COMPILED GENERICS FOR FUNCTIONAL PROGRAMMING LANGUAGES

Matthew Roberts

B.Com./B.Sc. (Hons)

This dissertation is presented for the degree of

Doctor of Philosophy

at

March 2011

Declaration

I certify that the research presented in this thesis is the original work of the author except where otherwise indicated. Any help and assistance that I have received in my research work and the preparation of the thesis itself have been appropriately acknowledged. This work has not been submitted for a degree or any other qualification to this or any other university or institution. All verbatim extracts have been distinguished by quotations, and all sources of information have been specifically acknowledged.

Acknowledgements

No doctoral thesis is completed without significant support from individuals other than the author and this work is no exception. Dr Dominic Verity and Dr Tony Sloane have both been wonderful research mentors and colleagues and this work would be much diminished without their efforts. Dr Richard Garner's type-theoretic advice has been invaluable as have my various fruitful discussions with Micah McCurdy, Michael Olney, Jette Viethen, Barry Jay, Thomas Given-Wilson and Jose Veragas. Sanity has prevailed largely thanks to the expert advice of Justine Corry.

Most important however, is the personal support given by my wife, without which this work would not exist at all. This thesis, and indeed my life, is happily devoted to her.

Abstract

We address the problem of extending existing functional language compilers to support generic programming constructs, such as those that arise in term rewriting and datatype generic programs.

In particular, we present a compiler capable of compiling a wide range of generic programs in a way which substantially reduces the execution overhead traditionally associated with such programs, without requiring type classes. We explicitly build upon a baseline functional compiler by extending it to support a universal spine view of data and by adding a mechanism for building polymorphic functions from monomorphic ones.

This work employs variants of standard pattern compilation and lambda lifting translations that render these generic extensions into efficient code while making only modest modifications to our baseline compiler and its run-time. We show that type inference (with annotations for higher-ranked types and polymorphic recursion) can be maintained, by building all the mechanisms needed for type inference of generic programs on top of an existing variant of Damas and Milner's algorithm W.

We demonstrate that this compiler achieves type safe and efficient compiled generic programs by showing the breadth of generic programs that we can encode with it and by benchmarking the execution speed and memory use of the programs output by the compiler. The generic programs we demonstrate are generic transformations, generic queries, generic traversals, generic equality, generic show plus a large number of variants of these. We also provide a proof of the soundness of the type system which underpins the compiler.

In summary, the primary outcome of this research is a new compiler for generic programs which uses a unique combination of encoding techniques and which generates efficient generic code. For each of our extensions (polymorphic functions with specific behaviour and the explicit spine view) we provide new algorithms for pattern compilation, type inference and conversion to primitive operations. These can be easily incorporated into existing functional language compilers.

Contents

1 Introduction **7**

1.1 The Problem . . . 7

1.2 The Context . . . 7

1.3 The Solution . . . 9

1.4 This Thesis . . . 10

2 Generic Functions **13**

2.1 Pattern Terminology . . . 13

2.2 DGEN . . . 14

2.3 Generic Update . . . 18

2.4 Generic Query . . . 20

2.5 Generic Traversal . . . 20

2.5.1 Bottom-Up Update . . . 22

2.5.2 Top-Down Update . . . 23

2.5.3 Top-Down Query . . . 23

2.5.4 Bottom-Up Query . . . 24

2.6 Generic Equality . . . 24

2.7 Generic Show . . . 26

2.8 Other "Generic" Functions . . . 27

2.8.1 Object Oriented Generics . . . 27

2.8.2 Term Rewriting . . . 27

2.8.3 Datatype Generics . . . 27

2.9 Summary . . . 27

3 Encoding Generic Functions **29**

3.1 Evaluation Criteria . . . 30

3.2 Approaches . . . 31

3.2.1 Term Rewriting 31
3.2.2 Generic Calculii 32
3.2.3 Datatype Generics 32
3.2.4 Others 33
3.3 Capabilities 33
3.3.1 Polymorphic Functions with Specific Behaviour 33
3.3.2 Our solution: Function Extension with a Single Operator 40
3.3.3 Structure Agnosticism 41
3.3.4 Our Solution: Compiling Explicit Spine View 48
3.4 Summary 50

4 A Baseline Functional Language Compiler 51
4.1 Pattern Compilation 53
4.1.1 Variable Patterns 55
4.1.2 Constructor Patterns 55
4.1.3 Literal Patterns 57
4.1.4 Heterogenous Patterns 58
4.1.5 Empty Patterns 59
4.1.6 Related Work 59
4.2 Dependency Analysis 60
4.2.1 Polymorphic Recursion 60
4.2.2 Dependency Analysis Algorithm 62
4.2.3 Related Work 63
4.3 Type Inference 63
4.3.1 FCP 63
4.3.2 Algorithm 64
4.3.3 Related Work 67
4.4 Lambda Lifting 67
4.4.1 Lifting Cases 68
4.4.2 Algorithm 70
4.4.3 Related Work 71
4.5 Conversion to Imperative 72
4.6 Summary 72

5 Compiling and Typing Polymorphic Functions with Specific Behaviour 75
5.1 Overall Journey 76

5.2 Type Preserving Transformations 77
5.3 Accumulators 77
5.4 Deriving a type inference rule for ▷ 78
5.4.1 `extT` and `extQ` 78
5.4.2 Emulating the Type Constraints 81
5.5 Discussion 82
5.5.1 Comparison to `extQ` and `extT` 82
5.5.2 Comparison to Extension Typing 83
5.6 Higher Rank Types 84
5.7 Deriving a Higher Rank Mechanism from FCP 85
5.7.1 Discussion 90
5.8 Translating ▷ to `typeOf` 90
5.9 Definition of `typeOf` 91
5.9.1 Discussion of the Semantics of `typeOf` 92
5.9.2 Discussion of Time and Space Cost 92
5.10 Summary 93

6 Compiling and Typing Structure Agnosticism **95**
6.1 Overall Journey 96
6.2 An Unresolved SOURCE Detail 96
6.3 Converting `x(y)` to `kar`, `kdr` and `ispair` 97
6.4 Type Inference Rules for `kar`, `kdr` and `ispair` 100
6.5 Polymorphic Recursion 103
6.5.1 Deriving Polymorphic Recursion from FCP 104
6.6 Definitions of `kar`, `kdr` and `ispair` 109
6.6.1 The Epic Runtime 109
6.7 Application of Partially Applied Constructors to Values 111
6.8 A Prescription for Lonely Constructors in the Runtime 112
6.9 Summary 114

7 Evaluating Expressiveness **115**
7.1 Five Kinds of Generic Programs 116
7.1.1 Generic Update 116
7.1.2 Generic Query 117
7.1.3 Generic Traversal 117
7.1.4 Generic Equality and Generic Show 118

