To see how they are built consider what happens when type checking *instance AlphaCap* (*Exp AlphaEnd*). First, we must check if an instance of the superclass exists. The leads to the following constraint chain.

$$\begin{aligned} & \mathit{Alpha\ AlphaEnd\ (Exp\ AlphaEnd)} \\ \rightsquigarrow\ & \mathit{Sat\ (AlphaEnd\ (Exp\ AlphaEnd)} \end{aligned}$$

$\rightsquigarrow$ *AlphaCap* (*Exp AlphaEnd*)

We are back where we started. Fortunately, recursive dictionaries allow such cyclic constraints to be resolved. A similar line of reasoning shows us how the *instance AlphaCap* (*Exp*_0 *AlphaEnd*) is typed and it is graphically represented in Figure 5.6h. The boxes outlined by broken lines represent dictionary transformers (which correspond to instances with contexts). One can also read the solid arrows as *application* to the box at its tip. Following Wadler and Blott's [102] original formulation of dictionary translation we can see the form of the recursive dictionary in d.

d :: *AlphaCapD* (*Exp AlphaEnd*)
d = *AlphaCapD* { *alphaD* = dt_1 (dt_2 d }
dt_1 :: *SatD cxt* (*Exp cxt*) $\rightarrow$ *AlphaD cxt* (*Exp cxt*)
dt_1 = ...
dt_2 :: *AlphaCapD b* $\rightarrow$ *SatD AlphaEnd b*
dt_2 = ...

A full dictionary translation of the code in Figures 5.4a – 5.4g appears in Appendix C.

5.6 Formalisation

In this section I present a formal translation from the language described in Section 5.2 to Haskell. However, so that we may concentrate on the important aspects we translate from an austere source language to a target language equivalent in expressiveness to Haskell. The running example, although legal Haskell, was written in a manner very close to the source language which is essentially the lambda calculus with algebraic data types, flat pattern matching and first order polymorphic types. It does not have type classes. Most importantly, it contains two new forms of algebraic data type declarations: *open data* and *extend data*.

The target language has type classes but the syntactic restrictions on them are less stringent than Haskell 98 and are equivalent to the leniency provided by GHC's "allow undecidable instances" flag. However, there is nothing undecidable about the instances introduced by the translation, they only fail a specific static test implemented in GHC.

The source language does not contain type classes only because they complicate the presentation. There is no reason why they should conflict with the translation.

5.6.1 The source and target languages

Apart from the *open data* and *extend data* declarations the lexical structure of the source language does not differ much from the lambda calculus extended with algebraic data types and pattern matching. However there are a number of non-lexical restrictions on the syntax. These have largely been put in place to simplify the presentation of the translation and, in such cases, other translations from the richer language constructs of full Haskell are known to exist. Some constraints are essential but these have already been

Symbol Classes

α, β, γ	$\rightarrow$	$\langle$type variable$\rangle$
T, E	$\rightarrow$	$\langle$type constructor$\rangle$
$\mathcal{C}, \mathcal{E}$	$\rightarrow$	$\langle$data constructor$\rangle$
x, f	$\rightarrow$	$\langle$term variable$\rangle$
$\overline{\nu}$	$\rightarrow$	$\langle$Collection of pattern variables$\rangle$

Declarations

pgm	$\rightarrow$	$\overline{decl}$	(whole program)
decl	$\rightarrow$	*data*; *tval*	(declaration)
data	$\rightarrow$	**data** $T\,\overline{\alpha} = \mathcal{C}\,\overline{\tau}$	(data type decl)
val	$\rightarrow$	$x = e \mid x\ p = e$	(value binding)
vsig	$\rightarrow$	$x : \sigma$	(type signature)
tval	$\rightarrow$	*vsig* ; $\overline{val}$	(top level binding)

Terms (Expressions)

$e,\ b$	$\rightarrow$	$e_1\ e_2 \mid \lambda x : \tau . e \mid x \mid \mathcal{C}$

Patterns

p	$\rightarrow$	$\mathcal{C}\ x_1\ \ldots\ x_n\ :\ \tau \quad (n \geq 0)$(pattern)

Types

$\tau,\ \xi$	$\rightarrow$	$T \mid \alpha \mid \tau_1\ \tau_2$	(monotype)
σ	$\rightarrow$	$\tau \mid \forall \alpha . \sigma$	(type scheme)

Figure 5.7a: Syntax of source language

enumerated in Section 5.2. This section will only describe those constraints that simplify the presentation.

There is at most one pattern match per function and it must be flat, i.e. not nested. The source language is explicitly typed. All functions have type signatures except new equations on existing functions. This is because signatures already exist for such equations albeit in a different module. It is an error to provide signatures for them.

Further, all value bindings in the source language are supercombinators. We overload the terminology and allow both value bindings and expressions to be supercombinators. An expression that is supercombinator has the form:

$\lambda x_1 : \tau_1 \ldots \lambda x_n : \tau_n . e$

It has the following properties.

- It has no free variables.
- Any sub-term in e that is a lambda abstraction is also a supercombinator.
- $n \geq 0$.

A value binding that is a supercombinator has the form:

$x\ p = \lambda x_1 : \tau_1 \ldots \lambda x_n : \tau_n . e$

- It has no free variables.
- The pattern, p, is optional if the function is not defined on an EDT. Otherwise it is required.

- Any sub-term in e that is a lambda abstraction is a supercombinator.

This restriction was introduced so that it would not be necessary to deal with *let expressions* and *where* clauses. Using lambda-lifting it is always possible to translate from a language containing these to one of supercombinators.

Syntactic conventions

The syntax is provided in Figure 5.7a. Overbar notation is used extensively. The notation $\overline{\alpha}^n$ means the sequence $\alpha_1 \ldots \alpha_n$; the "n" may be omitted when it is unimportant. The following notational shortcuts also apply:

$$\begin{aligned} \overline{\tau}^n \ &\rightarrow\ \xi \equiv \tau_1 \ \rightarrow\ \ldots \ \rightarrow\ \tau_n \ \rightarrow\ \xi \\ \forall\overline{\alpha}^n.\tau \quad &\equiv \forall\alpha_1 \ldots \forall\alpha_n.\tau \end{aligned}$$

Superscripts and subscripts make a difference to what overbars mean. $\overline{D}_i^m\ \delta$ $(1 \leq i \leq m)$ is shorthand for $D_i\ (D_{i+1}\ \ldots\ (D_{m-1}\ D_m)\ldots)$. $\overline{D}^m$ is shorthand for $\overline{D}_1^m\ \delta$. $\overline{D}_i^m \delta$ is the type of an explicit dictionary for functionality class F_i with the explicit dictionaries for functionality classes F_{i+1}, ..., F_m nested within it. Also, we accommodate function types $\tau_1 \ \rightarrow\ \tau_2$ by regarding them as the curried application of the function type constructor to two arguments, thus: $(\rightarrow)\tau_1\tau_2$.

The following conventions apply to the symbols used. The first symbol appearing in each symbol class is a generic symbol. Later symbols in the list often stand for explicit language entities. For example E is reserved for the type constructor of the extensible data type. The concrete symbols are listed in their entirety in Figure 5.7b.

The target language is the same as GHC Haskell 6.4 with the *glasgow extensions*[5] and *allow undecidable instance* options enabled, modulo the syntactic abbreviations we use. In particular, it has type classes, existential types and allows recursive dictionaries to be created during constraint resolution.

5.6.2 The rules

The translation is presented in an inductive manner. The "base case" concerns the translation of the *open data* declaration while the inductive step demonstrates the nth extension of the data type and the mth new function on that data type.

We've already introduced the terms *component type* and *functionality class*, but due to their specific meaning they are summarised again.

- *Component type* – A type that forms part of the EDT. There is the *initial component type* which is introduced when translating the *open data* declaration. Then there are the *extension component types* each introduced with the *extend data* declaration.

[5]We do not even require everything that this enables. We only need multi-parameter type classes, scoped type variables, kind annotations and zero constructor data types.

E	The extensible data type.
$\mathcal{E}_{i,j_i}$	Constructor of EDT ($0 \leq i \leq m, 1 \leq j_i \leq n_i$).
f_{i,k_i}	Function defined on EDT ($0 \leq i \leq m, 1 \leq k_i \leq p_i$).

Figure 5.7b: Concrete symbols of the source language

- *Functionality class* – Classes that provide the functionality for the EDT. They contain functions that correspond to functions written in the source language as well as new functions that result from case translation process. They are introduced as part of the translation.

There are three indexes, i, j_i and k_i used in the translation.

- The index i ranges over the component types and functionality classes. We have made another presentation simplifying assumption that whenever an extension is made to the open data type that a new function is also declared on the EDT[6].
- Index j_i ranges over the variants (constructors) of the component type and has values $1 \leq j_i \leq n_i$, where n_i is the number of variants for the i_{th} component type
- Index k_i ranges over the functions in a functionality class and has values $1 \leq k_i \leq p_i$, where p_i is the number of functions in the i_{th} functionality class.

$\mathcal{T}^{sort}_{description}$ is the way we denote translation rules. The *sort* is the language entity we are doing the translation on. For instance, $\mathcal{T}^{\sigma}_{method}$ transforms σ-types. Some of the translation rules take arguments e.g.$\mathcal{T}^{e}_{unwrap}$. A translation rule can also be mapped over a sequence; this is denoted $\overline{\mathcal{T}^{sort}_{description}}$.

The translation rules use a form of a pattern matching. Most symbols appearing between the Oxford brackets ($[\![\ldots]\!]$) are generic; they bind to whatever is in their position. However, some symbols are concrete and for a match to occur the symbol in the scrutinee of a translation function must match with the symbol in the pattern. Just like Haskell, a pattern match failure means that a match should be attempted on the next translation rule. A list of the concrete symbols for the source language appears in Figure 5.7b.

A syntax has been introduced to range over multiple, similar declarations. An expression of the form $\langle$ *expression*$\rangle^{m}_{j=a}$ means "range over the index j from a to m". There can be nested loops too. An expression of the form $\langle$ *expression* $\rangle^{m,n}_{j=a,k=b}$ means that k ranges over b to n for each j. When seen on the left hand side of a translation rule it *matches* on declarations. On the right hand side it *generates* declarations.

Certain information is required by the translation.

- The name of the extensible data type, denoted E in the translation rules.
- A collection, $\Gamma(E)$, of all type constructors whose definition directly or indirectly contain occurrences of the type constructor E

[6]One could always define an identity function or an empty component type if they didn't want one or the other.

E	Wrapper type for the EDT .
$\mathcal{E}$	Constructor for the wrapper type.
E_i	Component type of EDT.
$\mathcal{E}_{i,j_i}$	Constructor of component type. E_i $(0 \leq i \leq m, 1 \leq j_i \leq n_i)$.
S	*Sat* class.
F_i	Functionality class (for functions f_{i,k_i} $(1 \leq k_i \leq p_i)$.
P	Proxy type.
d_1	Method of S class. Returns explicit dictionary.
d_i	Selector method for next explicit dictionary in explicit dictionary D_{n-1} $(1 \leq i \leq m)$.
D_i	Explicit dictionary for functionality class F_i $(1 \leq i \leq m)$.
$\hat{F}_i$	Capping class for functionality class F_i.
$\hat{D}_i$	Capping type for functionality class F_i.
ε_{i,j_i}	Smart constructor for constructor $\mathcal{E}_{i,j_i}$ $(0 \leq i \leq m, 0 \leq j_i \leq n_i)$.

Figure 5.7c: Concrete symbols of the target language

- A collection, $\Delta(E)$, of all functions that directly or indirectly contain occurrences of a function, f_i $(i \geq 0)$, defined on the EDT, $E\,\overline{\alpha}$.

For example, an analysis on the following module would yield $\Gamma(E) = \{T, T'\}, \Delta(E) = \{g, h\}$.

$$\textbf{open data}\ E\ a\ =\ \ldots$$

$$\begin{aligned} &data\ T\ b\ c\ =\ T_1\ b\ (T'\ c)\\ &data\ T'\ a\ =\ T'_1\ (E\ a)\end{aligned}$$

$$\begin{aligned} &f\ ::\ E\ a\ \rightarrow\ a\\ &f\ =\ \ldots\end{aligned}$$

$$\begin{aligned} &g\ =\ \ldots f\ \ldots\\ &h\ =\ \ldots g\ \ldots\end{aligned}$$

The translation of a module containing an *extend data* requires additional information but we defer discussion of this until Section 5.6.4

5.6.3 Base case: Translating *open data*

The translation of a module containing an *open data* declaration is achieved using the rules in Figures 5.8a and 5.8b. The rule in the former introduces the first component type, E_0, and corresponding smart constructors $\varepsilon_{0,j}$ (for $1 \leq j \leq n_0$), the proxy type P, the base functionality class F_0, the wrapper type **E** and a corresponding unwrapping instance, and an instance for the first functions on the EDT, $f_{0,k}$ (where $1 \leq k \leq p_0$). A capping class,

$\mathcal{T}^{data}[\![\ \mathbf{open\ data}\ E\ \overline{\alpha}\ =\ \langle\ \mathcal{E}_{0,j_0}\ \overline{\tau_j}\ \rangle_{j=1}^{n_0};$
$\langle\ f_{0,k_0}\ :\ \sigma_{0,k_0}\ \rangle_{k_0=1}^{p_0};\ \langle\ f_{0,k_0}\ (\ \mathcal{E}_{0,j_0}\ \overline{\nu_{j_0}}\ :\ E\ \overline{\xi_{j_0}}\ )\ =\ b_{0,j_0,k_0}\ \rangle_{j=1,k_0=1}^{n_0,p_0}\ ]\!] =$

Proxy type
$\mathbf{data}\ P\ \delta\ ;$

Sat class
$\mathbf{class}\ S\ \alpha\ \mathbf{where}$
$d_1\ :\ \alpha$

Wrapper type
$\mathbf{data}\ \mathbf{E}\ (\delta\ :\ \mathcal{T}_{kind}^{\overline{\alpha}}[\![\overline{\alpha}]\!])\ \overline{\alpha}\ =\ \exists\beta.F_0\ \delta\ \beta\ \Rightarrow\ \mathcal{E}\ (\beta\ \overline{\alpha});$

Initial component type
$\mathbf{data}\ E_0\ \delta\ \overline{\alpha}\ =\ \langle\ \mathcal{E}_{0,j_0}\ \overline{\mathcal{T}_E^{\tau}[\![\overline{\tau_{j_0}}]\!]}\ \rangle_{j_0=1}^{n_0}\ ;$

Initial functionality class
$\mathbf{class}\ S\ (\delta\ \beta)\ \Rightarrow\ F_0\ \delta\ \beta\ \mathbf{where}$
$\langle\ f_{0,k_0}\ :\ \mathcal{T}_{method}^{\sigma}(0,k_0)[\![\sigma_{0,k_0}]\!]\ \rangle_{k=1}^{p_0}\ ;$

Initial functionality instance
$\mathbf{instance}\ (S\ (\delta\ (\mathbf{E}\ \delta)),\ S\ (\delta\ (E_0\ \delta)))\ \Rightarrow\ F_0\ \delta\ (E_0\ \delta)\ \mathbf{where}$
$\langle\ f_{0,k_0}\ (\_ : P\ \delta)\ (E_0\ \overline{\nu_{j_0}}\ :\ E_{0,j_0}\ \delta\ \overline{\xi_{j_0}})\ =\ \mathcal{T}_{method}^{e}[\![b_{0,j_0,k_0}]\!]\ \rangle_{j_0=1,k_0=1}^{n_0,p_0}\ ;$

Unwrapping instance
$\mathbf{instance}\ S\ (\delta\ (\mathbf{E}\ \delta))\ \Rightarrow\ F_0\ \delta\ (\mathbf{E}\ \delta)\ \mathbf{where}$
$\langle\ f_{0,k_0}\ (\_ : P\ \delta)\ (\ \mathcal{E}\ x\ :\ \mathbf{E}\ \delta\ \overline{\xi})\ =\ \mathcal{T}_{unwrap}^{\sigma}(f_{0,k_0},\ x, P\ \delta)[\![\sigma_{0,k_0}]\!]\ \rangle_{k=1}^{p_i}$

Capping class, type and instances
$\mathbf{data}\ \hat{D}_0\ \beta\ ;$
$\mathbf{class}\ F_0\ \hat{D}_0\ \beta\ \Rightarrow\ \hat{F}_0\ \beta\ ;$
$\mathbf{instance}\ \hat{F}_0\ (\mathbf{E}\ (\hat{D}_0))\ ;$
$\mathbf{instance}\ \hat{F}_0\ (E_0\ (\hat{D}_0))\ ;$
$\mathbf{instance}\ \hat{F}_0\ \beta\ \Rightarrow\ S\ (\hat{D}_0\ \beta)\ \mathbf{where}$
$d_1\ =\ \bot$

Smart constructors
$\langle\ \varepsilon_{0,j_0} : \mathcal{T}_{smart}^{\overline{\tau}}[\![\overline{\tau_{j_0}}]\!];\ \varepsilon_{0,j_0}\ (\_ : P\ \delta)\ =\ \mathcal{T}_{smart}^{e}(0,j_0)[\![\overline{\tau_{j_0}}]\!]\ \rangle_{j_0=1}^{n_0}$

Figure 5.8a: Translation for *open data* declaration in the initial module ($m = 0$).