7.2 Variants . . . 119
7.2.1 Structure Modification (with Type Preservation) . . . 119
7.2.2 Datatype Aware Traversals with no “Boilerplate” . . . 121
7.2.3 Function Extension with Multiple Arguments . . . 123
7.2.4 Generic Zip-With . . . 124
7.2.5 SYB Primitive Operations . . . 125
7.2.6 Expanding the Set of Built-In Operations . . . 126
7.2.7 Strategic Rewriting . . . 128
7.3 Limitations . . . 131
7.3.1 Can’t Give Best Possible Type to `geq` . . . 131
7.3.2 Can’t Encode Generic Map . . . 132
7.3.3 Can’t Encode Generic Read . . . 133
7.4 Summary . . . 133

8 Evaluating Compilation **135**
8.1 A Note About our Benchmarks . . . 135
8.2 Compiled Speed . . . 136
8.2.1 Maintaining Compilation to `switch` Statements. . . . 136
8.2.2 Generic Libraries Cause Slowdown . . . 137
8.2.3 DGEN has Relatively Little Slowdown due to Generics . . . 137
8.2.4 Comparisons to Other Tools . . . 138
8.2.5 Comparison to SYB . . . 141
8.2.6 Comparison to Type-Class based Haskell Libraries . . . 142
8.2.7 Summary of Speed Benchmarks . . . 143
8.3 Memory Use due to Generics . . . 144
8.3.1 Comparison to SYB . . . 145
8.4 Optimisation Opportunities . . . 148
8.4.1 An Optimisation: Trivial Super-Combinator Elimination . . . 148
8.4.2 A Potentially Compromised Optimisation . . . 152
8.5 Summary . . . 153

9 Evaluating the Type System **155**
9.1 The Evaluation Relation . . . 157
9.2 The Typing Relation . . . 158
9.3 Soundness Proof . . . 160
9.4 Summary . . . 171

10 Conclusion **173**
10.1 A Type-Safe Function Extension Mechanism 173
10.2 A Compilation Scheme and a Type Inference Algorithm for Application Pattern Matches 173
10.3 Demonstration of Techniques 174
10.4 Future Work 174
10.4.1 More Expressive Spine Views 174
10.4.2 Object Oriented Application Pattern Matching 175
10.4.3 Integration Into Other Compilers 175
10.4.4 Extending Support for the Pattern Calculus 175
10.4.5 Function Extension with Fewer Restrictions 176
10.4.6 Failure as Trivial Success 176
10.4.7 Joining the Rewriting Game 176

Chapter 1

Introduction

1.1 The Problem

At the time of writing, no widely used compiler for a general purpose functional language uses a truly general mechanism specifically for compiling generic functions. Three widely used compilers (GHC [56], Clean [11] and UHC [20]) build compilation of generics on top of advanced and specialised features, such as type classes. A number of other systems admit generic functions but compromise upon either the efficiency of the compiled code or the safety of their type systems.

1.2 The Context

A *generic* function, in this thesis, is a function that can work over values of any datatype without overloading the definition of that function.

More specifically, we will be considering the following functions in the remainder of this thesis:

Generic Update A function that is capable of working over any data structure, applying a type-preserving transformation at certain nodes.

Generic Query A function that is capable of working over any data structure and accumulating a single-value result from calculations on certain kinds of nodes.

Generic Traversal Generic query and generic update require only *some* traversal of an unknown structure. When we are able to write functions that traverse an unknown structure *according to a particular traversal strategy*, we have generic traversal.

Generic Equality A function that can compute equality for values of all data types in the language.

Generic Show A function that can compute a string representation for values of all data types in the language.

Outside of advanced Haskell and Clean compilers, there certainly exists tools which can either interpret, pre-process, or compile untyped versions of these generic functions. They can be grouped into one of three categories:

Term Rewriting Term Rewriting is a complete model of computation where the result of a program is achieved through successive translations of the input.

Datatype Generic Datatype generics is a style of programming where the structure of the data that the function is operating over becomes an input to the function. This structure argument may be explicit or implicit but it will always be used to guide the evaluation. For example, one branch of evaluation is taken for sum types and another for product types.

Pattern Calculus The pattern calculus [42, 45, 46, 41] is built around the idea that functions and structures are colleagues in the job of computation and that neither should be dominant over the other. As a result, there is a reified notion of data, making it "first class" in a very deep sense. In particular, any data can act as a pattern in the pure pattern calculus.

There are three general purpose functional language compilers which can compile and run generic functions today. The first is GHC which now provides both Generic Haskell and Scrap Your Boilerplate (SYB) [61, 60, 59]. The other two are UHC and Clean where a datatype generic library has been integrated into the compiler. While all three are wonderful tools, the methods by which they achieve compilation of generic functions are not appropriate for use in other functional language compilers. Template Haskell is a meta-programming tool specific to Haskell. SYB and the datatype generic libraries in UHC and Clean all require at least type classes [89], which is a feature particular to Haskell and Clean. In this thesis we describe three small additions that can be made to *any* functional language compiler to realise compilation of generic programs. We will discuss further the mechanisms used in GHC, UHC and Clean when we survey the field in Chapter 3.

1.3 The Solution

In this thesis we present extensions to a *baseline functional compiler* that can type-check these functions and compile them to C. A *baseline functional compiler* is one that only includes features common to most functional compilers. In other words, it does not have language-specific or advanced features like Haskell's type classes or OCaml's objects/-modules [66]. We will need to add a few advanced features (like rank-2 types and polymorphic recursion) but we do not take them as assumptions, we show how these are added to the baseline compiler.

This approach brings the benefits of compilation of generic functions to any functional language. In particular, we show how *relatively orthogonal changes to a standard functional compiler* can provide this, keeping the most important characteristics of the original compiler intact. This means that our techniques can be applied to almost any functional language compiler without compromising other important features such as type safety.

We achieve this with three new techniques, demonstrated and validated in a working compiler.

A type-safe function extension mechanism We create a technique for extending monomorphic functions to polymorphic ones which is both type-safe and has low overhead at run time. Our technique is a hybrid of those used in Scrap Your Boilerplate and in the pattern calculus. Both of those are strongly typed (although in quite distinct ways) but neither is immediately amenable to compilation in a baseline functional compiler.

A compilation scheme for *application pattern matches* Application pattern matches are a generic way to pull apart data which have so-far only ever been interpreted (in RhoStratego [22] and bondi [29, 44] for example). We demonstrate how to compile them to fast-running C code.

A type inference scheme for application pattern matches Our method of compiling application pattern matches requires unique typing rules. We formulate and demonstrate rules, which are related to those used in the static pattern calculus [29], but which are more explicit and simpler to incorporate into a baseline functional compiler. We achieve these simpler rules by creating some useful program invariants during pattern compilation.

At a higher level, the compiler presented in this thesis fills a number of holes in the literature/available tools, as it provides:

A practical implementation of various calculii. The only existing implementation of the *pattern calculus* is the interpreted language bondi. Although we don't support all pattern calculus features, our compiler is the first compiled implementation of a significant part of the pattern calculus. The most complete current implementation of the *ρ-calculus* [15, 16] is RhoStratego. Unfortunately it has not been updated in line with the compiler needed to use it and so can't currently be run. For people wanting to use a typed and compiled implementation of the ρ-calculus, our compiler is now the closest approximation available and could be expanded into a full implementation with relatively little work. Application pattern matching is supported and explicit failure is emulated with function extension.

Faster generic programs. Our compiler is able to compile the above generic programs into executables that have less run-time overhead due to generics than existing technologies which don't require type classes[1] (relative to its baseline speed). This is very useful in functional programming where some of the tools for writing these generic functions execute the code quite slowly.

Optimisable generic programs. By compiling these generic functions to very simple constructs within the compiler, they are more easily optimised. We show in Section 8.4 how existing functional programming optimisations are preserved by the compiler and how they can operate efficiently around generic code. This is not possible with libraries for example.

A unique combination of generic techniques. In achieving all these things we have assembled a unique combination of techniques for dealing with generic functions, including some which are completely novel. Even where the techniques themselves are not new, their specific juxtaposition in our compiler is.

1.4 This Thesis

Chapter 2 describes in greater detail the generic functions handled by our compiler and gives an encoding of them. The encodings are given in a concrete syntax which our compiler can process, called DGEN. This chapter also gives a brief introduction to DGEN itself in order that the encodings are clear.

[1] In Section 8.2.6 we discuss some very fast generic libraries that use type classes to achieve this speed. Until then we will stop comparing our compiler to techniques that use type classes since type classes are beyond the capabilities of a baseline functional compiler.

Chapter 3 explores existing solutions to the problem of *expressing* generic functions. We highlight where our solutions fit into this picture and explain why they are the most appropriate for a baseline functional compiler.

Chapter 4 describes a baseline functional language compiler which occupies a space which lies at the intersection of existing functional languages such as Standard ML [2], OCaml, Haskell and Mercury [81]. This compiler is the seed from which we grow our final compiler capable of compiling generic functions. We describe this compiler in some significant detail because the algorithms we use for our new features are dependent on the basic algorithms they extend.

Chapter 5 describes our function extension operation which solves the problem of *polymorphic functions with specific behaviour* (see Section 3.3.1). We will have already described function extension in Chapter 3, so in this chapter we explain how to incorporate it into a Hindley-Milner type system and how to compile it to a simple run-time operation.

Chapter 6 describes application pattern matches which solve the problem of *structure agnosticism* (see Section 3.3.3). Again, the mechanism will have been described in Chapter 3, so in this chapter we are concerned with compiling and typing this feature. We will compile application pattern matches to simple primitive operations by carefully extending a standard pattern compilation algorithm. We will also describe type inference rules for these primitives, inspired by a solution used in the static pattern calculus [43].

Chapters 7 and 8 evaluate the final compiler and thus the techniques we used to create it. Chapter 7 demonstrates that we can write in DGEN all the generic functions described in Chapter 2 and explores the space around these particular examples to demonstrate the generality of the solutions we have described. We will show that a very large class of programs can be written using these techniques. Chapter 8 evaluates how successfully we have achieved the goal of *being compiled*. We demonstrate the speed and the memory use of the compiled code and the opportunities for optimising in the compiler.

Chapter 9 validates the type system of DGEN by proving the soundness of the underlying type relation with regard to the operational semantics of DGEN's core language.

Chapter 10 concludes this work with a discussion of future directions for this research and a summary of what we have achieved.

The compiler which implements the algorithms and techniques described in this thesis is available online at `dgen.science.mq.edu.au` where one can compile and execute code directly and download the compiler for running locally.

Chapter 2

Generic Functions

In this chapter we first introduce the language our compiler speaks, DGEN/SOURCE, and then encode each of our generic functions in this language. While this chapter contains no details of our technical contributions, it does introduce them. Furthermore, each of the generic *snippets* we introduce in this chapter is new in the sense that, while there are broadly equivalent encodings elsewhere, the style of programming generic functions which we show here is interesting in its own right.

In this thesis, *generic function* refers specifically to one of the functions defined in this chapter. Each of these functions is able to do its job on any data structure, in fact on any input at all. They are also able to do this without being enumerated for each type, or each type constructor, i.e. they are not overloaded. The term *generic* has a much wider meaning outside this thesis (as we discuss in Section 2.8), however in this thesis its meaning is restricted to these five types of functions.

For each of these functions we give a general description and then show an encoding of it in our concrete language DGEN. In this way we can introduce *both* the generic functions and our language.

The subject of this thesis is a *compiler*, not a programming language. The concrete language we present here (DGEN) is the simplest practical concrete language corresponding to an abstract language SOURCE. It exists only to allow one to experiment with the compiler and to describe algorithms/programs without resorting to abstract syntax.

2.1 Pattern Terminology

Patterns and pattern matching are very important in this thesis. To smooth the discussion we will describe, by example, some specific terms that we will use in this thesis. Consider

the following Haskell case expression

```
1  case a of
2     B b -> foo
3     C c1 c2 -> bar
```

The value on which the `case` expression branches (`a`) we call the *scrutinee*. The alternatives (`B b -> foo` and `C c1 c2 -> bar`) we call *branches*, *case branches*, or *case alternatives*. The above code has two case branches. The pattern (`B b` and `C c1 c2`) of a branch we will call the *left hand side (LHS) of the branch*, while the expression that is returned when that branch is chosen (`foo` and `bar`) we call the *right hand side (RHS) of the branch*.

2.2 DGEN

DGEN is the name we have given to our compiler implementation and is also the name of the concrete programming language corresponding to an abstract syntax, SOURCE, that sits at the front of our compiler. SOURCE would be the target of a desugaring phase in, for example, a Haskell or ML compiler based on our techniques, so DGEN looks like a desugared version of those languages. Rather than describe DGEN in detail, we will take as our starting point the intersection of ML and Haskell98 [49] and describe where DGEN differs from that hypothetical programming language. We will start with the abstract syntax of DGEN (which is *actually* SOURCE), to make clearer what we mean by "intersection of Haskell98 and ML" and to separate meaning from syntax. Where appropriate, we will use commonly-understood functional programs to demonstrate DGEN. Figure 2.1 gives our starting point, the abstract syntax of DGEN, aka SOURCE.