$$\mathcal{T}^{data}[\![\mathbf{data}\ T\ \overline{\alpha}\ =\ \mathcal{C}\ \overline{\tau}]\!] = \begin{cases} \mathbf{data}\ T\ \delta\ \overline{\alpha}\ =\ \mathcal{C}\ \overline{\mathcal{T}^{\tau}_{E}[\![\overline{\tau}]\!]} & \text{, if } T\ \in\ \Gamma(E) \\ \mathbf{data}\ T\ \overline{\alpha}\ =\ \mathcal{C}\ \overline{\tau} & \text{, otherwise} \end{cases}$$

$$\mathcal{T}^{data}[\![\mathbf{type}\ T\ \overline{\alpha}\ =\ \overline{\tau}]\!] = \overline{\mathcal{T}^{\tau}_{E}[\![\overline{\tau}]\!]}$$

$$\mathcal{T}^{tval}[\![x\ :\ \sigma;\ \overline{val}]\!] = x\ :\ \mathcal{T}^{\sigma}_{E}[\![\sigma]\!]\ ;\ \overline{\mathcal{T}^{val}[\![\overline{val}]\!]}$$

$$\mathcal{T}^{val}[\![x\ p\ =\ e]\!] = \begin{cases} x\ (\_ : P\ (\overline{D}^{m}\delta_m)\ p\ =\ \mathcal{T}^{e}_{method}[\![e]\!] & \text{, if } x \in \Delta(E) \\ x\ p\ =\ e & \text{, otherwise} \end{cases}$$

$$\mathcal{T}^{val}[\![x\ =\ e]\!] = \begin{cases} x\ (\_ : P\ (\overline{D}^{m}\delta_m)\ =\ \mathcal{T}^{e}_{method}[\![e]\!] & \text{, if } x \in \Delta(E) \\ x\ =\ e & \text{, otherwise} \end{cases}$$

$$\mathcal{T}^{\sigma}_{E}[\![\forall\ \overline{\alpha}.\tau]\!] = \begin{cases} \forall\ \overline{\alpha}.\ S\ (\overline{D}^{m}\delta_m\ (\ \mathbf{E}\ (\overline{D}^{m}\delta_m))) \Rightarrow \\ \qquad \mathcal{T}^{\tau}_{E}[\![\tau]\!] & \text{, if } x \in \Delta(E) \\ \mathcal{T}^{\tau}_{E}[\![\tau]\!] & \text{, otherwise} \end{cases}$$

Figure 5.8b: Translation for regular declarations in the m_{th} module.

$$\begin{aligned}
&\mathcal{T}^{data}[\![\ \langle\ f_{0,k_0}\ (\ \mathcal{E}_{m,j_m}\ \overline{\nu_{j_m}}\ :\ E\ \overline{\xi_{j_m}})\ =\ b_{0,j_m,k_0}\ \rangle^{n_m,p_0}_{j_m=1,k_0=1};\\
&\qquad \ldots\ ;\\
&\qquad \langle\ f_{m-1,k_{m-1}}\ (\ \mathcal{E}_{m,j_m}\ \overline{\nu_{j_m}}\ :\ E\ \overline{\xi_{j_m}})\ =\ b_{m-1,j_m,k_{m-1}}\ \rangle^{n_m,p_{m-1}}_{j_m=1,k_{m-1}=1}\ ]\!] =\\
&\quad \langle\ \mathbf{instance}\ (\ S\ (\overline{D}^{m}\delta_m\ (\mathbf{E}\ (\overline{D}^{m}\delta_m)))\\
&\qquad\qquad ,\ S\ (\overline{D}^{m}\delta_m\ (E_0\ (\overline{D}^{m}\delta_m)))\\
&\qquad\qquad ,\ \ldots\\
&\qquad\qquad ,\ S\ (\overline{D}^{m}\delta_m\ (E_m\ (\overline{D}^{m}\delta_m)\ ))\\
&\qquad\qquad)\ \Rightarrow\ F_i\ (\overline{D}^{m}_{i}\delta_m)\ (E_m\ (\overline{D}^{m}\delta_m))\ \mathbf{where}\\
&\qquad \langle\ f_{i,k_i}\ (_ : P\ (\overline{D}^{m}\delta_m))\\
&\qquad\qquad (\ \mathcal{E}_{m,j_m}\ \overline{\nu_{j_m}}\ :\ E_m\ (\overline{D}^{m}\delta_m)\ \overline{\xi_{j_m}})\ =\ \mathcal{T}^{e}_{method}[\![b_{i,j_m,k_i}]\!]\ \rangle^{n_m,p_i}_{j_m=1,k_i=1}\\
&\rangle^{m-1}_{i=0}
\end{aligned}$$

Figure 5.8c: Translation for new equations on existing functions in the m_{th} extension module.

$\hat{F}_0$, and capping type, $\hat{D}_0$ are introduced. (There is no explicit dictionary for the base functionality class.) The *Sat* class, S is also introduced, once and for all.

Smart constructors are introduced so that the translation of regular data constructors in the source language is simplified; an occurrence of a constructor becomes a smart constructor instead. The proxy type is introduced for the purposes of type checking in the target language. Without it, the type checker does not have enough information. This is discussed in the next section. An extra argument of the proxy type is added for all functions, $f_{i,k}$, defined on the EDT and to the smart constructors.

5.6.4 Inductive step: Translating *extend data*

The translation of a module containing an *extend data* declaration is done using the rules in Figures 5.8b, 5.8c, and 5.8d. These rules introduce the mth new variant on the EDT and the mth function. It is assumed that the following information is available.

$$\mathcal{T}^{data}[\![\ \textbf{extend data}\ E\ \overline{\alpha}\ =\ \langle\ \mathcal{E}_{m,j}\ \overline{\tau_j}\ \rangle_{j_m=1}^{n_m}\ ;$$
$$\langle\ f_{m,k_m}\ :\ \sigma_{m,k_m}\ \rangle_{k_m=1}^{p_m};\ \langle\ f_{m,k_m}\ (\ \mathcal{E}_{m,j_m}\ \overline{\nu_{m,j_m}}\ :\ E\ \overline{\xi_{m,j_m}})\ =\ b_{m,j_m,k_m}\ \rangle_{j_m=1,k_m=1}^{n_m,p_m}\ ]\!] =$$

The m_{th} extension component type
$\textbf{data}\ E_m\ (\delta\ :\ \mathcal{T}^{\overline{\alpha}}_{kind}[\![\overline{\alpha}]\!])\ \overline{\alpha}\ =\ \langle\ \mathcal{E}_{m,j_m}\ \overline{\mathcal{T}^{\tau}_{E}[\![\overline{\tau_j}]\!]}\ \rangle_{j_m=1}^{n_m};$

The m_{th} functionality class
$\textbf{class}\ (S\ (\overline{D}^m\delta_m\ \beta),\ F_{m-1}\ (D_m\ \delta_m)\ \beta)\ \Rightarrow\ F_m\ \delta_m\ \beta\ \textbf{where}$
$\langle\ f_{m,k_m}\ :\ \mathcal{T}^{\sigma}_{method}[\![\sigma_{m,k_m}]\!]\ \rangle_{k_m=1}^{p_m};$

The m_{th} explicit dictionary
$\textbf{data}\ D_m\ \delta_m\ \beta\ =\ \mathcal{D}_m\ \{\ \langle\ f'_{m,k_m}\ :\ \mathcal{T}^{\sigma}_{method}[\![\sigma_{m,k_m}]\!];\ \rangle_{k_m=1}^{p_m}\ ;$
$,\ d_{m+1}\ :\ (\delta_m\ \beta)\ \}$

Unwrapping instance
$\textbf{instance}\ S\ (\overline{D}^m\delta_m\ (\mathbf{E}\ (\overline{D}^m\delta_m)))\ \Rightarrow\ F_m\ \delta_m\ (\mathbf{E}\ (\overline{D}^m\delta_m))\ \textbf{where}$
$\langle\ f_{m,k_m}\ (\_ : P\ (\overline{D}^m\delta_m))\ (\ \mathcal{E}\ x\ :\ \mathbf{E}\ (\overline{D}^m\delta_m)\ \overline{\xi})\ =$
$\mathcal{T}^{\sigma}_{unwrap}(f_{m,k_m},\ x,\ P\ (\overline{D}^m\delta_m))[\![\sigma_{m,k_m}]\!]\ \rangle_{k_m=1}^{p_m}$

Functionality instances (for component types $0 \le i \le m$)
$\langle\ \textbf{instance}\ (\ S\ (\overline{D}^m\delta_m\ (\mathbf{E}\ (\overline{D}^m\delta_m)\ ))$
$,\ S\ (\overline{D}^m\delta_m\ (E_0\ (\overline{D}^m\delta_m)\ ))$
$,\ \ldots$
$,\ S\ (\overline{D}^m\delta_m\ (E_m\ (\overline{D}^m\delta_m)\ ))$
$)\ \Rightarrow\ F_m\ \delta_m\ (E_i\ (\overline{D}^m\delta_m)\ )\ \textbf{where}$
$\langle\ f_{m,k_m}\ (\_ : P\ (\overline{D}^m\delta_m))\ (\ \mathcal{E}_{i,j_i}\ \overline{\nu_{j_i}}\ :\ E_i\ (\overline{D}^m\delta_m)\ \overline{\xi_{j_i}})\ =\ \mathcal{T}^{e}_{method}[\![b_{i,j_i,k_m}]\!]\ \rangle_{j_i=1,k_m=1}^{n_i,p_m}$
$\rangle_{i=0}^{m}$

Capping class, type and instances
$\textbf{data}\ \hat{D}_m\ \beta\ ;$
$\textbf{class}\ F_m\ \hat{D}_m\ \beta\ \Rightarrow\ \hat{F}_m\ \beta\ ;$
$\textbf{instance}\ \hat{F}_m\ (\mathbf{E}\ (\overline{D}^m\ \hat{D}_m))\ ;$
$\langle\ \textbf{instance}\ \hat{F}_m\ (E_i\ (\overline{D}^m\ \hat{D}_m))\ ;\ \rangle_{i=0}^{m}$

"Knot tying" instance
$\textbf{instance}\ \hat{F}_m\ \beta\ \Rightarrow\ S\ (\overline{D}^m\ \hat{D}_m\ \beta)\ \textbf{where}$
$d_1\ =\ D_1\ \{\ \langle\ f'_{1,k}\ =\ f_{1,k}\ \rangle_{k=1}^{p_1}$
$d_2\ =\ D_2\ \{\ \langle\ f'_{2,k}\ =\ f_{2,k},\ \rangle_{k=1}^{p_2}$
$\ldots$
$\ldots\ d_m\ =\ D_m\ \{\ \langle\ f'_{m,k}\ =\ f_{m,k}\ \rangle_{k=1}^{p_m},\ d_{m+1}\ =\ \bot\}\ \ldots\ \}$

Smart constructors
$\langle\ \varepsilon_{m,j_m} : \mathcal{T}^{\overline{\tau}}_{smart}[\![\overline{\tau_{j_m}}]\!];\ \varepsilon_{m,j_m}\ (\_ : P\ \delta)\ =\ \mathcal{T}^{e}_{smart}(m,j_m)[\![\overline{\tau_{j_m}}]\!]\ \rangle_{j_m=1}^{n_m}$

Figure 5.8d: Translation for *extend data* declaration and new function for the m_{th} extension module.

$$\mathcal{T}^{\overline{\alpha}}_{kind}[\![\alpha_1,\ \ldots,\ \alpha_k]\!] = \overbrace{(\star \rightarrow \ \ldots\ \rightarrow \star)}^{k+1} \rightarrow \star$$

$$\begin{array}{ll}
\mathcal{T}^{\sigma}_{method}[\![\forall\overline{\alpha}.E\,\overline{\tau} \rightarrow \xi]\!] & = \forall\overline{\alpha}.\, P(\overline{D}^m\delta_m) \rightarrow \beta\,\overline{\tau} \rightarrow \mathcal{T}^{\tau}_{method}[\![\xi]\!] \\
\mathcal{T}^{\tau}_{method}[\![\alpha]\!] & = \alpha \\
\mathcal{T}^{\tau}_{method}[\![T]\!] & = \begin{cases} T\ (\overline{D}^m\delta_m) & \text{, if } T \in \Gamma(E) \\ T & \text{, otherwise} \end{cases} \\
\mathcal{T}^{\tau}_{method}[\![E]\!] & = E\ (\overline{D}^m\delta_m) \\
\mathcal{T}^{\tau}_{method}[\![\tau_1\ \tau_2]\!] & = \mathcal{T}^{\tau}_{method}[\![\tau_1]\!]\ \mathcal{T}^{\tau}_{method}[\![\tau_2]\!] \\
\\
\mathcal{T}^{e}_{method}[\![f_{i,k_i}]\!] & = \begin{cases} f'_{i,k_i}\ \mathcal{T}^{dict}(i)\ (\bot : P\ (\overline{D}^m\delta_m)) & \text{, if } i > 0 \\ f_{i,k_i}\ (\bot : P\ (\overline{D}^m\delta_m)) & \text{, otherwise} \end{cases} \\
\mathcal{T}^{e}_{method}[\![x]\!] & = \begin{cases} x\ (\bot : P\ (\overline{D}^m\delta_m)) & \text{, if } x \in \Delta(E) \\ x & \text{, otherwise} \end{cases} \\
\mathcal{T}^{e}_{method}[\![\lambda x\ :\ \tau.e]\!] & = \lambda x\ :\ \mathcal{T}^{\tau}_{method}[\![\tau]\!].\ \mathcal{T}^{e}_{method}[\![e]\!] \\
\mathcal{T}^{e}_{method}[\![\ \mathcal{E}_{i,j_i}]\!] & = \varepsilon_{i,j_i}\ (\bot :\ P\ (\overline{D}^m\delta_m)) \\
\mathcal{T}^{e}_{method}[\![\ \mathcal{C}]\!] & = \mathcal{C} \\
\mathcal{T}^{e}_{method}[\![e_1\ e_2]\!] & = \mathcal{T}^{e}_{method}[\![e_1]\!]\mathcal{T}^{e}_{method}[\![e_2]\!]
\end{array}$$

$$\mathcal{T}^{\sigma}_{unwrap}(f_{i,k_i}, x, \gamma)[\![\forall\overline{\alpha}.\, E\,\overline{\tau} \rightarrow \tau_1 \rightarrow \ \ldots\ \rightarrow \tau_n \rightarrow \tau_{n+1}]\!] =$$

$$\begin{cases}
\lambda x_1 : \mathcal{T}^{\tau}_{method}[\![\tau_1]\!] \\
\ldots \\
\lambda x_n : \mathcal{T}^{\tau}_{method}[\![\tau_n]\!].f_0\ (\bot : \gamma)\ x\ x_1\ \ldots\ x_n & \text{, if } i = 0 \\
\lambda x_1 : \mathcal{T}^{\tau}_{method}[\![\tau_1]\!] \\
\ldots \\
\lambda x_n : \mathcal{T}^{\tau}_{method}[\![\tau_n]\!].f_{i,k_i}\ \mathcal{T}^{dict}(i)\ (\bot : \gamma)\ x\ x_1\ \ldots\ x_n & \text{, otherwise}
\end{cases}$$

Figure 5.8e: Translation rules

$$
\begin{array}{lll}
\mathcal{T}^{\overline{\tau}}_{smart}[\![\overline{\tau}]\!] & = \forall\, \delta_m . \forall\, \overline{\alpha} .\, (\ S\ (\overline{D}^m \delta_m\ (\mathbf{E}\ (\overline{D}^m \delta_m))) \\
 & \qquad\qquad , \ S\ (\overline{D}^m \delta_m\ (E_0\ (\overline{D}^m \delta_m))) \\
 & \qquad\qquad , \ \ldots \\
 & \qquad\qquad , \ S\ (\overline{D}^m \delta_m\ (E_m\ (\overline{D}^m \delta_m)))) \ \Rightarrow \\
 & \quad \mathit{connect}\ (P\ (\overline{D}^m \delta_m),\ \overline{\mathcal{T}^{\tau}_{method}}\ [\![\ \overline{\tau}\ ]\!],\ \mathbf{E}\ (\overline{D}^m \delta_m)\ \overline{\alpha}) \\
 & \\
\mathcal{T}^{\tau}_{smart}[\![T]\!] & = \begin{cases} T\ (\overline{D}^m \delta_m) & \text{, if } T \in \Gamma(E) \\ T & \text{, otherwise} \end{cases} \\
\mathcal{T}^{\tau}_{smart}[\![E]\!] & = \mathbf{E}\ (\overline{D}^m \delta_m) \\
\mathcal{T}^{\tau}_{smart}[\![\tau_1\ \tau_2]\!] & = \mathcal{T}^{\tau}_{smart}[\![\tau_1]\!]\ \mathcal{T}^{\tau}_{smart}[\![\tau_2]\!] \\
 & \\
\mathit{connect}\ [\![\ \tau_1,\ \ldots,\ \tau_k\ ]\!] & = \tau_1\ \rightarrow\ \cdots \rightarrow\ \tau_k \\
 & \\
\mathcal{T}^{e}_{smart}(i,j)[\![\tau_1,\ \ldots,\ \tau_k]\!] & = \lambda x_1 : \tau_1 \ldots \lambda x_k : \tau_k .\, \mathcal{E}\ (\ \mathcal{E}_{i,j}\ x_1\ \ldots\ x_k : E_{i,j}\ (\overline{D}^i \delta_i)\ \overline{\alpha})
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{T}^{dict}(i) & = (d_i\ (\ldots\ (d_2\ d_1) \ldots)) \\
\mathcal{T}^{\tau}_{E}[\![\alpha]\!] & = \alpha \\
\mathcal{T}^{\tau}_{E}[\![E]\!] & = \mathbf{E}\ \delta \\
\mathcal{T}^{\tau}_{E}[\![T]\!] & = \begin{cases} T\ (\overline{D}^m \delta_m) & \text{, if } T \in \Gamma(E) \\ T & \text{, otherwise} \end{cases} \\
\mathcal{T}^{\tau}_{E}[\![\tau_1\ \tau_2]\!] & = \mathcal{T}^{\tau}_{E}[\![\tau_1]\!]\ \mathcal{T}^{\tau}_{E}[\![\tau_2]\!]
\end{array}
$$

Figure 5.8f: More translation rules

- A list of m existing functionality classes $[F_0, ..., F_{m-1}]$, functions $[f_{0,k_0}, \ldots, f_{m-1,k_{m-1}}]$ (where $1 \leq k_i \leq p_i$) and explicit dictionaries $[D_0, \ldots, D_{m-1}]$.
- A list of m existing component types $[E_0, ..., E_{m-1}]$ and the variant constructors $[\mathcal{E}_{0,j_0}, \ldots, \mathcal{E}_{m-1,j_{m-1}}]$ (where $1 \leq j_i \leq n_i$) .
- A list of capping classes, $[\hat{F}_1, \ldots, \hat{F}_{m-1}]$ and capping types, $[\hat{D}_1, \ldots, \hat{D}_{m-1}]$. A capping type is just a zero constructor dummy type.

How this information would be acquired and passed around in an implementation is discussed in Section 5.6.7.

Similar to the base case, the rule that transforms an *extend data* declaration introduces a new component type and smart constructor, a new functionality class, function, and capping class. An instance is introduced for each existing component type and the newly introduced one.

This rule does more. It introduces instances to handle new equations on old functions (i.e. f_{i,j_i} $(i < m,\ 1 \leq j_i \leq n_i)$). (Remember, there is a syntactic restriction on the source language specifying that these must have been declared.) This rule also brings into being an explicit dictionary and associated capping type.

In many ways the inductive step of the translation is much more interesting. Consequently we spend some time explaining the subtleties of the rules.

The need for proxy arguments

Proxy arguments are required in order to guide the type checker for the target language. Consider the following function in the source language:

$$
\begin{aligned}
&\textbf{data}\ E\ =\ \mathcal{E}_0\ \mathit{String}\ (E\ \mathit{String})\\
&f_{0,1}\ ::\ E\ \rightarrow\ \mathit{String}\\
&f_{0,1}\ (\ \mathcal{E}_0\ s\ e)\ =\ s\ +\!\!+\ f_{0,1}\ e
\end{aligned}
$$

Now consider what we would get if the translation omitted to add proxy arguments.

$$
\begin{aligned}
&\textbf{class}\ S\ (\ \delta\ \beta)\ \Rightarrow\ F_0\ \delta\ \beta\ \textbf{where}\\
&\quad f_0\ ::\ \beta\ \rightarrow\ \mathit{String}\\
&\\
&\textbf{instance}\ S\ (\delta\ (\ \mathbf{E}\ \delta))\ \Rightarrow\ F_0\ \delta(\ \mathbf{E}\ \delta)\ \textbf{where}\\
&\quad f_{0,1}\ (\ \mathcal{E}\ x)\ =\ f_{0,1}\ x\\
&\textbf{instance}\ (S\ (\delta\ (\ \mathbf{E}\ \delta)),\ S\ (\delta\ (E_0\ \delta)))\ \Rightarrow\ F_0\ \delta\ (E_0\ \delta)\ \textbf{where}\\
&\quad f_{0,1}\ (\ \mathcal{E}_0\ s\ e)\ =\ s\ +\!\!+\ f_{0,1}\ e
\end{aligned}
$$

Among the constraints raised by the use of $f_{0,1}$ on the right hand side of the instance method equation is $F_0\ \delta'\ (\ \mathbf{E}\ \delta)$. The problem is that the δ' and δ aren't equal. The proxy ensures that they are equated. To see this consider the translation with proxy arguments attached.

$$\begin{array}{l}\textbf{class}\ S\ (\ \delta\ \beta)\ \Rightarrow\ F_0\ \delta\ \beta\ \textbf{where}\\ \quad f_{0,1}\ ::\ P\ \delta\ \rightarrow\ \beta\ \rightarrow\ \mathit{String}\end{array}$$

$$\begin{array}{l}\textbf{instance}\ (S\ (\delta\ (\ \mathbf{E}\ \delta)),\ S\ (\delta\ (E_0\ \delta)))\ \Rightarrow\ F_0\ \delta\ (E_0\ \delta)\ \textbf{where}\\ \quad f_{0,1}\ (\_\ :\ P\ \delta)\ (\ \mathcal{E}_0\ s\ e)\ =\ s\ +\!\!+\ f_{0,1}\ (\bot\ :\ P\ \delta)\ e\end{array}$$

The constraint raised by the expression $f_{0,1}\ (\bot\ :\ P\ \delta)\ e$ is now $F_0\ \delta\ (\ \mathbf{E}\ \delta)$.

S constraints in instance heads

The instance heads for new equations on existing component types and the instance heads for new functions both contain many occurrences of S constraints. This may seems strange considering that each functionality class has S as a superclass. The reason is that the S instance that "ties the knot" will be declared at some point in the future (possibly in another module). The S constraints in the instance head "promise" that this will happen.

These constraints mention the latest explicit dictionary (i.e. $\overline{D}^m$). The purpose of this is to allow the body of the instance method to contain occurrences of any of the functions so far ($f_{1,k_1}, \ldots, f_{m-1,k_{m-1}}$) *and* the latest ones – f_{m,k_m}). This is possible even inside new equations on existing functions, which may seem counter-intuitive at first. To see why consider the translation of the following new equation where $a < m$, $b <= m$, and $a < b$.

$$\begin{array}{l}\mathcal{T}^{data}[\![f_{a,1}\ (\ \mathcal{E}_{m,2}\ x)\ =\ \ldots\ f_{b,1}\ \ldots]\!]\ =\\ \ldots\\ \textbf{instance}\ (\ S\ (\overline{D}^m\ \delta_m\ (\ \mathbf{E}\ (\overline{D}^m\ \delta_m)))\\ \qquad\qquad ,\ S\ (\overline{D}^m\ \delta_m\ (E_0\ (\overline{D}^m\ \delta_m)))\\ \qquad\qquad ,\ \ldots\\ \qquad\qquad ,\ S\ (\overline{D}^m\ \delta_m\ (E_m\ (\overline{D}^m\ \delta_m)))\\ \qquad\qquad)\ \Rightarrow\ F_a\ (\overline{D}^m_{a+1}\ \delta_m)\ E_m\ (\overline{D}^m\ \delta_m)\ \textbf{where}\\ f_{a,1}\ (\ \mathcal{E}_{m,2}\ x)\ =\ \ldots\ f'_{b,1}\ (d_m\ \ldots\ (d_2\ d_1)\ldots)\ \ldots\\ \ldots\end{array}$$

The expression $f'_{b,1}\ (d_b\ \ldots\ (d_2\ d_1)\ldots)$ raises the following constraints.

$$\begin{array}{l}\quad S\ (\overline{D}^m\ \hat{D}_m\ (E_m\ (\overline{D}^m\ \hat{D}_m)))\\ \rightsquigarrow\ \hat{F}_m\ (E_m\ (\overline{D}^m\ \hat{D}_m))\end{array}$$

This instance for the capping class has been declared. The considerably involved way in which this is type checked is covered in the next section.

Capping classes

Instances of the capping class, and the associated S instance, are used to "tie the knot" during constraint resolution. They do this, not just for the m_{th} functionality class, but for all the others.

For each of the capping class instances we need to check for the existence of an instance of its super class, the m_{th} functionality class. Because constraint resolution is cycle aware

we first add the constraints $\hat{F}_m\ (E_i\ (\overline{D}^m \hat{D}_m))$ (for $0 \leq i \leq m$) to the current collection of assumptions. (Each of these constraints will only be resolved if a chain of resolutions reaches it again.) Now let's consider a particular superclass constraint for component type E_b (for some $0 \leq b \leq m$). It produces $m+1$ S constraints.

$$\begin{aligned} & F_m\ \hat{D}_m\ (E_b\ (\overline{D}^m\ \hat{D}_m)) \\ \rightsquigarrow\ & S\ (\overline{D}^m\ \hat{D}_m\ (E_j\ (\overline{D}^m\ \hat{D}_m))) \text{ for each } (0 \leq j \leq m) \end{aligned}$$

Each one of these S constraint is resolved by

$$\begin{aligned} & S\ (\overline{D}^m\ \hat{D}_m\ (E_j\ (\overline{D}^m\ \hat{D}_m))) \\ \rightsquigarrow\ & \hat{F}_m\ (E_j\ (\overline{D}^m\ \hat{D}_m)) \end{aligned}$$

But these are in the collection of assumptions, so they get resolved. Thus, a recursive dictionary is created for each component type and the wrapper type. This "ties the knot" for *all* of the functionality classes, not just the m_{th} one. To see why, consider how we type check

$f'_{a,c}\ (d_i\ \ldots\ (d_2\ d_1)\ldots)\ ::\ E_b\ (\overline{D}^m\ \hat{D}^m)\ \rightarrow\ \ldots$

for some $0 \leq a \leq m$, $0 \leq b \leq m$, and $0 \leq c \leq p_i$. This leads to the following constraint resolution. (The initial constraint comes from substituting $\delta = \overline{D}^m_{i+1}\ \hat{D}_m$ into $S\ (\overline{D}^i\ \delta\ (E_b\ (\overline{D}^i\ \delta)))$.

$$\begin{aligned} & S\ (\overline{D}^m\ \hat{D}_m\ (E_b\ (\overline{D}^m\ \hat{D}_m))) \\ \rightsquigarrow\ & \hat{F}_m\ (E_b\ (\overline{D}^m\ \hat{D}_m)) \end{aligned}$$

But this constraint has been provided by the capping class instance which type checks for the reasons stated earlier.

Translating regular declarations

Any type declaration in the source language that directly or indirectly contains a reference to E must be translated to contain an occurrence of the wrapper data type **E** in the target language. Any function which directly or indirectly contains an occurrence of a function $f_{i,k}$ (for a specific $0 \leq i \leq m, 0 \leq k \leq p_i$) must also have its body transformed to contain an occurrence of $f'_{i,k}\ d_i\ (\ldots\ (d_2\ d_1)\ldots)$. More importantly, an S constraint must be added to the type. However, this only needs to be done for the wrapper data type, **E**, as these are the only values that will be passed to such functions. S constraints on component types are only ever seen in the class and instance heads of functionality classes.

5.6.5 The link between the formalisation and the running example

To further the reader's understanding they are encouraged to apply the rules from Figures 5.8a and 5.8b to the initial module of the running example (Figure 5.1) to yield the result in Figures 5.4a–5.4f. By applying the rules in Figures 5.8c, 5.8d and 5.8b to to Figure 5.2 they will get the result in Figures 5.5a–5.5h.

S	*Sat*	F_1	*Eval*
δ_i	*cxt*	$f_{1,1}$	*eval*
β	*b*	$f_{1,2}$	*apply*
$\mathbf{E}$	*Exp*	$\hat{F}_0$	*AlphaCap*
E_0	*Exp_0*	$\hat{D}_0$	*AlphaEnd*
F_0	*Alpha*	d_1	*dict*
$\mathcal{E}$	*MkExp*	d_2	*expExt*
$\mathcal{E}_{0,1}$	*Var*	D_1	*Eval*
$\mathcal{E}_{0,2}$	*Lam*	$\hat{F}_1$	*EvalCap*
$\mathcal{E}_{0,3}$	*App*	$\hat{D}_1$	*EvalEnd*
$f_{0,1}$	*alpha*	$(n_0 = 3,\ p_0 = 1,$	$n_1 = 1,\ p_1 = 2)$
$\mathcal{E}_{1,1}$	*LetE*	$(m = 2)$	
E_1	*Exp_1*		

Figure 5.9: A mapping from symbols in the formal translation to identifiers in the running example.

However, the translation rules use an abbreviated syntax. In order to aid the reader Figure 5.9 shows the correspondence between the abbreviated syntax and the syntax used in the translation of the running example.

5.6.6 Creating values of the EDT

For an extensible data type to be created and used, it is necessary for the translator to insert the latest capping type in place of the δ_m in the type $\mathbf{E}\,(\overline{D}^m)\,\delta_m\overline{\tau}$). However, the only functions that can be called on a value of this type are those that have the constraint $S\,(\overline{D}^a\,\delta_a\,(\,\mathbf{E}\,\overline{D}^a\,\delta_a))$ (where $a \leq m$) in their types. The translation ensures that the functions f_{i,k_i} (where $0 \leq i \leq m, 1 \leq k_i \leq p_i$) and any function which directly or indirectly call them satisfy this condition.

5.6.7 Implementation

At this point, an implementation of this translation does not yet exist. However, these rules were applied by hand to implement extensible abstract syntax in the PHRAC compiler in Chapter 6. In this section I sketch some of the features that an implementation would need to have.

For the translation rules to be applied certain information is assumed to be available. The information specific to each rule has already been covered in a piecemeal fashion in the preceding text. Here is a summary:

- A list of m existing functionality classes $[F_0, \ldots, F_{m-1}]$, functions $[f_{0,k_0}, \ldots, f_{m-1,k_{m-1}}]$ (for each $1 \leq k_i \leq p_i$) and explicit dictionaries $[D_0, \ldots, D_{m-1}]$.
- A list of n existing component types $[E_0, ..., E_{m-1}]$ and the variant constructors $[\mathcal{E}_{0.j_0}, \ldots, \mathcal{E}_{m-1,j_{m-1}}]$ (for each $1 \leq j_i \leq n_i$).
- A list of capping classes, $[\hat{F}_1, \ldots, \hat{F}_{m-1}]$ and capping types, $[\hat{D}_1, \ldots, \hat{D}_{m-1}]$.

- A list of functions that either directly or indirectly reference a function on the EDT. This is contained in the function environment, $\Gamma(E)$, which is just a list of names.

- A list of types the either directly or indirectly reference the EDT type. This is a list of type constructors: $\Delta(E)$.

In the presence of separate compilation the constraint resolution for type classes requires that interface files be generated for each module. Since the encoding of EDTs relies heavily upon constraint resolution a natural place to put the information required is in the interface files.

5.7 Pattern matching and Binary Functions

There are two remaining questions. The first is how pattern matching is done on Haskell augmented with syntactic sugar for open abstract types. The second is, can we handle binary functions? This section provides answers to both these questions.

5.7.1 Stratified pattern matching

For Haskell to be augmented with *open data* and *extend data* declarations it is also necessary to introduce a slight change to the pattern matching semantics for equations written on open data types. Consider the following declaration:

module $M1$ **where**
open data $E\ =\ \mathcal{E}_0\ String$

...
module $M2$ **where**
extend data $E\ =\ \mathcal{E}_1\ Int$

The translation would create two component types, E_0 and E_1 to contain the constructors $\mathcal{E}_0$ and $\mathcal{E}_1$. When a function is called on a value of type **E** in the target language the value is unwrapped and the internal value dispatched to the appropriate instance. It is only at this point that Haskell's regular pattern matching algorithm, which dispatches to the body associated with the first matching pattern, would come into effect.

From the perspective of augmented Haskell this amounts to a slightly different form of pattern matching: *stratified pattern matching*. The semantics of pattern matching in augmented Haskell is as follows. Equations in the modules are grouped according to which *open data* or *extend data* declaration introduced the constructors they are defined over. The relative order is preserved. Pattern matching is done by first finding the *group* of equations corresponding to the value a function is being applied to and then doing pattern matching as usual *within that group*.

5.7.2 Binary functions

An important question that arises repeatedly when evaluation solutions to the expression problem is "can one write extensible binary functions?" By binary function we mean those that take two occurrences of the open data type as parameters (in addition to any other parameters). Although our source language is restricted to at most one pattern match per function it is possible to encode binary functions simply by dispatching to others. The technique is illustrated now. Consider a data type that represents shapes and let us define an equality function on it.

```
open data Shape = Square Int

eq :: Shape → Shape → Bool
eq (Square s) shape = eqSquare shape s

eqSquare (Square s') s = s == s'
```

By dispatching to *eqSquare* we are able to complete the definition of *eq*. Extending this definition is also possible.

```
extend data Shape = Rect Int Int

eq (Rect l w) shape = eqRect shape l w

eqSquare (Rect l w) s = l == s && w == s

eqRect (Square s) l w = l == s && w == s
eqRect (Rect l' w') l w = l == l' && w == w'
```

These definitions, while legal in augmented Haskell, are not what a programmer would expect to write. It is also fairly obvious into which type class instances the equations will go during translation; the *eqRect* equations matching on *Square s* and *Rect l' w'* will be defined in separate instances because the component types containing the *Square* and *Rect* variants are different.