SOURCE has all the usual functional programming features. Data is described as algebraic datatypes (`adt`). Top-level function definitions bind an expression to a name. Anonymous functions are defined by lambda abstractions with normal function application for evaluating functions. There are `let` and `letrec` expressions for local function naming. SOURCE also has the usual literal values and an error term ($\varnothing$) which halts execution. Case expressions pull apart data and operate similarly to those in Haskell and ML, with the exception that each `case` expression matches a number of values against an equal number of patterns. A successful match only occurs if *all* values match their corresponding pattern. A pattern is either a variable, literal value, a constructor with patterns as arguments (nested patterns) or an *application pattern match* of two variables, which is one of our extensions for generic functions. You will also notice a primitive operation $\triangleright$ which is our *function extension operation*, also used for generic functions and not usually

Figure 2.1: Source, the abstract syntax of dgen

$$
\begin{array}{rcll}
s & ::= & \overline{sd} & \text{(Source program)} \\
sd & ::= & \texttt{adt}\ K\ \overline{x} = \overline{sk} \mid \texttt{def}\ x\ ot = se \mid \texttt{main} = se & \text{(top-level definition)} \\
sk & ::= & K\ \overline{(sk \mid x)} & \text{(constructor definition)} \\
se & ::= & x & \text{(expression)} \\
 & \mid & K\ \overline{se} & \text{(constructed value)} \\
 & \mid & se\ se' & \text{(application)} \\
 & \mid & \texttt{fun}\ x = se & \text{(anonymous function)} \\
 & \mid & \texttt{let}\ \overline{x\ ot = se}\ \texttt{in}\ se' & \text{(let binding)} \\
 & \mid & \texttt{letrec}\ \overline{x\ ot = se}\ \texttt{in}\ se' & \text{(recursive let binding)} \\
 & \mid & \texttt{case}\ \overline{x}\ \texttt{of}\ \overline{\overline{sp} \to se}\ \texttt{otherwise}\ se' & \text{(case expression)} \\
 & \mid & l & \text{(literal value)} \\
 & \mid & lo & \text{(built-in operation)} \\
 & \mid & \varnothing & \text{(error)} \\
sp & ::= & x \mid l \mid K\ \overline{sp} \mid x(x') & \text{(pattern)} \\
l & ::= & char & \text{(literal)} \\
 & \mid & number & \\
lo & ::= & se\ (+ \mid - \mid = \mid \neq \mid < \mid > \mid \leqslant \mid \geqslant)\ se' & \text{(built-in operation)} \\
 & \mid & se \triangleright se' & \text{(function extension)} \\
ot & ::= & \tau \mid \epsilon & \text{(optional type annotation)}
\end{array}
$$

Figure 2.2: Types in Source

$$
\begin{array}{rcll}
\tau & ::= & \alpha & \text{(type variable)} \\
 & \mid & T\ \overline{\tau} & \text{(parameterised type constructor)} \\
 & \mid & \text{Char} \mid \text{Integer} & \text{(literal types)}
\end{array}
$$

part of a functional language. We will discuss these mechanisms in more detail in Chapter 3. We see them in use in our encoding of the generic functions in this chapter, but a full description of how they work will be deferred until Chapter 3

Named functions can be annotated with a type (ot) which is used to guide type inference for features, such as polymorphic recursion, where types can't be inferred.

Figure 2.2 describes the language of types in Source. Types are either variables, named types where T is the type constructor (which can be parameterised by other types) or one of the literal types.

The concrete syntax (DGEN) is a very simple realisation of this abstract syntax. We will introduce the concrete syntax (and make clearer the semantics) by way of some familiar programming tasks. Listing 2.1 shows DGEN code for extracting the head of a list, including the definition of the list type.

Listing 2.1: head function on lists

```
1  adt list(a) = Nil() | Cons(a, list(a))
2
3  def head(lst) = case [lst] of
4                    { [Cons(x,xs)] -> x
5                    } otherwise    -> error "partial definition error in head"
```

The adt keyword defines a new datatype which is defined in the normal algebraic datatype fashion (as sums of products), while the def keyword defines a new function. Errors in DGEN carry a string to emit on the console upon the program halting. The abstract syntax in Figure 2.1 omits the string from errors for brevity.

Listing 2.2: merge sort (ported from [31])

```
1  #include "dgen_lib/std.dgen"
2
3  def cmp_less() = -1
4  def cmp_greater() = 1
5  def cmp_equal() = 0
6
7  def merge_cmp(cmp,y,z) = case [y,z] of
8           { [Nil(), y] -> y
9           ; [Cons(a,x),Nil()] -> Cons(a,x)
10          ; [Cons(a,x),Cons(aa,xx)] ->
11                if (cmp(a,aa) < cmp_equal | cmp(a,aa) i== cmp_equal)
12                then Cons(a,merge_cmp(cmp,x,Cons(aa,xx)))
13                else Cons(aa,merge_cmp(cmp,Cons(a,x),xx))
14          } otherwise -> error "partial definition error in merge_cmp"
15
16 def merge_sort(cmp,x) = letrec n() = length(x)
17                         and nn() = n / 2
18                         in  if (n < 1 | n i== 1)
19                                then x
20                                else merge_cmp( cmp
21                                              , merge_sort( cmp
22                                                          , take(nn,x)
23                                                          )
24                                              , merge_sort( cmp
25                                                          , drop(nn,x)
26                                                          )
27                                              )
```

Listing 2.2 shows a merge sort program which demonstrates more features. There are no modules, but a C-like preprocessor supports importing definitions from other files. Re-

cursive let bindings are introduced with the `letrec` keyword while `let` introduces non-recursive bindings. Such bindings are separated with the keyword `and`. All functions and constructors take their arguments in parentheses, separated by commas and empty argument definitions can be given with empty parentheses. Separate from programmer-defined functions are the built-in operations which can have infix syntax. SOURCE has different operations for each of string equality (`s==`), integer equality (`i==`), boolean equality (`b==`) and character equality (`c==`), a fact that is omitted from Figure 2.1 for brevity. Functions and constructors can be "curried" by giving only some of the arguments. The `case` expressions branch on a *list* of expressions (i.e. there is a list of scrutinees), rather than a single expression. There is thus a corresponding list of patterns in each branch of the `case` expression (hence the list notation we used in Listing 2.1, even when there was only one scrutinee and one pattern). This allows desugaring of the equational style of function definition and simplifies the definition of pattern compilation (Section 4.1). Each `case` expression has a distinguished and compulsory default branch, identified by the keyword `otherwise` which is the result if none of the branches match. Case branches are listed in curly braces separated by semi-colons. DGEN has list syntax using square braces and commas, which is the only desugaring done in the parser. This desugaring does not apply to `case` scrutinees and patterns where our abstract data type for `case` expressions receives the lists. DGEN passes parameters by value and has no side-effect-causing built-in operations[1], thus no side-effects at all.

Function application to arguments is done with standard parameter list syntax but DGEN also allows *constructor* application to arguments when the (partially applied) constructor is the result of some function. In this case the application uses function application syntax except that an @ annotation is added to the front of the application. For example, the following function, `structural_id`, will pull apart a value and then put it back together (giving an error for data that is not structured). It uses a feature we will talk more about later, *application pattern matches* (`x(y)`), for pulling data apart.

```
1 def structural_id(a) = case [a] of
2                            { [x(y)] -> @x(y)
3                            } otherwise -> error "err"
```

Notice that we need to annotate the code for putting the data back together with @

[1] Output to the console is supported, and is strictly a side-effect but since there is no mechanism for those actions to feed back into the program, we don't have any great need to think about side-effects for the purposes of this work. Furthermore, DGEN uses eager evaluation which makes is simple to reason about the order of console output operations.

to indicate this is a data application, not a function application. This is necessary only for our parser, internally all applications are the same. When we want to apply (as data) the result of a function to its remaining arguments, the parser is sometimes not able to identify this as legal (in fact, in this particular case it could, but this is a "toy" example). It would require us to first bind the result to a variable. We wanted to avoid this extra binding for reasons of clarity, so we introduced the @ annotation as a clue to the parser. In practice we find it a useful way to document the code for data applications and we use it in that way throughout the thesis[2].

Anonymous functions (lambdas) are introduced with the `fun` keyword.

```
1 def id = fun(x) = x
```

Finally, there is a distinguished function `main` which defines the function to run on program execution[3]. We will now introduce the five kinds of generic functions which we will use as examples throughout this thesis.

2.3 Generic Update

A generic update function is one that can traverse any data value and perform a type-preserving transformation at specific nodes in the structure. It can be considered a type-preserving map capable of operating on any structure. Both the structure and the operation are parameters to the generic update function. Throughout this thesis we use the *salary update snippet* to demonstrate this type of function.

The salary update snippet (shown in Listing 2.3, which is a translation of an example from [60]) defines an algebraic datatype for company structures, and code for updating the salary of all employees of the company. Both the particular company to work over, and the updating function are arguments to the generic update function, `generic_update`.

The `generic_update` function actually defers its job to one of our generic traversal functions, `apply_to_all`, discussion of which we defer to Section 2.5. If we assume that `apply_to_all` takes a function and applies it to each part of the input data, our job is to construct a function which knows how to process each part of the input data. Since `apply_to_all` has the type $(\forall a.a \to a) \to b \to b$, this function must have type $\forall a.a \to a$. Typically, simple polymorphic languages only allow one function with that type, the

[2] At this point we would like to remind the reader that DGEN's concrete syntax is a means to an end, and effort that could have been put into improving it has gone into other work instead. The focus of this thesis is the abstract syntax SOURCE and the compiler that processes it.

[3] When we want to demonstrate use of a library module, which can't have a `main` definition, we will call it `main_func`.

Listing 2.3: The salary update snippet

```
adt company()   = C(list(dept()))
adt dept()      = D(string, manager(), list(sub_unit()))
adt sub_unit()  = PU(employee())
                | DU(dept())
adt employee()  = E(person(), salary())
adt person()    = P(string, string)
adt manager()   = M(employee())
adt salary()    = S(int)

// setup
def gen_com() = let ralf()   = E(P(”Ralf”, ”Amsterdam”), S(8000))
                and joost()  = E(P(”Joost”, ”Amsterdam”), S(1000))
                and marlow() = E(P(”Marlow”, ”Cambridge”), S(2000))
                and blair()  = E(P(”Blair”, ”London”), S(100000))
                in  C([ D(”Research”, M(ralf), [PU(joost), PU(marlow)])
                      , D(”Strategy”, M(blair), [])
                      ]
                     )

// logic
def incS(amt, s) = case [s] of
                        { [S(s)]    -> S(s + amt)
                        } otherwise -> error ”partial definition error in incS”
def id(x) = x
def increment(amt) = incS(amt) ▷ id
def generic_update(func, dat) :: (∀ a . (a) -> a, b) -> b = apply_to_all(func,dat)

//do it
def main_func() = generic_update(increment(237),gen_com)

// > output is:
// > Company[ Department: Research, Ralf<Amsterdam, 8237>,
// >                                [ Joost<Amsterdam, 1237>
// >                                , Marlow<Cambridge, 2237>
// >                                ]
// >     and  Department: Strategy, Blair<London, 100237>, []
// >     ]
```

identity function (`fun(x)= x`). We use a *function extension* operator (▷) to add specific behaviour to the identify function. This is one of our extensions to the baseline functional compiler and its semantics will be clarified in Chapter 3. The resulting function will give the specific behaviour on inputs of the specific type, while still deferring to the identity for all other types. Thus, `increment` is a function which will increment a salary if applied to one and do nothing to its input otherwise. It is this function that we set to be applied to all nodes in the data structure in question. Upon running `generic_update` with `increment(237)` and the example company structure, the result is the same company structure with every salary increased by 237. Note that DGEN type annotations are all given in an“uncurried” style but the functions can be curried. It is for ease of parsing that the uncurried style is used in the concrete syntax.

2.4 Generic Query

A generic query function is able to traverse any data structure, accumulating a single value as its result. It can be considered a fold that can operate on any data structure. Both the data and the accumulation operation are parameters to the function, but the same mechanism can be used to define functions with a set accumulator. Throughout this thesis we use the *name analysis snippet* (Figure 2.4) to demonstrate this type of function.

The name analysis snippet includes the definition of an abstract syntax tree datatype (adapted from a simple imperative language by Reynolds [76]), and code to check that every *use* of a name is preceded by the *definition* of that name. The `check_it` function is the accumulating operation, and `a_correct_command` gives some data to test the function on.

We assume the presence of a `generic_query` function, whose definition we give in Section 2.5. It is like a fold over any data structure. Its type signature is $(\forall a.r \rightarrow a \rightarrow r) \rightarrow r \rightarrow b \rightarrow r$ and operates thus.

> This function takes an accumulating (or *folding*) operation and a starting value as its first two parameters. The third parameter is the data to work over and the result is the final result of applying the accumulating function at every node in the input value.

With this function present, we can define an actual generic update example by defining the accumulating function. As in generic update, this function needs a very polymorphic type since it is applied at every node, and again we build it up with the function extension mechanism. We define separate functions for each *interesting* data type (i.e for `comm` and `int_exp`, but not for `bool_exp` since it plays no part in the computation) and combine them with a generic failover[4] case (`fun(a)= strbool` on line 36). The resulting function, `check_it` will use `check_comm` when it encounters a command node, `check_intexp` when it encounters an integer expression and the failover case for all other nodes.

2.5 Generic Traversal

The generic query and generic update functions above have assumed *some* way to visit all the values in a structure. Generic traversal functions are *programmer-defined* functions that

[4]We like the term "failover" for this function since it is a function that protects us from unsafe generic traversal by stepping in when all other options fail.