A more Haskell-like definition would start with:

```
open data Shape = Square Int
eq :: Shape → Shape → Bool
eq (Square s) (Square s') = s == s'
```

And it would be extended as follows:

```
extend data Shape = Rect Int Int

eq (Square s) (Rect l w) = l == s && w == s
```

$$eq\ (Rect\ l\ w)\ (Square\ s)\ =\ l\ ==\ s\ \&\&\ w\ ==\ s$$
$$eq\ (Rect\ l\ w)\ (Rect\ l'\ w')\ =\ l\ ==\ l'\ \&\&\ w\ ==\ w'$$

A more straightforward translation of from this into the target language in which two pattern matches appear inside instance methods is not possible—*Square* and *Rect* are part of different component types in the target language. However, providing that the compiler keeps track of suitable information, it is always possible to translate from such definitions to ones that are single pattern-match based. For instance, it is clear that the first new equation on *eq* would become a new equation for *eqSquare* while the next two would be for *eqRect*.

The crucial factor that makes such a pre-translation possible is that we have a *named handle* on the patterns. For instance, in the example above we associate the name *eq* with the two patterns on the left hand sides of the equations. Unfortunately there is no way in which the pattern match introduced by the case expression in the example below could be extended.

$$eq\ ::\ Shape\ \rightarrow\ Shape\ \rightarrow\ Bool$$
$$eq\ (Square\ s)\ shape\ =\ \textbf{case}\ shape\ \textbf{of}$$
$$\quad (Square\ s')\ \rightarrow\ s\ ==\ s'$$

A programmer would have to avoid definitions such as this if they wanted them to be extensible. I do not provide any further details of the pre-translation would be done but note that the approach generalises easily to n-ary functions.

5.8 Related work

Several papers ([107], [58], [72], [21]) have focused on extending object-oriented languages in order to make the addition of extra functionality easier. (Of these, only Zenger and Odersky's and Bruce's solutions can be statically type checked.) However, I wish to do the converse by making the addition of variants easier in a functional language. Fortunately, solutions in functional languages have also been studied. Two notable solutions have been proposed for OCaml [42] and the hybrid object-oriented/functional language, Scala [109]. Initially only informal solutions [59] were provided for Haskell. Unfortunately, these only work on the small examples the techniques were demonstrated upon. However, recently Löh and Hinze [66] proposed a complete solution.

In this section we compare the aforementioned solutions in OCaml, Scala and Haskell with our own. The first two solutions satisfy a property that mine does not – *independent extensibility*. This property was defined by Zenger and Odersky [109] as the fifth criterion that any solution to the expression problem should satisfy. Independent extensibility is the ability to combine independently developed extensions so that they can be used jointly. Although this property is desirable, providing a useful form of modularity for writing extensions, it is not necessary for implementing a front-end plug-in enabled compiler.

5.8.1 OCaml's solution to the expression problem

Polymorphic variants[42], a novel feature of the OCaml language, provide a solution to the expression problem. Unlike normal data type variants in functional languages, polymorphic variants can be shared between several data types. For example, the following function takes arguments of type `[< 'Var of string | 'Lam of string * 'a ]`

```
let f = function 'Var s -> s | 'Lam name body -> n
```

Types containing polymorphic variants come in three sorts: lower bounded, upper bounded and fully determined. A notion of *subtyping* is associated with such types; for example, a type containing more polymorphic variants than a lower bounded variant type is a subtype of it. The function above takes an upper bounded type; this is indicated by the `<` character after the left square bracket. This means that the function accepts arguments of any type which has *at most* the polymorphic variants `'Var of string` and `'Num of int`. For instance, it accepts the variant type `['Num of int]` (which incidentally is a fully determined type). Functions which match on polymorphic variants accept upper bounded types, while functions which a yield a polymorphic variant yield a type that is lower bounded. Polymorphic variants also support recursion. By way of example, the type of a map function of a variant type for lists is:

```
# let rec map f = function
      'Nil -> 'Nil;
    | 'Cons(x, xs) -> 'Cons(f x, map f xs);;
val map:
  ('a -> 'b) ->
  ([< 'Cons of 'a * 'c | 'Nil ] as 'c) ->
  ([> 'Cons of 'b * 'd | 'Nil ] as 'd) = <fun>
```

The type, `[< 'Cons of 'a * 'c | 'Nil ] as 'c`, is an equi-recursive type[7].

The nature of lower bounded variant types makes them ideal for extending existing data types with new variants. Figures 5.10, 5.11, and 5.12 fully implement the running example in OCaml. For a similar example see Garrigue's paper [42].

Function `alpha_f0` (Figure 5.10) implements the alpha functionality on the variable, abstraction and application variants of the lambda calculus. Note that it takes a *recursive processor* as an argument. This technique is known as *open recursion*. The fixed point of the function is defined only once all of the required functionality has been defined. In this module this is done by function `alpha_exp_0`. In general, this must be done in every extension module for all functions defined on the open data type so far. Note the close similarity of this with our requirement for capping classes. Capping classes close the recursion on type classes while defining functions such as `alpha_exp_0` closes the recursion on values and types.

Figures 5.11 defines extensions to the `alpha` function on let-expressions while 5.12 defines the `eval` function on ordinary lambda expressions and let-expressions. Note the

[7] An equi-recursive type, as opposed to an iso-recursive type, is one whose one-step unfolding is *equal* to itself. An iso-recursive type is merely *isomorphic* to its one-step unfolding.

```
type 'a exp_0 = [`Var of string | `Lam of string * 'a | `App of 'a * 'a ]

let swap (a,b: string * string) (o: string): string =
  if a = o then b else o

let alpha_f0 (alpha_rec: 'a -> string * string -> 'a)
             (expr: [< `Var of string
                    | `Lam of string * 'a
                    | `App of 'a * 'a])
             (s: string * string): 'a =
  match expr with
    | `Var v         -> `Var (swap s v)
    | `Lam (v, body) -> `Lam (swap s v,  alpha_rec body s)
    | `App (a, b)    -> `App (alpha_rec a s, alpha_rec b s)

(* Closes function [alpha] on the [exp_0] data type *)
let rec alpha_exp_0 (expr: 'a exp_0 as 'a) (s: string * string) =
  alpha_f0 alpha_exp_0 expr s
```

Figure 5.10: The *F0_Alpha* module implemented OCaml.

special syntax used in the matches: `#exp_0 as term`. This matches on the variants inside the type `exp_0`.

The recursion is closed at the value and type level—by equating `'a` with `'a exp_1`—simultaneously with the following declaration.

```
let rec eval_exp_1 expr env = eval_f1 eval_exp_1 expr env
```

It is important that this fixed point is only found at the point where one wishes to use the extensible data type. For instance, `eval_exp_1` cannot be further extended. In the case of using polymorphic variants inside a plug-in compiler one would have to ensure that that the fixed point was defined inside the plug-in and not in the source code of the compiler.

This solution satisfies the four base requirements of a solution to the expression problem proposed by Zenger and Odersky. In particular, separate compilation is satisfied because the function `alpha_f0` and the functions `eval_f0` and `eval_f1` can be written in separate modules and type checked independently. The solution also satisfies the extra requirement of *independent extensibility*. How it achieves this is discussed in the next section.

Comparison

One disadvantage of open abstract types is that they do not satisfy the *independent extensibility* criterion while polymorphic variants do. The key feature of polymorphic variants that makes this possible is their ability to define a *coalesced sum* of types. Type `'a let_lambda` is just such a coalesced sum of the type `'a lambda` and the following.

```
[> `Lam of string * 'a * 'a] as 'a
```

```
exception Lookup_Failure of string
exception Eval_Failure of string
open F0_Alpha

(* Extend [exp_0] with let expressions *)
type 'a exp_1 = [ `Var of string | `Lam of string * 'a | `App of 'a * 'a
                | `Let of string * 'a * 'a ]

let alpha_f1 alpha_rec expr s =
  match expr with
    | `Let (name, body, exp) ->
        `Let (swap s name, alpha_rec body s, alpha_rec exp s)

(* Closes function [alpha] on [exp_1] data type *)
let rec alpha_exp_1 expr s = alpha_f1 alpha_exp_1 expr s
```

Figure 5.11: New functionality defined on the *alpha* function in OCaml module *F1_Eval* .

It is not a strict sum in the usual sense because the constructors of both types are merged. The following ordinary data type declaration demonstrates this. In order to include lambda terms in type `'a let_lambda` it is necessary to wrap them in the `Lam` constructor.

```
type 'a lambda      = VarL of string | Abs of string * 'a | App of a' * 'a
type 'a let_lambda  = Lam of 'a lambda  | Let of string * 'a * 'a
```

Although I have not found a way to provide for independent extensibility using open abstract types a comparison with OCaml has provided some insight. In some sense we are encoding a form of subtyping using Haskell's unification. This subtyping relationship is present natively in the type system for polymorphic variants. Unfortunately, our form of encoded subtyping is rather primitive and only allows for linear extensibility. To illustrate this we refer to the example of the previous chapter.

The heads for the *Eval* class and *Exp* instance are:

$$\begin{aligned}&\textbf{class}(\ \mathit{Pretty}\ (\mathit{EvalD}\ \mathit{cxt})\ a\\&\qquad,\ \mathit{Sat}\ ((\mathit{EvalD}\ \mathit{cxt})\ a)\ \Rightarrow\ \mathit{Eval}\ \mathit{cxt}\ a\ \textbf{where}\\&\quad\ldots\end{aligned}$$

$$\begin{aligned}&\textbf{instance}\quad \mathit{Sat}\ (\mathit{EvalD}\ \mathit{cxt}\ (\mathit{Exp}\ (\mathit{EvalD}\ \mathit{cxt})))\\&\qquad\Rightarrow \mathit{Eval}\ \mathit{cxt}\ (\mathit{Exp}\ (\mathit{EvalD}\ \mathit{cxt}))\ \textbf{where}\\&\quad\ldots\end{aligned}$$

The type variable *cxt* is what allows for the extension. The capping class, *EvalEnd* has the following head (and associated *Exp* instance head):

$$\textbf{class}\ \mathit{Eval}\ \mathit{EvalEndD}\ a\ \Rightarrow\ \mathit{EvalEnd}\ a$$

```
type 'a env = (string * 'a) list

let rec lookup_env (env: 'a env) (name: string): 'a =
  match env with
      [] -> raise (Lookup_Failure ("Variable " ^ name ^ " not found"))
    | ((name', term)::rest) ->
       if name = name' then term else lookup_env rest name

let ext_env (env: 'a env) (x: string * 'a): 'a env = x :: env

let eval_f0 (eval_rec: 'a -> 'a env -> 'a)
            (expr: [< `Var of string
                   | `Lam of string * 'a
                   | `App of 'a * 'a])
            (env: 'a env) =
  match expr with
    | `Var name -> lookup_env env name
    | `Lam (name, body) as lam -> lam
    | `App (f, x) ->
        match eval_rec f env with
          | `Lam (name, body) ->
              eval_rec body (ext_env env (name, eval_rec x env))
          | _ -> raise (Eval_Failure "function expected")

let eval_f1 (eval_rec: 'a -> 'a env -> 'a)
            (expr: [< `Var of string
                   | `Lam of string * 'a
                   | `App of 'a * 'a
                   | `Let of string * 'a * 'a])
            (env:'a env) =
  match expr with
      #exp_0 as term -> eval_f0 eval_rec term env
    | `Let (x, exp, body) ->
        eval_f0 eval_rec (`App (`Lam (x,body), exp)) env


(* Closes the [eval] function on type [exp_1] *)


let rec eval_exp_1 expr env = eval_f1 eval_exp_1 expr env
```

Figure 5.12: Function *eval* defined on ordinary lambda expressions and the *let* extension in OCaml module *F1_Eval*.

instance *EvalEnd* (*Exp* (*EvalD EvalEndD*))

In practice this means that the type *Exp* (*EvalD EvalEndD*) is a subtype of the polymorphic type *Exp* (*EvalD a*). Unfortunately, this particular encoding of a subtyping relationship forces us to extend in a linear manner even if the functionality of the extension is orthogonal to the original. It is an open question as to whether an encoding of another subtype relationship is even possible.

I mentioned before that the fixed point of each open function needs to be declared for every function that has been defined on an open data type and that this is a close analogue of capping classes in Haskell. Recent work by Garrigue [43] has removed the aforementioned shortcoming from OCaml via the use of recursive modules and private row variables. Private row variables provide a mechanism to abstract on lower bounded variant types. Previously, the programmer had to rely on type inference for such types as only fully determined variant types could be given type synonyms. Combining this feature with recursive modules, and the ability to find the fixed-points of functors, the programmer is able to close the recursion at the module level. The reader is referred to the paper for more details.

OCaml has support for dynamic loading of byte-code (and even native code on some platforms). It would also be suitable as the implementation language for a plug-in compiler

5.8.2 Scala's solution to the expression problem

Scala [73], like OCaml, is a hybrid of object oriented and functional language features. However, while OCaml adds a relatively lightweight object system to the functional language ML, Scala does the converse by adding functional language features to a Java-like object oriented language.

As with OCaml's polymorphic variants it is the notion of subtyping that makes a solution to the expression problem possible. Notably, independent extensibility is also possible in Scala using mixins.

Scala's solution using subtyping and mixins

The solution is demonstrated by programming our running example in Scala. First, we start with the definition of alpha renaming functionality.

```
trait Alpha {
  type exp <: Exp;
  trait Exp {
      def alpha(s:Pair[String,String]): exp;
  }
  class Var(name: String) extends Exp {
      def alpha(s:Pair[String,String]) =
          new Var(swap (s,name)).asInstanceOf[exp];
  }
  class Lam(name: String, body: exp) extends Exp {
```

```
        def alpha(s:Pair[String,String]) =
            new Lam(swap(s,name), body.alpha(s)).asInstanceOf[exp];
    }
    class App(a: exp, b: exp) extends Exp {
        def alpha(s:Pair[String,String]) =
            new App(a.alpha(s), b.alpha(s)).asInstanceOf[exp];
    }
}
```

The pertinent aspects of this declaration are as follows. First, a *trait* is like a class (in OO languages) but has neither constructors nor state. Second, the declaration `type exp <: Exp` declares that `exp` is constrained to be a *subtype* of the nested trait `Exp` declared just below it. In the nomenclature of Scala `exp` is known as an *abstract type*.

Extending the functionality of the `Var`, `Lam` and `App` classes is done using *mixin composition*. Mixins are a mechanism by which the member declarations of one class can be included into another class. Mixin declarations have the form:

class `extends` *class* `with` *class*

In the following code mixins are used to extend each of the variants with the `eval` method.

```
trait Eval extends Alpha {
  type exp <: Exp;
  type Env = List[Pair[String, exp]];

  def lookup(name: String,env: Env) : exp = env match {
    case List() => throw new Error("Could not find variable " + name);
    case pair :: rest => pair match {
      case Pair(name1, term) =>
        if (name1 == name) {term} else {lookup(name, rest)};
    }
  }

  def extEnv(env: Env, pair: Pair[String, exp]) : Env = {
    return pair :: env;
  }

  trait Exp extends super.Exp {
    def eval(env: Env) : exp;
  }

  case class Var(name: String) extends super.Var(name) with Exp {
    def eval(env: Env) = lookup(name, env)  ;
  }

  case class Lam(v: String, body: exp) extends super.Lam(v, body) with Exp {
    def eval(env: Env) = this.asInstanceOf[exp];
  }
```

```
    case class App(f: exp, x: exp) extends super.App(f, x) with Exp {
      def eval(env: Env) = {
        f.eval(env).asInstanceOf[Exp] match {
          case Lam(name, body) => {
            def value = body.eval(extEnv(env,
                                         new Pair(name, x.eval(env))));
            value.asInstanceOf[exp];
          }
          case _               => throw new Error("Function expected");

        }
      }
    }
  }
```

Another thing to note is that although the first line of the `Eval` trait appears to be the same as the one in the `Alpha` trait, it actually declares `exp` to be a subtype of the `Exp` declared in the `Eval` trait. In Scala, new declarations merely *shadow* existing declarations of that name. The old trait is still able to be referenced, in this example as `super.Exp`.

Finally, the manner in which a `Let` variant can be added is demonstrated.

```
object ExpLet {
  trait EvalLet extends Eval {
    case class Let(name: String, body: exp, exp: exp) {
      def alpha(s:Pair[String,String]) =
          new Let(swap(s,name), body.alpha(s), exp.alpha(s));
      def eval(env: Env) = App(Lam(name, exp).asInstanceOf[exp], body).eval(env);
    }
  }
}
```

There is one important final point. To create, and use the values requires that the abstract type `exp` be made concrete. The first line of the body of the object: `type exp = Exp` in the following declaration identifies the abstract type with the concrete `Exp` class that was declared in the `Eval` trait.

```
object ExpTest extends EvalLet with Application {
    type exp = Exp;
    val test0 = Lam("x",Var("x"));
}
```

Zenger and Odersky [109] also provide a more functional solution to the expression problem using the Visitor pattern. Whether a programmer uses the approach above or one based on the Visitor pattern depends on whether they feel adding variants or functionality

is more common; in each approach extension in one direction is easy but hard in the other. We elide the details here as they do not add to the merit of the comparison.

Comparison

The Scala language is founded upon a recently developed calculus and dependent type system: νObj. νObj [74] is a nominal theory and describes objects and classes which can have types and nested classes as members. The system is highly expressive and can support via encodings most concepts of Standard ML-style module systems, System $F_{<:}$ and the virtual types and family polymorphism of Ernst's GBETA [38] language. Like $F_{<:}$ subtyping is a key feature of νObj.

In fact, in both OCaml and Scala it is this feature of the type system that provides the functionality for a solution to the expression problem. While subtyping can be used in OCaml without explicit declaration, to solve the expression problem in Scala it is necessary to explicitly declare a subtyping relation using the notion of abstract types.

Abstract types in Scala are similar to our notion of open abstract types. In Scala, the constraint on an abstract type guarantees that the object contains certain methods. Similarly, a value of an open abstract type is also guaranteed to have certain methods defined upon it. Indeed, just like an object in Scala the methods are *contained* within it because dictionaries are packed inside existentially quantified values. In fact, Läufer [63] notes that this is similar to the dynamic dispatch mechanism of many object-oriented languages.

Since Scala is interoperable with Java, its compiler generates Java byte-code. The JVM is quite capable of loading code dynamically so in principle Scala should be another fine candidate for implementing a plug-in enabled compiler.

5.8.3 Other solutions in Haskell

To date, the only published (Haskell) solution to the expression problem is Löh and Hinze [66]. Their syntax is very similar to the syntactic sugar proposed in this chapter. The basic idea is to separate out the open declarations from the closed declarations and place all of them in a single module (*Main*) via a source to source translation. A naive implementation requires recompilation of the entire program each time an open declaration was extended. This is because the translation is applied to the entire program.

However, the authors describe a method whereby the amount of recompilation can be kept to a minimum. Each module M is translated to a module M' which contains all the original declarations except the declarations of open entities. (In particular, *Main* is translated to *Main'* and if M imports N then M' imports N'. The resulting program consists of an additional module, a new *Main* module which contains all the collected open entities of the program.

Unfortunately, this often results in a mutual dependency with each module that contained open declarations. Each module M that defines an open entity results in a module M' that imports *Main* which in turn depends on many modules of the program. Although

the modules can be compiled separately they cannot be re-compiled independently; a change to an open entity necessitates recompilation of all modules depending on *Main*.

To prevent this, the left and right hand sides of all open equations are separated into two new equations. The first does the pattern matching and dispatches to the second which is moved back to the module it was originally declared in. As long as the interface between this module and the *Main* module remains stable a change to open entities only results in a re-compilation of the *Main* module. Unfortunately, this fact disqualifies the solution from achieving true separate compilation. It is unclear how well Löh and Hinze's solution works in a plug-in environment. It may be possible to use Stewart and Chakravarty's [88] method to re-load the entire application but this seems much more complicated that our solution and would require loading the entire program not just the plug-in module.

Also the aforementioned solution to separate compilation relies on an unimplemented feature of Haskell's module system – the ability to specify constructors as single entities within an interface. There are no theoretical difficulties in lifting this restriction, but nevertheless remains an impediment to an implementation.

A number of informal type-classed based (e.g. [59]) have also been proposed for Haskell. However, there is a crucial difference with our solution. Where these solutions lift constructor values to the type level, ours does not. This means that functions can still be written in a natural way using the full power of Haskell's pattern matching. Also, there is still a clear relation between a constructor and the data type it creates; a constructor creates values of its component type.

Another notable solution to the expression problem is provided by Kiselyov and Lämmel [56]. This requires that programs be written in an object oriented style. In our solution, functions on open data types are merely overloaded functions and the construction of values by smart constructors is almost as natural as with regular constructors. It is also a very heavy weight solution relying on the *HList* library which utilises Haskell's type system beyond what it was designed for.

5.9 Summary

In this chapter I have introduced a method for encoding extensible data types in Haskell using type classes. It relies upon several non-standard features of Haskell: multi-parameter type classes, overlapping instances and recursive dictionaries. I have also shown that it can encode binary (and, in general, n-ary) functions on such data types. The encoding is undeniably heavy but certainly no heavier than the solution of Kiselyov and Lämmel (which is the only other currently usable solution).

We have also surveyed two solutions in other languages to the expression problem which both rely on the notion of subtyping. This has been observed before by Bruce [21] who notes that System $F_{<:}$ (also known as F-bounded polymorphism) adequately solves many of the typing problems thrown up by the expression problem. In fact, Scala is based upon a calculus which is even more expressive, νObj [74], and can encode System $F_{<:}$.

This leads us inexorably to the conclusion that our approach to implementing exten-

sible data types relies on an *encoding* of a subtyping relationship. Unfortunately, this encoding is rather primitive, relying as it does on the unification of types. The fact that it is inferior to the native subtyping notions in OCaml and, particularly, Scala should not surprise us as both have sophisticated type systems that were designed to allow both functional and an object oriented style of programming.

Haskell's type classes are based upon a theory of qualified types [55]. The theory is quite general and can be used to model more than just type classes. Interestingly, it can be used as a foundation for subtyping. However, this requires a rule for subsumption be added to the type checker. Nevertheless, this opens up the possibility that a modest extension to Haskell's type system might allow a cleaner solution to the expression problem and in particular allow for independent extensibility.

Chapter 6

Front-end plug-ins

Now that an encoding of extensible data types has been developed we are ready to put them to use. Front-end plug-ins can be used to extend the syntax, semantics and safety checks of a language. In this chapter the ease with which a new form of syntactic sugar[1] and additional safety checks can be added to an existing language via a front-end plug-in is demonstrated. In particular, list comprehensions are implemented.

At first, it may seem like this is merely a syntactic extension. However, the type checking rules *are* extended by the plug-in. This presents the opportunity for a custom error message for list comprehensions to be displayed to the user rather than a less intuitive one thrown up by type checking the equivalent desugared program.

There is also the question of what to do when a more sophisticated extension is required. e.g. Generalised Algebraic Data Types [79]. When the new type system is more general than the old one there is no choice but to replace the old one. In such a setting plug-ins are of limited value. However, the more general a type system becomes the greater the chance in future that one can make many useful extensions to it via plug-ins.

The chapter is organised as follows. First, the basic structure of a front-end plug-in is described, followed by a high-level description of how the list comprehension plug-in is implemented. Code is provided but it has been written using the extensible data type syntax introduced in Section 5.2. A real implementation exists which contains a hand translation of the code presented in this chapter. The compiler is called PHRaC and a section is devoted to describing issues encountered such as how well the extensible data type encoding works in practice. The chapter concludes with a discussion of related and future work.

6.1 List comprehensions

List comprehensions [80] are a useful syntactic feature of the Haskell language[2]. They provide a convenient syntax for defining lists via the use of guards (or predicates) and

[1] This is not to be confused with the syntactic sugar introduced in Section 5.2. This is syntactic sugar for the language being compiled.

[2] They are also found in Python and Erlang.

$$
\begin{array}{l}
[\, e \mid] \;=\; [\, e \,] \\
[\, e \mid b,\; Q \,] \;=\; \textbf{if}\ b\ \textbf{then}\ [\, e \mid Q \,]\ \textbf{else}\ [] \\
[\, e \mid p \;\leftarrow\; xs\,,\; Q \,] \;=\; \textbf{let} \\
\qquad\qquad\qquad\qquad\qquad\qquad ok\ p = [\, e \mid Q \,] \\
\qquad\qquad\qquad\qquad\qquad\qquad ok\ \_ = [] \\
\qquad\qquad\qquad\qquad\quad \textbf{in} \\
\qquad\qquad\qquad\qquad\quad concatMap\ ok\ xs
\end{array}
$$

Figure 6.1: Rules for translation list comprehensions into more primitive expressions.

generators (i.e. functions that generate lists of values). They are of the following form: $[\, exp \mid qual_1,\ \ldots,\ qual_n \,]$ where $qual_i$ is short for a *qualifier*. A qualifier is either a *generator* or a *guard*. Generators have the form $p \leftarrow e$ where p is a pattern and e is an expression. Guards are arbitrary expressions of type *Bool*.

A simple demonstration of their power (in Haskell) is:

$$
\begin{array}{l}
>\ [\, (x, y) \mid x \;\leftarrow\; [1..4],\ odd\ x,\ y \;\leftarrow\; [1..4],\ even\ y \,] \\
[\, (1,2),\ (1,4),\ (3,2),\ (3,4) \,]
\end{array}
$$

The generators $x \leftarrow [1..4]$ and $y \leftarrow [1..4]$ provide the lists of numbers from which the values (x, y) are drawn from. The guards *odd* x and *even* y filter out values. Only values for which the guards evaluate to *True* are kept. This example also implicitly demonstrates the scoping rules of list comprehensions. Each generator introduces a number of bound variables, those of the pattern, which are in scope in all subsequent qualifiers and on the left hand side of the vertical bar: |. Any variables occurring on the left hand side must either be already in scope or appear on the right hand side of the list comprehension.

List comprehensions are a classic example of syntactic sugar; they can readily be translated to simpler expressions in the language. The list comprehensions implemented in this chapter are the same as those defined in the Haskell 98 Report [80] except that we omit *local bindings* (i.e. the ability to define local functions inside list comprehensions). Fig 6.1 presents the translation rules.

As an example, the expression $[\, x \mid x \;\leftarrow\; [1..4],\ even\ x \,]$ would be translated to:

$$
\begin{array}{l}
\textbf{let} \\
\quad ok\ x = \textbf{if}\ even\ x\ \textbf{then}\ [x]\ \textbf{else}\ [] \\
\quad ok\ \_ = [] \\
\textbf{in} \\
concatMap\ ok\ [1..4]
\end{array}
$$

6.2 Adding front-end plug-in support for a compiler

A sensible design for a front-end plug-in is for it to closely follow the structure of the front-end of a compiler. Figure 6.2 shows the structure, which is organised into *phases*, of a typical compiler for a functional language.

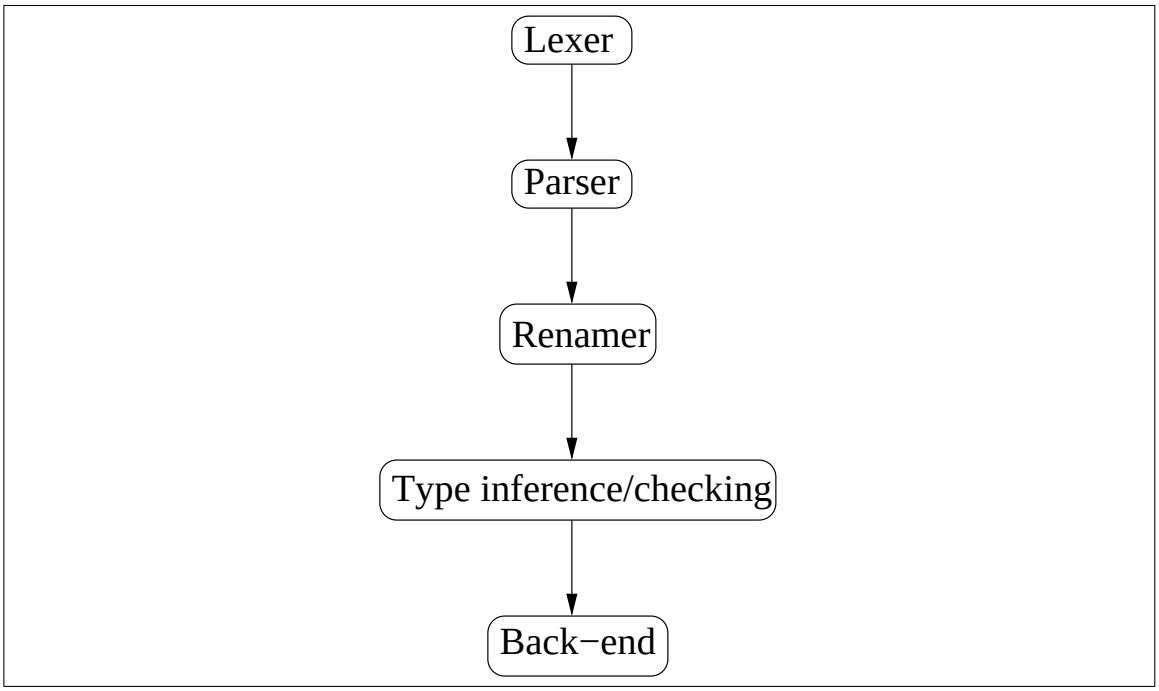

Figure 6.2: The structure of a compiler for a typical functional language

The lexer and parser are responsible for reading in the plain text of source files and translating them into abstract syntax. Collectively, these phases can detect purely syntactic errors. The *renamer* traverses over the abstract syntax tree assigning a unique identifier to all binding occurrences (or *definitions*) of variables and assigning the correct unique identifier to all *uses* of those variables. In the process it checks *scope* according to the rules of the language and signals an error whenever a variable is free (i.e. not in scope). The renaming function translates from an abstract syntax data type in which names are not unique to another which contains unique identifiers. Next, the *type inference/checking* phase checks the types of declarations that have programmer declared signatures and infers the type of those that do not. The greatest proportion of detectable errors are caught in this stage. The type checked code is then passed onto the back-end of the compiler.

The back-end of the compiler has been left intentionally abstract in Figure 6.2 because it has no bearing on the design of a front-end plug-in.

6.2.1 The anatomy of a front-end plug-in

A front-end plug-in does not simply affect a single phase; it necessarily affects many. We must do the following things:

- perhaps extend the lexer with new tokens. Often this is not necessary.
- extend the grammar, and hence the parser, with the new syntax.
- extend the data type used to represent abstract syntax with new variants.
- extend the renaming function to handle the new abstract syntax variants.

- extend the type checking code. Portions of the new syntax may have special type constraints upon them; this part of the plug-in checks those.
- add a desugaring function that translates the new syntax into the intermediate language. This is the least straightforward of all the parts of the plug-in and should be carefully scrutinised—or better yet, proved—for correctness by the plug-in writer.

The reader may wonder why desugaring comes so late in the pipeline and not before the renaming and type checking phases. This way there would be no need to extend the functionality of these two phases via plug-ins. However, better error messages can be generated by deferring desugaring until later.

It is widely recognised that error messages heavily affect the usability of a language and improving their understandability is an active area of research [46, 45]. One way in which a compiler can improve the readability of its errors messages is to provide program fragments with its error output. However, this means that abstract syntax must be available when checking errors. This allows for error messages tailored to a particular sort of syntax. Providing this functionality through the agency of a plug-in captures the essence of *domain specific error messages* as the plug-in writer has provided a custom message for their new syntax.

6.2.2 Extensible abstract syntax

In order to extend the abstract syntax data type it is necessary to declare it as extensible using the syntactic sugar defined in Section 5.2. To implement list comprehensions it is only necessary that expressions be extensible.

```
open data Exp = VarE Name
              | AppE Exp Exp
              | LamE Name Exp
              | ...
```

6.2.3 The front-end plug-in data type

A front-end plug-in is composed of several functions which are used instead of the default scanning, parsing, renaming, type inference and desugaring functions. In PHRAC the data structure representing a front-end plug-in is as follows:

```
data FrontEndPlugin token =
  FrontEndPlugin{plugScan :: String → [token]
                 , plugParse :: [token] → PS.Program
                 , plugRename :: PS.Program → SymTabM AS.Program
                 , plugTypeInfer :: Gamma → AS.Program → SymTabM AS.Program
                 , plugDesugar :: AS.Program → SymTabM IntermediateRep
                 }
```

We parameterise on the type of tokens to increase the flexibility of front-end plug-ins. The *PS.Program* is a data type representing the abstract syntax of the program but that doesn't use unique identifiers for names; *AS.Program* uses unique identifiers. *Gamma* refers to an environment (or type assignment) for type inference while *SymTabM* is a symbol table monad containing support for name generation and look-up of names. *IntermediateRep* is the data type of the intermediate representation for the compiler.

6.3 The list comprehension plug-in

We are now ready to take a detailed look at how a list comprehension plug-in is implemented. The presentation of the plug-in has been simplified by using the syntax introduced in Section 5.2. The actual implementation appears in Appendix D. It should be stressed that plug-ins are developed entirely independently of the compiler; it is only necessary to have the plug-in API available.

A value of type *PS.Program* is produced by the parser, renamed and the resulting value, of type *AS.Program*, then has type inference and desugaring applied to it.

6.3.1 New variants

List comprehensions are represented by extending the *Exp* data type and introducing a new data structure for qualifiers.

$$\textbf{extend data}\ \mathit{Exp} = \mathit{ListCompE}\ \mathit{Exp}\ [\mathit{Qual}]$$

$$\begin{aligned}\textbf{data}\ \mathit{Qual} \quad &= \mathit{GenQual}\ \mathit{Pat}\ \mathit{Exp} \\ &\mid \mathit{GuardQual}\ \mathit{Exp}\end{aligned}$$

The constructor, *ListCompE*, bundles together an expression with a list of *Qual*s, which represent either generators or guards. The equations which provide the functionality for this new variant of the *Exp* data type will appear in subsequent sections.

A note should be made here about function names. The phases of the compiler each have a top-level driver function. These have names *parseProgram*, *renameProgram*, etc. Since abstract syntax is represented using a large, mutually recursive data structure it is necessary, within a phase, to declare a function for each data type that makes up the representation. For example, within the renaming phase there are functions such as *renameBind*, *renameExp*, *renamePat*, etc. We follow the convention that each one of these functions is prefixed with the phase name.

6.3.2 Lexing and parsing

For the purposes of this case study we are not concerned with extending the parser. This is because, more often than not, parsers are generated using tools. Work has been done on extensible parser specification [68, 19] and where available such tools should be used in plug-in compilers. (PHRaC's parser is generated using the *happy*[4] parser

generator, which unfortunately is inextensible. For the real implementation I just replaced the parser.)

Another option would be to use *parser combinators* [53, 89, 65] to write the parser. It should be relatively easy to design them in such a way that they could support grammar extension. In fact, a similar problem has already been solved by Stewart and Chakravarty [88]; they have designed a library of lexer combinators for specifying keystroke interfaces for the Yi text editor. Using these combinators it is possible to extend and override existing keystroke mappings.