Listing 2.4: The name analysis snippet

```
// -- Abstract Syntax --                          -- Example Concrete Syntax--
adt comm() = CAssign(string, int_exp())          // ident := exp
           | CDecl(string, int_exp(), comm())    // let ident := exp in comm
           | CSkip()                             //
           | CSeq(comm(), comm())                // comm1 ; comm2
           | CWhile(bool_exp(), comm())          // while (bool) { comm }
           | CPut(int_exp())                     // printf(exp)

adt int_exp() = IUse(string)                     // ident;
              | ILit(int)                        // 5
              | IPlus(int_exp(), int_exp())      // 5 + ident

adt bool_exp() = BTrue()                         // true
               | BFalse()                        // false
               | BEq(int_exp(), int_exp())       // 5 == ident
               | BNEq(int_exp(), int_exp())      // 5 != ident
               | BAnd(bool_exp(), bool_exp())    // true && ident
               | BOr(bool_exp(), bool_exp())     // true || ident

def check_comm(strbool,comm)
  :: (pair(list(string),bool),comm()) -> pair(list(string),bool)
  = case [strbool,comm] of
         { [Pair(lst,b), CAssign(s, ie)] -> Pair(lst,elem(fun(p,q) = p s== q, s,lst) & b)
         ; [Pair(lst,b), CDecl(s,ie,c)]  -> Pair(Cons(s,lst), b)
         ; [Pair(lst,b), z]              -> strbool
         } otherwise        -> error "partial definition error in check_comm"

def check_intexp(strbool,comm)
  :: (pair(list(string),bool),int_exp()) -> pair(list(string),bool)
  = case [strbool,comm] of
         { [Pair(lst,b), IUse(s)] -> Pair(lst,elem(fun(p,q) = p s== q, s,lst) & b)
         ; [Pair(lst,b), z]       -> strbool
         } otherwise              -> error "partial definition error in idbu"

def check_it(strbool)  :: (pair(list(string),bool), a) -> pair(list(string),bool)
     = check_comm(strbool) ▷ check_intexp(strbool) ▷ fun(a) = strbool

def decl_before_use(comm) = snd(generic_query(check_it,Pair([],true),comm))

def a_correct_command() =
  CDecl( "v"
       , ILit(1)
       , CWhile( BNEq(IUse("v"), ILit(3))
               , CSeq( CAssign("v",  IPlus(IUse("v"),ILit(1)))
                     , CPut(IUse("v"))
                     )))
```

do this visiting and which have been crafted to *traverse the structure in whatever way suits the problem at hand*.

To demonstrate generic traversal we will encode two traversal strategies for both queries and updates; top-down and bottom-up. The same techniques used to define these two examples can be used to define myriad other traversal strategies.

It is possible to define a set of generic traversal operations up-front and to require your

programmer to use one of these for all generic updates and generic queries. This can work quite well in practice, but is not what we mean when we say "generic traversal". When we use this term we are referring to programmer-defined generic traversals, and the ability for the programmer to create custom traversals that suit their purpose. For example, our name analysis snippet can only work if the `generic_query` operation traverses top-down, applying the accumulating function to nodes higher in the value before passing the result to those lower in the value. This is not a universal requirement and it may be necessary at some time to have a similar function that works some other way. With generic traversal, the programmer is free to create whatever traversal they need.

2.5.1 Bottom-Up Update

The `apply_to_all` function, as used in the salary update snippet, is actually a bottom-up traversal; its full definition in DGEN is shown in Listing 2.5

Listing 2.5: A bottom-up generic traversal

```
1 def apply_to_all(f,g) :: (∀ a . (a) -> a, b) -> b =
2     case [g] of
3          { [c(a)]    -> f(@apply_to_all(f,c)(apply_to_all(f,a)))
4          ; [o]       -> f(o)
5          } otherwise -> error "partial definition error in apply_to_all"
```

The `apply_to_all` function has two arguments, a function to apply at every node (`f`), and a value to traverse (`g`). It works by inspecting its argument (`case [g]`) and branching based on whether it is a *compound* (`c(a)`) or a *atom* (`o`). These are the application pattern matches we saw in Section 2.2. These are one of our extensions to the baseline functional compiler and their semantics will be clarified in Chapter 3. This relies on the ability to see all values in the language as binary trees: compounds are the internal nodes and atoms are the leaves. We describe how to do this in Section 3.3.3, but for now we must take as an assumption that it works.

With this in place, we can apply `f` to all nodes by applying it to the current node and recursively applying `apply_to_all` to each child (`c` is the left child and `a` is the right). We then stitch the two transformed halves back together with the @ annotation, which helps the parser understand curried application. Recall that we will always use @ to annotate data application even though there is no difference in the abstract syntax.

The `apply_to_all` function works in a bottom-up manner, which is encoded by applying the function `f` to the result of stitching together the transformed children. We show

how to encode a top-down traversal in the next subsection.

2.5.2 Top-Down Update

It is quite straightforward to write a version of `generic_update` which processes its second argument from the top-down instead of from the bottom up. Listing 2.6 shows just such a function.

Listing 2.6: Top-down update

```
1  def apply_to_all_td(f,g) :: (∀ a . (a) -> a, b) -> b =
2      let fg() = f(g)
3      in  case [fg] of
4              { [c(a)]    -> @apply_to_all_td(f,c)(apply_to_all_td(f,a))
5              ; [o]       -> o
6              } otherwise -> error "partial definition error in apply_to_all_td"
```

The general mechanism is the same but we first call the transformation function (`f`) on the current node and then recursively call `apply_to_all_td` on the result of that function call. We must be careful *not* to re-apply `f` when the result of calling it on the current node is an atom (line 5).

2.5.3 Top-Down Query

The `generic_query` function — used in Listing 2.4 and defined in Listing 2.7 — is actually a *top-down* generic query function. The accumulation function will first be applied to the node being inspected, then the result is threaded to the right-hand argument, finally to the left-hand argument. So in fact, it is top-down, right-to-left. Top-down left-to-right could also be easily encoded but it would require a little extra work since using application pattern matches exposes children right-to-left.

Listing 2.7: A top-down (right-to-left) generic query

```
1  def generic_query(f,start,dat) :: (∀ a . (r,a) -> r, r, b) -> r =
2      case [dat] of
3           { [c(z)]    -> generic_query(f,generic_query(f,f(start,dat),z),c)
4           ; [o]       -> f(start,o)
5           } otherwise -> error "partial definition error in generic_query"
```

As with `apply_to_all`, we rely on the ability to see any value as a tree, meaning our job is to apply the accumulator at the correct places and to thread the output of the accumulator to the recursive calls in the correct order. We first apply the accumulator to

the whole value at the current node (`f(start, dat)`), then use the result of this as the start value for a recursive call on the right sub-tree, finally passing that result to the start parameter of a recursive call for the left sub-tree.

2.5.4 Bottom-Up Query

We can easily create a bottom-up version of `generic_query` by changing the places at which we call the accumulator function. Listing 2.8 shows such a function. Instead of first calling the accumulator on the current node, we defer that job until the left and right subtrees are processed, passing the final result of the right subtree as the start value for the current node.

Listing 2.8: A bottom-up (left-to-right) generic query

```
1 def generic_query_bu(f,start,dat) :: (∀ a . (r,a) -> r, r, b) -> r =
2     case [dat] of
3         { [c(z)]    -> f(generic_query_bu(f,generic_query_bu(f,start,z),c),dat)
4         ; [o]       -> f(start,o)
5         } otherwise -> error "partial definition error in generic_query_bu"
```

2.6 Generic Equality

Generic equality is a single function which can determine the equality of two values of any type. The `geq` function in Listing 2.9 is a generic equality function.

First, we must define a sequence of generic versions of the built-in equality functions, each able to deal with one addition built-in type. The first of these, `g_str_equals`[5], can deal with either two strings (by the left argument to the function extension of line 6, or two constructors (by the right argument). It is built up in two steps, an outer instance of function extension (line 6) creates a function that calls another instance of function extension (line 3) if the first argument is a string. The "inner" extended function then calls string equality if the second argument is also a string, otherwise it returns `false`. The outer extended function defers to constructor equality if the first argument is not a string. This *constructor* equality (===) is the secret to writing this function. By pulling apart data as we did in generic traversal, constructors can be exposed without their arguments. If we

[5] `g_` and `generic_` are prefixes we use in function names to indicate a function is structure agnostic. Sometimes we use `g_` to keep the identifier short (as we have done here), other times it is to match existing functions we are emulating (`geq` is the identifier used in bondi for generic equality for example). Where there is no particular reason to use `g_` we prefer the more precise `generic_`.

Listing 2.9: Generic Equality

```
def g_str_eq() :: (a,b) -> Bool =
      (fun(a) = (fun(b) = a s== b)
                 ▷
                 (fun(b) = false)
      )
      ▷ (fun(a) = (fun(b) = a === b))

def g_int_eq() :: (a,b) -> Bool =
      (fun(a) = (fun(b) = a i== b)
                 ▷
                 (fun(b) = false)
      )
      ▷ g_str_eq()

def g_char_eq() :: (a,b) -> Bool =
      (fun(a) = (fun(b) = a c== b)
                 ▷
                 (fun(b) = false)
      )
      ▷ g_int_eq()

def g_bool_eq() :: (a,b) -> Bool =
      (fun(a) = (fun(b) = a b== b)
                 ▷
                 (fun(b) = false)
      )
      ▷ g_char_eq()

def bi_eq(x,y) :: (a,a) -> Bool
            = g_bool_eq(x,y)

def geq(a,b) :: (a,b) -> bool = case [a,b] of
                                     { [c1(a1),c2(a2)] -> geq(c1,c2) & geq(a1,a2)
                                     ; [c1(a2),z2]     -> false
                                     ; [z1,c2(a2)]     -> false
                                     ; [z1,z2]         -> bi_eq(z1, z2)
                                     } otherwise       -> error "partial definition error in
                                         geq"

def main_func() = geq([One(),Zero()],[One(),One()])
```

add some basic built-in functions that work on these “lonely” constructors (constructors with none of their arguments attached), we can write functions like equality and the next, show. In this process, the constructor equality function is used as the right hand argument to an extension operator, which means it must have type `(a,b)-> bool`. We then build this up through the next three definitions until we have a function with extensions for all built-in types plus constructors. This function (`bi_eq`) is used to check the equality of atoms, and all other equality checks are done by the main `geq` function. `geq` inspects its two arguments and, if they are the same structure, either recurses into the branches for compound inputs or calls the atomic equality for atomic inputs. It gives `false` for inputs of different structure. The generic equality in Listing 2.9 is longer than it should

be because of deficiencies in the DGEN parser. If we bypass the parser and use SOURCE we can define all the `g_*_eq` functions within `bi_eq`.

2.7 Generic Show

Generic show is a single function which can encode, as a string, any value of any type. Listing 2.10 is a code snippet demonstrating this function.

Listing 2.10: Generic Show

```
1  def gshow(a) :: (a) -> String
2              = case [a] of
3                    { [c(p)] -> gshow(c) ++ "(" ++ gshow(p) ++ ")"
4                    ; [z]    -> bishow(z)
5                    } otherwise -> error "partial definition error in gshow"
6
7  def bishow() :: (a) -> String
8              = let si(x) = show_int(x)
9                and sc(x) = show_char(x)
10               and sb(x) = show_bool(x)
11               and ss(x) = if (x s== "") then x else x
12               and ds(x) = show_constr(x)
13               in  si ▷ sc ▷ sb ▷ ss ▷ ds
```

We use exactly the same mechanism in `gshow` as we did in `geq`. The main difference is that we need a *different* built-in function working on lonely constructors. In this case we use `show_constr` which will convert any lonely constructor to a string representation, and will give a run-time error otherwise. Thus, like ===, it has a very polymorphic type (`(a) -> String`), which means it can be used in the right hand side of the extension operator (▷). Note that DGEN does not support type annotations that could be used to indicate `ss (x)` should be `(String)-> String`. Thus we have had to pad the encoding with an `if` expression to enforce the desired type. Because we are describing a baseline functional compiler with additions that are orthogonal to other functional language features, there is no impediment to implementing more general type annotations. However, this is beyond the focus of this work, hence the work-around for `ss(x)`. Furthermore, the simple parser in DGEN has forced us to bind each built-in function to a let-bound variable since the parser expects functions (not built-ins) as arguments to ▷. We can write the same code in the abstract syntax without the let-bindings for `si`, `sc`, etc.

Note that these are not the only kinds of generic functions that DGEN can compile. They are a minimal set of abilities that a compiler should have to call itself "generic" in the sense we use it here. In Chapter 7 we explore the breadth of generic code that can be compiled with DGEN.

2.8 Other "Generic" Functions

"Generic" is a heavily overloaded term in computing, in this section we further clarify how we use it in this thesis by explaining what it is *not* in this context.

2.8.1 Object Oriented Generics

The generic functions of object oriented (OO) languages like Java and C# bear no relation to the generic functions we have listed above. This thesis starts from the assumption that the parametric polymorphism that these mechanisms implement is already available as a starting point. The functions above are all from a higher-level form of generics.

2.8.2 Term Rewriting

Term rewriting libraries for functional programming languages are sometimes referred to as generics. Generic update and generic query with generic traversal gives us term rewriting ability. So our use of generics is compatible with that used in these libraries. We show in Section 7.2.7 how to emulate term rewriting in DGEN.

2.8.3 Datatype Generics

Datatype generics is a more specifically defined term, characterised by a set of canonical examples. All five of our functions are included in this definition. There are also other functions in this set, such as generic map and generic read, which we do not include because the "spine view" (see Section 3.3.3) of data exposed with application pattern matches does not immediately support them. We discuss these functions and how we would go about supporting them in Section 7.3.

2.9 Summary

In this chapter we have made precise the term "generic" in the context of this thesis; it refers to the language/compiler features require to implement these eight program snippets:

1. Salary Update

2. Name Analysis

3. Apply to All

4. Top-Down Apply to All
5. Generic Query
6. Bottom-Up Generic Query
7. Generic Equality
8. Generic Show

In Chapter 7 we will look at extensions of this set but until then we refer only to these features. We also saw a baseline functional programming language – with just *two* non-standard features; application pattern matches and function extension – that was able to describe all of these generic examples. The remainder of this thesis is concerned with compiling this language, in particular, the two non-standard features. In this chapter we have only given a hint of how these features work. In the next chapter we explore the specific problems they solve and how they do it, greatly expanding our understanding of how they work. In chapters 5 and 6 we will complete the picture by describing how to implement them.