6.3.3 Renaming

The renaming is straightforward. We simply need to call *renameExp* and *renamePat* recursively on the components of *ListCompE* variant.

```
renameExp (ListCompE exp quals) = do
                                    enterBlock
                                    quals' ← mapM renameQual quals
                                    exp' ← renameExp exp
                                    exitBlock
                                    return (AS.ListCompE exp' quals')
renameQual (GenQual pat exp) = do
                                 exp' ← renameExp exp
                                 pat' ← renamePat pat
                                 return (AS.GenQual pat' exp')
renameQual (GuardQual exp) = do
                               exp' ← renameExp exp
                               return (As.GuardQual exp')
```

6.3.4 Type Inference

Let us assume (for now) that our language uses Hindley-Milner type inference. We wish to extend the algorithm to handle list comprehensions. It must:

- check that generators are of list type, be it a polymorphic type or a specific one.
- check that guards are of type *Bool*
- check that the expression on the left hand side of list comprehension is of the same type as the elements of the list types of generators

```
tiExpr :: Gamma → Exp → TI (Subst, Type)
tiExpr gamma (ListCompE exp quals) =
```

```
    do
      gammaQuals ← mapM (tiQual gamma) quals
      alpha ← freshTypeVar
      (sub, tau) ← tiExpr (gamma ++ gammaQuals) exp
      unifier ← unify tau alpha
      return (unifier `composeSubst` sub, TyApp ListTyCon alpha)

tiQual :: Gamma → Qual → TI Gamma
tiQual gamma (GuardQual exp) =
    do
      (sub, tau) ← tiExpr gamma exp
      if appSubst sub tau `eqType` BoolType then
        return emptyGamma
      else
        fail ⟨custom error message⟩
tiQual gamma (genQual pat exp) =
    do
      (sub, tau) ← tiExpr gamma exp
      alpha ← freshTypeVar
      unifier ← unify (TyApp ListTyCon alpha) tau
      return (appSubst unifier alpha)
```

6.3.5 Desugaring

The desugaring phase transforms list comprehensions into calls to *ok* and *concatMap* as per the rules in Figure 6.1.

```
desugarExpr (ListCompE exp []) = do
  exp' ← desugarExpr exp   -- parser never produces this case
  return (Con consDataId [exp', Con nilDataId []])
desugarExpr (ListCompE exp (GenQual pat bindExp : quals)) = do
  bindExp' ← desugarExpr bindExp
  okId ← freshValVar
  x ← freshValVar
  okBody ← do
    alt1Body ← desugarExpr (ListCompE exp quals)
    return (Case (Var x)
      [Alt{ alt_pat = pat
          , alt_exp = alt1Body}
          , Alt{ alt_pat = PWildcard
          , alt_exp = Con nilDataId []}
      ])
  return $
    Letrec
```

```
        [ValBind{ vbind_name = okId
                , vbind_type = Nothing
                , vbind_pats = [PVar x]
                , vbind_exp = okBody
                , vbind_info = emptyInfo}]
        (Var concatMapId `App` Var okId `App` bindExp')
desugarExpr (ListCompE exp (GuardQual bexp : quals)) = do
  bexp' ← desugarExpr bexp
  rest ← desugarExpr (ListCompE exp quals)
  return $ (If bexp' rest (Con nilDataId []))
```

6.3.6 The *FrontEndPlugin*

Perhaps surprisingly, the value that defines the plug-in does not differ much from the default. The only fields of the record that have non-standard values are those for lexing (i.e.scanning) and parsing. Let's call these newly defined values *listCompScan* and *listCompParse*.

```
frontEndPlugin =
  FrontEndPlugin{plugScan      = listCompParse
                , plugParse     = listCompScan
                , plugRename    = renameProgram
                , plugTypeInfer = tiProgram
                , plusDesugar   = desugarProgram
```

The other fields simply do not require replacement in this case. We have extend the functionality of these phases simply by extending the *Exp* data type and defining new equations on the new *ListCompE* variant.

6.4 A real implementation: PHRaC

In this section I describe the issues encountered while implementing front-end plug-in support for PHRaC a compiler for a small, Haskell-like language. Extensible data types were implemented by hand using the translation from Chapter 5.

6.4.1 The PHRaC API

Plug-in writers must compile against the PHRaC API. Due to PHRaC's prototypical nature its API actually exposes everything. However, during the implementation use was only made of the following aspects:

- *Abstract Syntax* – all types representing abstract syntax including smart constructors.

- *Symbol table* – functions for creating fresh variables on the type and value level, look-up functions.
- *Renaming* – functions for entering and leaving scopes. e.g. *enterBlock*, *exitBlock*.
- *Type inference* – functions for unification, substitution, extension of environments, etc.

Since PHRaC is compiled with GHC, and seeing that GHC has a package system, I chose to compile PHRaC *as a package*—package `phrac-1.0`. Plug-in writers then simply need to include this package on the command line using a command line flag when compiling.

6.4.2 Extensible abstract syntax in PHRaC

Although we focus on adding a new form of expression to a Haskell-like language in this chapter, one could easily imagine adding new forms of declarations. Unfortunately, retrofitting PHRaC for each new language entity comes with a reasonably heavy syntactic cost to the implementation of the compiler. It must be stressed that this syntactic overhead only exists because extensible abstract syntax was encoded by hand. An implementation of the translation presented in Section 5.6 would remove the overhead completely.

To see just how much overhead the encoding introduces consider the type of the expression desugarer. (The *DesugarCxt* is a type synonym for a hierarchy of functionality including overloading resolution, type inference and desugaring.)

```
data DesugarD cxt a =
  DesugarD{ desugarExprD :: a → ST' cxt ExpInterp
          , desugarExt :: cxt a }
type DesugarCxt cxt = OverD (InferD (DesugarD cxt))
class (Sat (DesugarCxt cxt a)
  , Inference (DesugarD cxt) a
  ) ⇒ Desugar cxt a where
  desugarExpr :: a → ST' cxt ExpInterp
```

Even worse, the syntactic cost is cross-cutting, asserting itself in a systematic way across many modules and upon many functions. *Sat* predicates are a necessity of our encoding of EDTs. They allow for a newly defined class to become, in effect, the superclass of an existing one; this also allows for cyclic dependencies among classes. While this allows for the functionality on an EDT to be extended without the necessity of modifying existing code, it has an annoying side effect. This results from the fact that any expression that refers to overloaded identifiers (e.g. *renameExpr*) requires *Sat* instances to be available or a context to be supplied. Thus, when an instance has not been provided a context must be added to every function which directly or indirectly refers to an overloaded identifier. For instance, consider the top-level driver for the desugaring phase:

```
desugarProgram :: Sat (DesugarCxt cxt (EXP (DesugarCxt cxt)))
                  ⇒ Program (EXP (DesugarCxt cxt))
                  → ST' cxt (Program ExpInterp)
desugarProgram prog = do
  instances ← mapM desugarClassInst (prog_instances prog)
  bindings  ← mapM desugarValBind   (prog_bindings prog)
  return $ prog{ prog_instances = instances
               , prog_bindings = bindings
               }
```

There is no way around this problem. The only case in which we *can* provide *Sat* instances is after a *capping class* (see Section 5.6.4) has been written. For a phase, such as renaming, which is later extended in the module declaring type inference functions, this is simply not possible.

As a result only a single member of the mutually recursive data types that represent abstract syntax was made extensible: expressions. The syntactic burden would have been too great otherwise. Also, in making expressions extensible it was important the original expression data type was not lost because the back-end of the compiler accepts this data structure as input. (This is because PHRAC is actually an interpreter.) It was declared exactly the same way but was not made extensible.

Loading plug-ins

The user specifies that they wish to use a plug-in on the command line with the command line flag `--front-end-plugin=<name>`[3]. Plug-ins themselves are compiled as GHC *packages*. PHRAC expects this package to expose a module `Phrac.FrontEndPlugins` containing the function `phracPlugins`. This design was motivated purely by simplicity. A package is essentially a library plus accompanying interface files, with information on the paths to these files included. Since the `hs-plugins` library uses GHC's package management system in a fundamental way, the choice to compile plug-ins as packages immensely simplified the code I wrote to load plug-ins. It also increases the ease with which future plug-in writers can make them known to PHRAC. They just use the package management tool: *ghc-pkg*.

The front-end plug-in data structure for PHRAC is almost the same as the one defined earlier in the chapter but is complicated by our EDT encoding. It is defined as follows:

```
data PhracPlugins token cxt ascxt =
  PhracPlugins
    { plugScan      :: String → [token]
    , plugParse     :: [token] → PSProgram cxt ascxt
    , plugTransform :: PSProgram cxt ascxt → IO (ASProgram ascxt)
```

[3] Another flag `--package-conf=<conf>` allows the user to provide a path to alternative package configuration file.

$$
\begin{aligned}
&, plugTiMain && :: Assumps \\
& && \rightarrow ASProgram\ ascxt \\
& && \rightarrow ST\ (ASEXP\ ascxt) \\
& && \qquad (Assumps, ASProgram\ ascxt) \\
&, plugDesugar && :: ASProgram\ ascxt \\
& && \rightarrow ST\ (ASEXP\ ascxt)\ (AS.Program\ AS.ExpInterp) \\
&\}
\end{aligned}
$$

As it stands, this data type is unsuitable for loading dynamically because it is polymorphic. To remedy this, either we can make it an instance of class *Typeable* and use `hs-plugins`' *dynload* function, or wrap the type in an existential data type. I have opted for the latter approach as writing a *Typeable* instance, although not too hard in itself, requires the plug-in writer to implement *Typeable* instances for whatever the type variables *token*, *cxt* and *ascxt* are instantiated with.

6.5 Related work

We begin by examining the Magik [36] plug-in compiler which we previously mentioned in section 4.8. Magik, apart from allowing domain specific optimisations to be written, can also be used to add safety checks to a language. However, this is done by analysing the intermediate representation and adding code to it. My approach differs in that additional safety checks are added directly after parsing in the phases in which the abstract syntax tree has a direct correspondence with the source code. This allows better error messages to be generated. We are by no means limited by this; using the techniques of Chapter 4, we can add Magik-like safety checks.

At first glance it also seems as if Magik has the ability to extend the compiler's internal data structures. This capability is crucial in PHRaC allowing us to easily extend renaming, type checking and desugaring code to provide new semantics and safety checks. However closer examination reveals that Magik only allows one to modify the data structures declared in the source code being transformed. The plug-in enabled GHC, as introduced in Chapter 4, also has the same capability.

The SUIF2 Compiler Infrastructure [13] provides a high level specification language for describing abstract syntax. The paper claims that new nodes can be defined but that modules that operate on the old abstract syntax will not be affected by this. Unfortunately, too little information is provided in the paper to assess with conviction whether the Hoof language solves the expression problem.

The work of Dijkstra [29] is relevant in two ways. First, he presents an extensible Attribute Grammar system called AG. It is possible to introduce new variants and rules for inherited and synthesised attributes in new files. AG then collects these variants and rules, does basic consistency checks and generates a full attribute grammar. Second, a system called Ruler is presented for incrementally developing a type inference and checking system. However, the focus is more on maintaining an incremental description than in

making facilitating extension a type checker. Differences between versions of the type checker are more like diffs than plug-ins. Still, the possibility exists that the ideas could be adapted to a plug-in setting.

The remaining systems we consider in this section all extend expressiveness (and sometimes safety) via purely syntactic means. The as yet unimplemented Fortress language [14] provides a language feature known as *syntax expanders* that allow a mapping between user-defined syntax and existing terms in the language to be defined. This expansion occurs before the traditional parsing phase.

Metaborg [20] is a language independent system which allows syntax—defined using the Syntax Definition Formalism (SDF) [47]—to be translated to concrete syntax in a particular language using a term rewriting system called Stratego/XT. The system is general, any language specific knowledge must be encoded into the term rewriting rules. The *metafront* [19] system provides a similar capability. One useful aspect of it is that it is designed to support the gradual extension of grammars in a modular fashion. This is done through the use of a novel parsing algorithm in which ambiguities are resolved through a notion of specificity.

The drawback of these syntactic techniques is that they lack knowledge of the semantics of the language being extended. However, PHRAC's front-end plug-ins allow the programmer to *extend* the compiler's knowledge of structural properties, such as scope and type. These need to be encoded from scratch when using any of the purely syntactic methods above.

6.6 Future work

There are a number of areas in which this work could be improved.

6.6.1 A plug-in DSL for type inference

There is a marked difference between the presentation of type systems in the literature and their implementation in programming languages. Almost uniformly type systems are presented in a relational or judgement style. Sometimes, but not always, this may be accompanied by a type checking algorithm written in pseudocode. The advantage of the relational style of presentation is that they are more amenable to proofs of preservation and progress. They are also more concise.

An intriguing paper [93] by Tomb and Flanagan presents a method in Prolog to take a type system specified using relations and generate, using partial evaluation, a type inference algorithm. Using the techniques from this paper it may be possible to develop a type inference DSL that allows a programmer to write the type checker in a relational style. This should make it easier to write type system extensions and possible even make the type checker amenable to automated proof.

6.6.2 Desugaring through lengthening plug-ins

In retrofitting PHRAC with front-end plug-in capability I added a new phase: desugaring. It is a fundamental phase of other compilers, such as GHC, which justifies its addition to some extent. However, there is a more generic manner in which new phases can be added: through the use of *lengthening plug-ins*. These plug-in points should be placed between each of the phases. Plug-ins written for these points take the input of the last phase and output data suitable for the next phase. In this way arbitrary code can be run between phases. Thus, desugaring could be implemented as a lengthening plug-in between type inference and the back-end.

6.6.3 A client/server compiler

There is a performance hit associated both with loading plug-ins and making PHRAC dynamic from the ground up. Each time a program is compiled, PHRAC is dynamically loaded along with any plug-ins specified on the command line. There is no way of surpassing this necessity. Unfortunately, when multiple files are compiled this performance hit is taken multiple times. Operating systems often provide support for precisely this problem by ensuring that program code is not immediately removed from memory upon termination. When a program is reloaded and makes use of system calls for dynamic linking the program code is still available and does not need to be reloaded. Unfortunately, PHRAC uses GHC's dynamic linker which does not provide this functionality.

Another solution would be to design PHRAC as a client/server application so that the performance hit would only be taken once. The idea would be load PHRAC as a resident program and invoke it by sending messages to it. Multiple plug-ins could even be resident even though only one of them might be used.

6.7 Summary

This chapter has presented a design for front-end plug-ins and shown its effectiveness via a case study. To the author's knowledge, PHRAC is the first compiler that has taken the approach of defining a front-end plug-in as a collection of extensions on multiple phases. Using front-end plug-ins one can extend the syntax, semantics and safety checks of the compiler. As opposed to purely syntactic methods, this approach allows the plug-in writer to reuse much of the language knowledge encoded in the various phases of the compiler.

To summarise, PHRAC is designed as follows:

- A front-end plug-in is made up of components, stored in a record data structure, that may extend each phase of the front-end. Functionality can also be extended simply by introducing new data types and writing instances for renaming, dictionary resolution, type inference and desugaring.
- At present the API exposes everything that each module in PHRAC does. PHRAC is compiled as a GHC package so that plug-in writers can easily compile against it.

- PHRAC is dynamic from the ground-up. It consists of a small boot-loader that dynamically loads the rest of the compiler. The compiler can in turn load plug-ins.
- Plug-ins are compiled as GHC packages which must, at the very least, expose a module called *Phrac.FrontEndPlugins* containing a function, *phracPlugins* of type *WrappedPlugins*.
- Plug-ins are loaded on the command line by providing the name of the package; a new flag has been introduced.

Chapter 7

Conclusion

PROGRAMMING languages are the medium of expression for writers of software. Hence, the features that a particular language provides directly affect the manner in which software is written. A good language should allow the programmer to write applications easily using abstractions that are appropriate to the domain they are working in. They should also allow code to be written that is safe, maintainable, portable and efficient, either when interpreted or translated to machine code. At present general purpose languages are a popular choice for writing software. They promise a feature set that is sufficient to write code suitable for a wide range of applications. However, there has been an increasing recognition that no single language can ever achieve this ideal. The mere fact that a multitude of languages exist many being used in niche application domains is evidence of this claim.

New language features provide concise notations, improved safety checks, and improved efficiency. Unfortunately, language design and implementation is difficult and time consuming. On the one hand developing and formalising the semantics of languages involves extensive experimentation, rigorous reasoning and widespread advocacy among the general programming community. On the other hand the implementation of a language is challenging. They must lex, parse and analyse source code, optimise the intermediate representation and the generate machine code for a variety of architectures. This is not all; language implementations also require a suite of tools such as debuggers, profilers, and language aware editors. Collectively this known as the programming environment.

Chapter 2 covered these points and suggested a solution based on reuse of language infrastructure. The basic idea is that domain specific features be added to existing general purpose programming languages. In fact entire languages can be implemented in this manner, via a combination of embedding—implementing a language as a library in a rich host language; known as EDSLs or DSELs—and additional safety checks and domain specific optimisations. Two high-level approaches to language extension were surveyed: extension via language extension features and via compiler extension. Both approaches are promising. More importantly they can be used in conjunction. However, the primary focus of this dissertation is on the second approach, that of compiler extension. In particular it

focuses on a specific kind of extensible compiler: the plug-in compiler. These compilers load modules dynamically to extend the language and provide new optimisations, also allowing the safety and efficiency of the language to be improved in the process.

Chapter 3 included a case study where we used language extension features, in the form of meta-programming, to optimise an embedded domain specific language called Pan. This chapter was included for two reasons. First, it demonstrated that meta-programming languages still have a way to go before they can be used actively for language extension. A number of shortcomings in a state-of-the-art compile-time meta-programming language, Template Haskell, were covered. This provided a good motivation for plug-in compilers; they approach the problem of extension from the unsafe, yet powerful space of solutions. Second, the techniques used in this chapter were, for the most part, readily applicable inside a plug-in compiler. The similarities between meta-programming and plug-in optimisations are interesting in their own right.

Chapter 4 began with a survey of techniques that could be used to provide back-end plug-in optimisation capability. I then presented a case study in back-end plug-ins, specifically plug-in optimisations, for the same EDSL as the previous chapter. A monadic API that provides functionality similar to Template Haskell was presented and followed by an explanation of a new domain specific optimisation for Pan called image lifting. I then showed how this optimisations was implemented using the API. For those that were interested implementation details specific to the Glasgow Haskell Compiler were present. I showed how back-end plug-in support was added, and the manner in which the API was implemented. Additionally, benchmarks proving the efficacy of the image lifting optimisation were provided. Finally, suggestions were put forward for a plug-in DSL that would make the writing of optimisations convenient and safe.

Chapter 5 tackled the issue of extensible data types, also known as the *expression problem*. The expression problem poses the question of how one can extend the variants of a mutually recursive data type without modifying existing source code and while still allowing new functions to be easily defined on the data type. We also required that the solution to this problem be statically typeable. Several solutions to this problem have been proposed for languages with advanced type systems. I have provided a solution to the expression problem for the Haskell language using a combination of multi-parameter type classes with recursive dictionaries and existential types. This work provides a foundation upon which front-end plug-ins can be written. Although another solution to the expression problem exists for Haskell it is not obvious how well it would work in a plug-in setting.

Chapter 6 directly used the solution to the expression problem for Haskell. I showed how front-end plug-in support was added to a compiler, written in Haskell, for a small Haskell-like language. Using this infrastructure a case study was presented in which *list comprehensions* were added to the language. This particular plug-in is typical of front-end plug-ins; it extends the code in a number of phases of the compiler namely the lexer/parser, renamer, and type checker. The new abstract syntax for list comprehensions is then desugared into existing language constructs.

The results of Chapters 4, 5 and 6 provide a generic framework for extending the

expressiveness, safety and efficiency of languages and the case studies provide a broad, but not deep, demonstration of their use. The most promising direction for future work lies in the design and implementation of useful plug-in DSLs that make writing front-end and back-end plug-ins easier and less error prone.

Appendix A

Module *Pan.Image*

Below is a listing of just the parts of *module Image* needed to understand the examples presented in Chapter 3.

```
module Pan.Image
where

type Point = (Float, Float)
type Colour = (Float, Float, Float, Float)
type Image c = Point → c
type ImageC = Image Colour
type Warp = Point → Point

whiteT :: Colour
whiteT = (0, 0, 0, 0)

whiteH, blackH :: Colour
whiteH = (1, 1, 1, 0.5)
blackH = (0, 0, 0, 0.5)

lift0 h     = λp → h
lift1 h f1 = λp → h (f1 p)
lift2 h f1 f2 = λp → h (f1 p) (f2 p)


empty :: ImageC
empty = lift0 whiteT

distO :: Point → Float
distO (x, y) = sqrt (x * x + y * y)

swirl :: Float → Warp
swirl r p = rotateP ((distO p) * (2 * pi / r)) p

cOver :: Colour → Colour → Colour
cOver (r1, g1, b1, a1) (r2, g2, b2, a2) = (h r1 r2, h g1 g2, h b1 b2, h a1 a2)
  where h x1 x2 = a1 * x1 + (1 − a1) * x2

over :: ImageC → ImageC → ImageC
```

$over = lift2\ cOver$

Appendix B

Images used in image lifting benchmarks

B.1 *WhiteOnRedOnBlack*

Three cheap, liftable images are placed on top of each other in this effect. Since each image is so cheap to compute the extra overhead of pre-computing the values of lifted images and passing that to the inner loop of the display function actually results in 5% slow down.

```
effect :: UI DisplayFun
effect = imageToDisplayFun $ whiteI `over` redI `over` blackI

whiteI, redI, blackI :: ImageC

whiteI _ = Col 1 1 1 0.5
redI _ = Col 1 0 0 0.5
blackI _ = Col 0 0 0 1
```

B.2 *Stripes*

The simplest example of an image that is only dependent on its y axis. Use has been made of faster versions of *even* and *floor* in this example. If we use the library definitions of these functions it leads to a larger speed difference between the two, although naturally both images are displayed slower.

```
effect :: UI DisplayFun
effect = imageToDisplayFun stripesImage

stripesImage :: ImageC
stripesImage (Pt _ y) = if even' (floor' y) then red else white

floor' :: Float → Int
floor' (F # f) = case ltFloat # f 0.0 # of
  True → I # ((float2Int # f) -# 1#)
  _ → I # (float2Int # f)
```

```
even' :: Int → Bool
even' (I # n) =
  case modInt # n 2 # of
    0# → True
    _ → False
```

B.3 *ColouredStripes*

The computation of the row colour for each stripe is a little more expensive in this example.

```
effect :: UI DisplayFun
effect = do
  imageToDisplayFun $ colouredStripes 20
colouredStripes barWidth (Pt _ y) =
  if even (floor (y / barWidth)) then blueShade v else redShade v
  where
    v = (abs y `realMod` barWidth) / barWidth
blueShade v = Col 0 0 v 0.5
redShade v = Col v 0 0 0.5
realMod :: RealFrac a ⇒ a → a → a
x `realMod` y =
  let (i, f) = properFraction (x / y)
  in x − (fromIntegral i ∗ y)
```

B.4 *StripesOnStripes*

We layer coloured stripes of differing widths on top each other in this example. The image lifting pass correctly lifts both stripe images.

```
effect :: UI DisplayFun
effect = do
  imageToDisplayFun $ colouredStripes 7 `over` colouredStripes 11
colouredStripes barWidth (Pt _ y) =
  if even (floor (y / barWidth)) then blueShade v else redShade v
  where
    v = (abs y `realMod` barWidth) / barWidth
blueShade v = Col 0 0 v 0.5
redShade v = Col v 0 0 0.5
realMod :: RealFrac a ⇒ a → a → a
x `realMod` y =
```

```
  let (i, f) = properFraction (x / y)
  in x − (fromIntegral i ∗ y)
```

B.5 *StripesOfWidth*

This example was added to show that the image lifting pass correctly handles the fact the top-level image generator, *stripesOfWidth*, is applied to various arguments.

```
effect :: UI DisplayFun
effect = imageToDisplayFun $ stripesOfWidth 10 red green

stripesOfWidth :: Frac → Colour → Colour → ImageC
stripesOfWidth width colour1 colour2 (Pt _ y) =
  if even (floor $ y / width) then colour1 else colour2
```

B.6 *CircleOnStripes*

A mixture of liftable and un-liftable images appears in this example. The image lifting pass correctly lifts only the stripe image.

```
effect = imageToDisplayFun $ circle 20 `over` stripes

circle r (Pt x y) = if x ∗ x + y ∗ y < r ∗ r then blue else invisible

stripes (Pt _ y) = if even (floor y) then red else invisible
```

Appendix C

Dictionary translations of *module F0_Alpha*

The code demonstrates the resulting of performing dictionary translation (in the style of Wadler and Blott [102]) on the code in Figures 5.4a – 5.4g. It makes it clear where the recursive dictionaries are built.

```
module Alpha
where
data P d
u = ⊥
 {-DI stands for dictionary implicity. D is an explicit dictionary -}
 {-class Sat a where dict :: a -}
data SatDI a = SatDI{ dict :: a }
data Exp (cxt :: ∗ → ∗) = forall b. MkExp (AlphaDI cxt b, b)
data Exp_0 cxt = Var String
   | Lam String (Exp cxt)
   | App (Exp cxt) (Exp cxt)
 {-class Sat (cxt b) ⇒ Alpha cxt b where -}
 {-alpha :: P cxt → b → (String, String) → Exp cxt -}
data AlphaDI cxt b =
  AlphaDI{ alpha :: P cxt → b → (String, String) → Exp cxt }

 {-instance (Sat (cxt (Exp cxt)), Sat (cxt (Exp_0 cxt))) ⇒ Alpha cxt (Exp_0 cxt) -}
alphaDExp0 :: forall cxt. (SatDI (cxt (Exp cxt)), SatDI (cxt (Exp_0 cxt))) →
  AlphaDI cxt (Exp_0 cxt)
alphaDExp0 (satExp, satExp0) = AlphaDI{ alpha = alpha′ }
  where
    alpha′ (_ :: P cxt) (Var v) =
      λs → var (satExp, satExp0) (u :: P cxt) (swap s v)
```

```
    alpha' (_ :: P cxt) (Lam v body) =
      case body of
        MkExp (alphaD, body') ->
          \s -> lam (satExp, satExp0) (u :: P cxt) (swap s v)
                (alpha alphaD (u :: P cxt) body' s)
    alpha' (_ :: P cxt) (App a b) =
      case a of
        MkExp (alphaDa, a') ->
          case b of
            MkExp (alphaDb, b') -> \s ->
              app (satExp, satExp0)
                (u :: P cxt) (alpha alphaDa (u :: P cxt) a' s)
                  (alpha alphaDb (u :: P cxt) b' s)


 {-instance Sat (cxt (Exp cxt)) => Alpha cxt (Exp cxt) -}
alphaDExp :: forall cxt. SatDI (cxt (Exp cxt)) -> AlphaDI cxt (Exp cxt)
alphaDExp satExp = AlphaDI{ alpha = alpha' }
  where
    alpha' (_ :: P cxt) exp =
      case exp of
        MkExp (alphaD, e) ->
          \s -> alpha alphaD (u :: P cxt) e s
swap :: (String, String) -> String -> String
swap ((a, b) :: (String, String)) = \(o :: String) -> if a == o then b else o


var :: forall cxt. (SatDI (cxt (Exp cxt))
                   , SatDI (cxt (Exp_0 cxt))) -> P cxt -> String -> Exp cxt
var (satExp, satExp0) (_ :: P cxt) =
  \(x1 :: String) -> MkExp (alphaDExp0 (satExp, satExp0), Var x1)
lam :: forall cxt. (SatDI (cxt (Exp cxt))
                   , SatDI (cxt (Exp_0 cxt))) -> P cxt
                    -> String -> Exp cxt -> Exp cxt
lam (satExp, satExp0) (_ :: P cxt) =
  \(x1 :: String) (x2 :: Exp cxt)
  -> MkExp (alphaDExp0 (satExp, satExp0), Lam x1 x2)
app :: forall cxt. (SatDI (cxt (Exp cxt))
                   , SatDI (cxt (Exp_0 cxt))) -> P cxt -> Exp cxt
                    -> Exp cxt -> Exp cxt
app (satExp, satExp0) (_ :: P cxt) =
  \(x1 :: Exp cxt) (x2 :: Exp cxt)
  -> MkExp (alphaDExp0 (satExp, satExp0), App x1 x2)
```

```
  --
  -- Capping class
  --
data AlphaEnd b
 {-class Alpha AlphaEnd b ⇒ AlphaCap b -}
data AlphaCapDI b = AlphaCapDI{ alphaD :: AlphaDI AlphaEnd b }
  --
  -- d and d0 are the recursive dictionaries for "instance AlphaCap (Exp
  -- AlphaEnd)" and "instance AlphaCap (Exp0 AlphaEnd)" respectively.
  --

 {-instance AlphaCap (Exp0 AlphaEnd) -}
d0 :: AlphaCapDI (Exp_0 AlphaEnd)
d0 = AlphaCapDI{ alphaD = alphaDExp0 (satD d, satD d0)}
 {-instance AlphaCap (Exp AlphaEnd) -}
d :: AlphaCapDI (Exp AlphaEnd)
d = AlphaCapDI{ alphaD = alphaDExp (satD d)}
 {-instance AlphaCap b =¿ Sat (AlphaEnd b) -}
satD :: AlphaCapDI b → SatDI (AlphaEnd b)
satD _ = SatDI{ dict = error "Capped at Alpha"}
 {-test = alpha (var (u::P AlphaEnd) "x") ("x", "y") -}
test = let p = u :: P AlphaEnd
       in  alpha (alphaDExp (satD d)) p
                 (var (satD d, satD d0) p "x") ("x", "y")
```

Appendix D

Plug-in functions in PHRaC

This appendix presents the plug-in functions as they were actually implemented. The syntactic verbosity would be greatly reduced by an implementation of the translation presented in Chapter 5.

D.1 Class declarations in PHRaC

D.1.1 Module *SyntaxTransformation*

```
data TransD ascxt cxt a =
  TransD{ rwExpD :: Proxy cxt → a → NA (AS.EXP ascxt)
        , rwApp1D :: a → EXP (TransD ascxt cxt) → NA (AS.EXP ascxt)
        , rwApp2D :: a → EXP (TransD ascxt cxt)
            → EXP (TransD ascxt cxt)
            → NA (AS.EXP ascxt)
        , transExt :: cxt a }
type TransCxt ascxt cxt = TransD ascxt cxt
type EXPTrans ascxt cxt = EXP (TransD ascxt cxt)
class (Sat (TransD ascxt cxt a)
  , Expr (TransD ascxt cxt) a
  ) ⇒ Transform ascxt cxt a where
  rwExp  :: Proxy cxt → a → NA (AS.EXP ascxt)
  rwApp1 :: a → EXP (TransD ascxt cxt) → NA (AS.EXP ascxt)
  rwApp2 :: a → EXP (TransD ascxt cxt) → EXP (TransD ascxt cxt)
              → NA (AS.EXP ascxt)
```

D.1.2 Module *TypeInference*

```
type InferCxt cxt = OverD (InferD cxt)
type Infer' cxt a =
```

```
  Assumps → a → TI cxt ([Constraint], Type, EXP (InferCxt cxt))
data InferD cxt a = InferD{ tiExprD :: Infer' cxt a
  , inferExt :: cxt a }
class (Sat (InferCxt cxt a)
  , Overload (InferD cxt) a
  ) ⇒ Inference cxt a where
  tiExpr :: Infer' cxt a
```

D.1.3 Module *Desugar*

```
data DesugarD cxt a =
  DesugarD{ desugarExprD :: a → ST' cxt ExpInterp
    , desugarExt :: cxt a }
type DesugarCxt cxt = OverD (InferD (DesugarD cxt))
class (Sat (DesugarCxt cxt a)
  , Inference (DesugarD cxt) a
  ) ⇒ Desugar cxt a where
  desugarExpr :: a → ST' cxt ExpInterp
```

D.2 The list-comprehension plug-in

D.2.1 Renaming

```
instance (Sat (TransD ascxt cxt (ExpLC (TransD ascxt cxt)))
      , Sat (TransD ascxt cxt (Exp (TransD ascxt cxt)))
      , Sat (TransD ascxt cxt (EXP (TransD ascxt cxt)))
      , AS.Sat (ascxt (AS.ExpLC ascxt))
      , AS.Sat (ascxt (AS.Exp ascxt))
      ) ⇒ Transform ascxt cxt (ExpLC (TransD ascxt cxt)) where
  rwExp (_ :: s) (ListCompE exp quals) = do
        -- renaming of Quals needs to go first so that
        -- variables bound by patterns are in scope.
    enterBlock
    quals' ← mapM rw_qual quals
    exp' ← rwExp (⊥ :: s) exp
    exitBlock
    return (AS.listCompE exp' quals')
  rwApp1 e1 (e2 :: EXP (TransD ascxt cxt)) =
    do e1' ← rwExp p e1
       e2' ← rwExp p e2
```

```
      return (AS.appE e1' e2')
    where p = ⊥ :: Proxy cxt
  rwApp2 f1 (f2 :: EXP (TransD ascxt cxt)) e2 =
    do let e1 = appE (mkExpLC f1) f2
       e1' ← rwExp p e1
       e2' ← rwExp p e2
       return (AS.appE e1' e2')
    where p = ⊥ :: Proxy cxt
rw_qual :: Sat (TransCxt ascxt cxt (EXP (TransCxt ascxt cxt)))
  ⇒ Qual (EXP (TransCxt ascxt cxt)) → NA (AS.Qual (AS.EXP ascxt))
rw_qual (GenQual pat (exp :: EXP (TransCxt ascxt cxt))) = do
    exp' ← rwExp (⊥ :: Proxy cxt) exp
    pat' ← rw_pattern pat
    return (AS.GenQual pat' exp')
rw_qual (GuardQual (exp :: EXP (TransCxt ascxt cxt))) = do
  exp' ← rwExp (⊥ :: Proxy cxt) exp
  return (AS.GuardQual exp')
```

D.2.2 Type inference

tiExpr

```
instance (Sat (OverD (InferD cxt) (ExpLC (InferCxt cxt)))
  , Sat (OverD (InferD cxt) (EXP (InferCxt cxt)))
  ) ⇒ Inference cxt (ExpLC (InferCxt cxt)) where
  tiExpr as (ListCompE exp quals) = do
    (ps, as', quals') ← tiQuals as quals
    alpha ← liftST $ freshTypeVar KindStar
    (ps', t, exp') ← tiExpr (as' `merge` as) exp
    ecs ← unify [] t alpha
    return (ps' ++ ps ++ map EC ecs, tyList alpha, listCompE exp' quals')
tiQual :: Sat (InferCxt cxt (EXP (InferCxt cxt)))
  ⇒ Infer cxt (Qual (EXP (InferCxt cxt))) Assumps
tiQual as (GenQual pat exp) = do
  (ps1, as1, t1) ← tiPat pat
  (ps2, t2, exp') ← tiExpr as exp
  alpha ← liftST $ freshTypeVar KindStar
  ec1 ← unify [] t1 alpha
  ec2 ← unify [] t2 (tyList alpha)
  return (ps1 ++ ps2 ++ map EC (ec1 ++ ec2), as1, GenQual pat exp')
tiQual as (GuardQual exp) = do
```

```
  (ps, t, exp') ← tiExpr as exp
  qs ← unify [] t boolType
  return (ps ++ map EC qs, emptyAssumps, GuardQual exp')
```

tiQuals

```
tiQuals :: Sat (InferCxt cxt (EXP (InferCxt cxt)))
   ⇒ Infer cxt [Qual (EXP (InferCxt cxt))] Assumps
tiQuals as [qual] = do
  (ps, as', qual') ← tiQual as qual
  return (ps, as', [qual'])
tiQuals as (qual : quals) = do
  (ps', as', qual') ← tiQual as qual
  (ps'', as'', quals') ← tiQuals (as' `merge` as) quals
  return (ps' ++ ps'', as' `merge` as'', qual' : quals')
```

D.2.3 desugarExpr

```
desugarExpr (ListCompE exp []) = do
  exp' ← desugarExpr exp   -- parser never produces this case
  return (Con consDataId [exp', Con nilDataId []])
desugarExpr (ListCompE exp (GenQual pat bindExp : quals)) = do
  bindExp' ← desugarExpr bindExp
  okId ← freshValVar
  x ← freshValVar
  okBody ← do
    alt1Body ← desugarExpr (ListCompE exp quals)
    return (Case (Var x)
      [Alt{ alt_pat = pat
          , alt_exp = alt1Body}
          , Alt{ alt_pat = PWildcard
          , alt_exp = Con nilDataId []}
      ])
  return $
    Letrec
      [ValBind{ vbind_name = okId
              , vbind_type = Nothing
              , vbind_pats = [PVar x]
              , vbind_exp = okBody
              , vbind_info = emptyInfo}]
      (Var concatMapId `App` Var okId `App` bindExp')
```

```
desugarExpr (ListCompE exp (GuardQual bexp : quals)) = do
  bexp' ← desugarExpr bexp
  rest ← desugarExpr (ListCompE exp quals)
  return $ (If bexp' rest (Con nilDataId []))
```

Bibliography

[1] Cryptol. http://www.cryptol.net/.

[2] Firefox browser. http://www.mozilla.org/.

[3] GNU Emacs. http://www.gnu.org/software/emacs/emacs.html.

[4] Happy. http://haskell.org/happy/.

[5] International Standard. ISO 9899:1990:Information technology – Programming Languages – C.

[6] International Standard. ISO 9899:1999:Information technology – Programming Languages – C.

[7] International Standard. ISO/IEC 8652:1995(E): Information technology – Programming Languages – Ada.

[8] Object Management Group. Common Object Request Broker Architecture (CORBA). http://www.omg.org.

[9] The GIMP. http://www.gimp.org/.

[10] Winamp. http://www.winamp.com/.

[11] XML Path Lanaguage (XPath) Version 1.0. W3C Recommendation 16 November 1999. http://www.w3.org/TR/xpath.

[12] XSL Transformations (XSLT) Version 1.0. W3C Recommendation 16 November 1999. http://www.w3.org/TR/xslt.

[13] Gerald Aigner, Amer Diwan, David L. Heine, Monica S. Lam David, L. Moore, Brian R. Murphy, and Constantine Sapuntzakis. An Overview of the SUIF2 Compiler Infrastructure. Technical report, Stanford University, 2000.

[14] Eric Allen, David Chase, Victor Luchangco, Sukyoung Ryu, Guy L. Steele Jr, and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2005.

[15] Martin Alt, Uwe Aßmann, and Hans van Someren. Cosy Compiler Phase Embedding with the CoSy Compiler Model. In *Computational Complexity*, pages 278–293, 1994.

[16] Krasimir Angelov and Simon Marlow. Visual Haskell: A full-featured Haskell development environment. In *Haskell'05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 5–16, Tallinn, Estonia, September 2005. ACM Press.

[17] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[18] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V.Pascual. Centaur: the system. In *SDE 3: Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 14–24, Boston, USA, November 1988. ACM Press.

[19] C. Brabrand, M. Schwartzbach, and M. Vanggaard. The `metafront` System: Extensible Parsing and Transformation. In *LDTA'03: Proceedings of the 3rd ACM SIGPLAN Workshop on Language Descriptions, Tools and Applications*, April 2003.

[20] Martin Bravenboer and Eelco Visser. Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2004. ACM Press.

[21] Kim B. Bruce. Some Challenging Typing Issues in Object-Oriented Languages: Extended Abstract. In , volume 82.8 of *Electronic Notes in Theoretical Computer Science*, pages 1–29, 2003.

[22] Helen Cameron, Peter King, and Simon Thompson. Modeling Reactive Multimedia: Events and Behaviors. *Multimedia Tools and Applications*, 19(1):53–77, 2003.

[23] Manuel M. T. Chakravarty, Gabriele Keller, and Patryk Zardarnowski. A functional perspective on SSA optimisation algorithms. In *Proceedings of the 2nd International Workshop on Compiler Optimisation Meets Compiler Verification (COCV 2003)*, volume 82.2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2003.

[24] Douglas Crockford. Javascript: The World's Most Misunderstood Programming Language . `http://javascript.crockford.com/javascript.html`.

[25] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control depen-

dence graph. *ACM Transactions on Programming Language Systems (TOPLAS'91)*, 13(4):451–490, 1991.

[26] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vanderboorde, and Todd Veldhuizen. Generative Programming and Active Libraries. In *Generic Programming: International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39, Dagstuhl Castle, Germany, April/May 1998. Springer.

[27] Krzysztof Czarnecki, John O'Donnell, Jörg Striegnitz, and Walid Taha. DSL Implementation in MetaOCaml, Template Haskell, and C++. URL: http://www.cs.rice.edu/~taha/publications.html, 2003.

[28] Olivier Danvy and Karoline Malmkjaer. Intensions and Extensions in a Reflective Tower. In *Proceedings of the 1988 ACM conference on LISP and Functional Programming (LFP)*, pages 327–341. ACM Press, 1988.

[29] Atze Dijkstra. *Stepping through Haskell*. PhD thesis, Department of Information and Computing Sciences, 2005.

[30] ECMAScript. Standard ECMA-262 ECMAScript Language Specification 3rd edition (December 1999) http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[31] Conal Elliott. Functional Implementations of Continuous Modeled Animation. *Lecture Notes in Computer Science*, 1490:284–, 1998.

[32] Conal Elliott. Functional Image Synthesis. In *Proceedings Bridges 2001, Mathematical Connections in Art, Music, and Science*, 2001.

[33] Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, May 2003.

[34] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[35] H. Emmelmann, F.-W. Schröer, and L. Landwehr. Beg: a generation for efficient back ends. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 227–237, New York, NY, USA, 1989. ACM Press.

[36] Dawson R. Engler. Interface Compilation: Steps toward Compiling Program Interfaces as Languages. *IEEE Transactions in Software Engineering*, 25(3):387–400, 1999.

[37] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: a language for high-level, efficient, and machine-independent dynamic code generation. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 131–144, New York, NY, USA, 1996. ACM Press.

[38] Erik Ernst. Family Polymorphism. In Jørgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object Oriented Programming (ECOOP'2001)*, volume 2072 of *LNCS*, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.

[39] ACE Associated Compiler Experts. CoSy Compilers: Overview of Construction and Operation. CoSy System Documentation, ACE Associated Compiler Experts 2003.

[40] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without Metaphysics. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP)*, pages 348–355, 1984.

[41] Matteo Frigo and Steven G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing: (ACASSP'98)*, pages 1381–1384, Seattle, Washington, USA, May 1998. IEEE Service Center.

[42] Jacques Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, Electronic Notes in Theoretical Computer Science, Sasaguri, Japan, November 2000. Elsevier Science.

[43] Jacques Garrigue. Private row types: abstracting the unnamed. *Electronic Notes in Theoretical Computer Science*, 2005.

[44] Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane, and William M. Waite. ELI: A Complete, Flexible Compiler Construction System. *CACM'92: Communications of the ACM*, 35(2):121–130, 1992.

[45] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the Type Inference Process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.

[46] Bastiaan Heeren, Johan Jeuring, S. Doaitse Swierstra, and Pablo Azero Alcocer. Improving type-error messages in functional languages. Technical Report UU-CS-2002-009, Institute of Information and Computing Science, University Utrecht, Netherlands, February 2002. Technical Report.

[47] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF: Reference Manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[48] W. Wilson Ho and Ronald A. Olsson. An Approach to Genuine Dynamic Linking. *Software—Practice and Experience*, 24(4):375–390, April 1991.

[49] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, New York, NY, USA, 2000.

[50] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, Robots, and Functional Reactive Programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[51] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore Music Notation - An Algebra of Music. *Journal of Functional Programming*, 6(3):465–483, 1996.

[52] R.J.M Hughes. Restricted data types in Haskell. In *Proceedings of the 1999 Haskell Workshop*, 1999.

[53] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[54] Stanley Jefferson and Daniel P. Friedman. A Simple Reflective Interpreter. In *IMSA'92 International Workshop on Reflection and Meta-level Architecture*, Tokyo, November 92.

[55] Mark P. Jones. A theory of qualified types. In *European Symposium on Programming (ESOP'92)*, volume 582 of *LNCS*, Rennes, France, 1992. Springer-Verlag.

[56] Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. Draft; Online since 30 Sep. 2004; Updated 13 June 2005.

[57] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161, New York, NY, USA, 1986. ACM Press.

[58] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing Object-Oriented and Functional Design to Promote Re-use. In *ECOOP'98 – Object Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 91–113. Springer Berlin / Heidelburg, 1998.

[59] Ralf Lämmel. *Extensible grammars*, on the `comp.compilers` newsgroup, http://compilers.iecc.com/comparch/article/04-12-111, 2004.

[60] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[61] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*. ACM Press, September 2005.

[62] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[63] Konstantin Läufer. Type Classes with Existential Types. *Journal of Functional Programming*, 6(3):485–517, May 1996.

[64] Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. Cetus – An Extensible Compiler Infrastructure for Source-to-Source Transformation. In *LCPC'03: Proceedings of the 16th Annual Workshop on Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 539–553, College Station, TX, USA, October 2003.

[65] Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[66] Andres Löh and Ralf Hinze. Open data types and open functions. In *PPDP'06: Eighth ACM-SIGPLAN International Symposium on Principles and Practice of Declaritive Programming*, Venice, Italy, July 2006.

[67] Ian Lynagh. Unrolling and Simplifying Expressions with Template Haskell. URL: http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/, May 2003.

[68] R. Marti and T. Murer. Extensible Attribute Grammars. Technical Report TIK-Report 92-6, Laboratory of Computer Engineering and Networks, Swiss Federal Institute of Technology, Gloriastrasse 35, ETH-Zentrum, CH-8092 Zürich, Switzerland, 1992.

[69] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[70] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[71] Henrik Nilsson, John Peterson, and Paul Hudak. Functional Hybrid Modeling. In *Proceedings of PADL'03: 5th International Workshop on Practical Aspects of Declarative Languages*, pages 376–390. Springer Verlag LNCS 2562, January 2003.

[72] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proc. 12th International Conference on Compiler Construction*, number 2622 in Lecture Notes in Computer Science, pages 138–152. Spring-Verlag, April 2003.

[73] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erike Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. Technical Report IC/2004/64, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, 2004.

[74] Martin Odersky, Vincent Cremet, Christine Rockl, and Matthias Zenger. A Nominal Theory of Objects with Dependent Types. Technical Report IC/2002/070, Écolé Polytechnique Féderalé de Lausanne, 2002.

[75] André Pang, Don Stewart, Sean Seefried, and Manuel M. T. Chakravarty. Plugging Haskell in. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 10–21. ACM Press, 2004.

[76] Emir Pasalic. *The roll of Type Equality in Meta-programming.* PhD thesis, OGI School of Science and Engineering, 2004.

[77] Simon Peyton Jones. Compiling Haskell by Program Transformation: A Report from the Trenches. In *ESOP'96: The 6th European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 18–44, 1996.

[78] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the Rules: Rewriting as a practical optimisation technique in GHC. *International Conference on Functional Programming (ICFP 2001). Haskell Workshop.*, September 2001.

[79] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: Type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, Computer and Information Science Department, Levine Hall, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104-6389, July 2004.

[80] Simon Peyton Jones et al. The Haskell 98 Report. http://www.haskell.org, 1998.

[81] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In S. A. Schuman, editor, *New Directions in Algorithmic Languages*, pages 157–168, 1975.

[82] Arch D. Robinson. The Impact of Economics on Compiler Optimization. In *Proceedings of the ACM 2001 Java Grande Conference, Standford*, pages 1–10, June 2001.

[83] Meurig Sage. FranTk – a declarative GUI language for Haskell. *ACM SIGPLAN Notices*, 35(9):106–117, 2000.

[84] Tim Sheard. Accomplishments and Research Challenges in Meta-programming. In *Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, volume 2196 of *Lecture Notes in Computer Science*, pages 2–44. Springer, 2001.

[85] Tim Sheard and Simon Peyton Jones. Template Meta-Programming for Haskell. *ACM SIGPLAN Notices: PLI Workshops*, 37(12):60–75, 2002.

[86] Brian Cantwell Smith. Reflection and Semantics in Lisp. *Conf. Rec. 11th ACM Symp. on Principles of Programming Languages*, pages 23–35, 1984.

[87] Oliver Steele. The IDE Divide. http://osteele.com/archives/2004/11/ides.

[88] Don Stewart and Manuel M. T. Chakravarty. Dynamic Applications From the Ground Up. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*. ACM Press, September 2005.

[89] Doaitse Swierstra. Combinator Parsers: From Toys to Tools. In Graham Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publishers, 2001.

[90] Walid Taha and Tim Sheard. Multi-Stage Programming with Explicit Annotations. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, Amsterdam, The Netherlands, June 1997. New York: ACM.

[91] Walid Taha and Tim Sheard. MetaML and Multi-Stage Programming with Explicit Annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

[92] Peter Thiemann. Towards a Type System for Analyzing JavaScript Programs. In *Programming Languages and Systems: 14th European Symposium on Programming (ESOP 2005)*, number 3444 in Lecture Notes in Computer Science, pages 408–422, 2005.

[93] Aaron Tomb and Cormac Flanagan. Automatic Type Inference via Partial Evaluation. In *PPDP'05: Seventh ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, Lisbon, July 2005. ACM Press.

[94] Todd Veldhuizen. Techniques for Scientific C++. Technical Report 542, Indiana University Computer Science, 2000.

[95] Todd L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University Computer Science, May 2004.

[96] Todd L. Veldhuizen and Dennis Gannon. Active Libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific Engineering and Computing (OO'98)*, Yorktown Heights, New York, 1998. SIAM Press.

[97] Eelco Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *RTA 2001: The 12th International Conference on Rewriting Techniques and Applications*, volume 2051 of *LNCS*, page 357, Utrecht, The Netherlands, May 2001. Spring-Verlag.

[98] Eelco Visser. Program Transformation with Stratego/XT: Rules, strategies, tools and systems in Stratego/XT 0.9. In *Domain Specific Program Generation*, volume 3016 of *LNCS*, page 216, Dagstuhl Castle, Germany, March 2004. Spring-Verlag.

[99] Philip Wadler. Comprehending monads. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, New York, NY, USA, 1990. ACM Press.

[100] Philip Wadler. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995. Springer-Verlag.

[101] Philip Wadler. *The expression problem*, Discussion on the Java Genericity mailing list, 1998.

[102] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Conference Record of the Sixteenth Annual Symposium on Principles of Programming Languages (POPL'89)*, pages 60–76, Austin, Texas, January 1989. ACM Press.

[103] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34–9, pages 148–159, N.Y., 27–29 1999. ACM Press.

[104] Mitchell Wand and Daniel P. Friedman. The Mystery of the Tower Revealed: A Nonreflective Description of the Reflective Tower. In *Proceedings of the 1986 ACM conference on LISP and Functional Programming (LFP)*, pages 298–307. ACM Press, 1986.

[105] Mitchell Wand and Daniel P. Friedman. The Mystery of the Tower Revealed: A Nonreflective Description of the Reflective Tower. *Lisp and Symbolic Computation*, 1:11–37, 1988.

[106] Gregory V. Wilson. Extensible programming for the 21st century. *Queue*, 2(9):48–57, 2005.

[107] Matthias Zenger and Martin Odersky. Extensible Data Types with Defaults. In *International Conference on Functional Programming (IFCP'01)*, pages 241–252, Firenze, Italy, September 2001.

[108] Matthias Zenger and Martin Odersky. Implementing Extensible Compilers. In *Workshop on Multiparadigm Programming with Object-Oriented Languages*, volume 7 of *NIC Series*, pages 61–80, Budapest, Hungary, June 2001. NIC-Directors.

[109] Matthias Zenger and Marting Odersky. Independently Extensible Solutions to the Expression Problem. Technical Report IC/2004/33, École Polytechnique Fédérale de Lausanne, Lausanna, Switzerland, 2004.