Chapter 3

Encoding Generic Functions

In this chapter we will survey existing techniques for *expressing* generic functions. We will highlight those that are currently used in compilers and explain why those techniques are not appropriate for a *baseline* functional compiler. We will also evaluate the other techniques for their applicability to our task and describe the novel techniques we use. This chapter will make clear why none of the existing techniques are appropriate without some modification and explain where our new techniques fit in relation to these existing ones.

Most of the existing tools for processing generic functions fit into one of three broad *computational styles*:

- term rewriting,
- pattern calculii, and
- datatype generics.

We begin with a brief introduction to each of these in Section 3.2 before describing in more detail the two particular capabilities required to support generic functions:

- *polymorphic functions with specific behaviour* (Section 3.3.1) and
- *structure agnosticism* (Section 3.3.3).

Every tool capable of processing generic functions will have a solution for both of these capabilities. It is the different solutions to each that we are primarily concerned with in this chapter. We will show that none of the existing solutions suffices for our purpose and we will describe our techniques as variants/extensions of existing techniques. We split our survey according to these two capabilities, first discussing polymorphic functions with specific behaviour, then discussing structure agnosticism.

As solutions to polymorphic functions with specific behaviour, we will discuss:

- function extension,
- extension types,
- by-passing static type checks,
- universal representation of data,
- type manipulation, and
- explicit failure.

We evaluate each of these as a solution for a baseline functional compiler and then describe the novel technique we use in DGEN, *function extension with a single operator* (see Section 3.3.2). This technique is a new variant of both function extension and extension types.

As solutions to structure agnosticism, we discuss:

- full universal representation of data,
- pre-defined generic traversals, and
- object-based reflection.

We evaluate each of these as a solution for a baseline functional compiler and then describe the technique used in DGEN, *explicit spine view via application pattern matches* (Section 3.3.4). This approach to structure agnosticism has been used before in interpreters, but it has never been compiled. Our contribution is the compilation of this feature described in Chapter 6.

3.1 Evaluation Criteria

If we are to evaluate each existing solution, we must decide what constitutes a desirable set of characteristics. In this section we outline the characteristics which we require of a technique to be appropriate for use in a baseline functional compiler.

No changes to the semantics of the original language (abbreviated to "semantic") We are describing a way to turn any functional language compiler into one capable of compiling generic functions. If changes to the semantics of the original language were required it would not be possible to describe these techniques as *extensions*. Their

incorporation into a compiler would potentially render invalid all existing code for that compiler.

No dilution of the static type system (abbreviated to "type") Our starting point is statically typed compilers with a Hindley-Milner type system and thus the audience is developers who value static typing. We cannot admit any technique that requires a relaxation of the static type safety of the language.

Minimal changes to the compiler (abbreviated to "changes") The smaller the number of changes, and the easier they are to implement, the more likely they will be incorporated into existing compilers. A compiler is already a very complicated artefact, thus techniques that are difficult to implement in isolation could be prohibitively difficult to add to an existing compiler. Furthermore, minimal changes are easier to verify as correct.

No use of advanced language features (abbreviated to "baseline") As we have already discussed, a solution that relies on a particular language feature like lazy evaluation, type classes or advanced module systems, will not be simple to implement in a language without those features.

We don't have a separate criteria relating to the expressiveness of the tool. However, only tools capable of encoding all the generic snippets in Chapter 2 are included in this survey.

3.2 Approaches

Before looking in detail at each technique, we give a general overview of the three computation styles in which generic functions are commonly expressed.

3.2.1 Term Rewriting

Term rewriting is a complete model of computation where results are achieved by successive translations of the input. Input is treated as a tree of data that is traversed, applying translations at certain nodes. This is particularly suited to tasks like program compilation, but any data can be treated as a tree of nodes.

There are broadly two categories of term rewriting systems. Those that rely on predefined traversal algorithms, and those that allow the programmer to create their own traversal strategies from smaller, more primitive strategies. In this work, "term rewriting"

refers to both types, "strategic rewriting" refers to the later, and if we need to refer to the former exclusively, we will call it "pure term rewriting".

Term rewriting systems tend to have less-strict type systems than typical statically types programming languages, although there are exceptions to this (such as Asf+Sdf [86]). For example, Kiama [80] is a term rewriting library embedded in the statically typed functional/OO language Scala but it uses very general types during re-writing. All values on which rewriting occurs are treated as the same type (`Term`) meaning that many of the assurances of the static type system are lost.

3.2.2 Generic Calculii

Under this umbrella we include *the pattern calculus* [46], *Faure's patched pattern calculus* [28], *the ρ-calculus* [15, 16], *the lambda calculus with constructors* [4], *bondi* [29, 44] and *RhoStratego* [22]. The first four are extensions to the lambda calculus which attempt to capture: the role of data (the pattern calculus and the lambda calculus with constructors), and the semantics of rewriting (the ρ-calculus and Faure's patched pattern calculus). Interestingly, the two concerns have converged on very similar calculii, so we restrict our discussion to the pattern calculus when considering details. RhoStratego is a programming language built originally as an implementation of the ρ-calculus, but eventually relaxing its adherence to that calculus. It is the only example we know of a compiled and typed functional language which embraces failure as a primitive. The bondi language is an interpreted implementation of the pattern calculus.

All these systems take the simplest, most internally consistent approach possible and thus are a good source of desugaring techniques and compilation ideas. However, the calculii are fairly relaxed about what can be done at run-time, something a compiler must be more careful of. RhoStratego, while more concerned with such things, does not compile patterns, which is a major impediment to raw speed and optimisation possibilities.

3.2.3 Datatype Generics

Datatype generics make most sense in the context of statically typed, polymorphic and functional programming languages like Haskell, ML and Clean, although the ideas actually apply in a somewhat broader context than that [73]. Languages like Haskell and ML have advanced type systems that admit *parametric polymorphism*. Datatype generic tools introduce another level of polymorphism above that, also referred to as *polytypic* or *type indexed* functions. This extra level of polymorphism is enough to encode our generic functions.

3.2.4 Others

There are many other systems which can encode our generic functions but which are too far removed from our *compiled* and *strongly typed* constraints. However, we mention them where the comparison is particularly relevant. Untyped functional languages like Lisp and Scheme allow encoding of generic functions with similar techniques, but devoid of static types. Particular DSLs like XQuery [91] can elegantly encode, again without the kind of type system we are talking about, some of the generic functions. However, these DSLs are not general purpose tools. We pay particular attention to how you could encode generic functions in statically typed, object oriented languages since many people will have tried (perhaps successfully) to do just that, and the comparison is enlightening.

3.3 Capabilities

We break the generic functions down into two capabilities that are required to implement them. We got a sneak-peek of these in the previous chapter when we saw DGEN snippets implementing generic functions; now we will look at them in detail.

3.3.1 Polymorphic Functions with Specific Behaviour

The salary update snippet requires that a single function be applied to particular parts of a composite value. Those parts are identified by type. We want to take in any possible structured value and apply an increment at all *salaries* within it. We will work under the assumption that there are already ways to traverse structured values, Section 3.3.3 goes into detail on how to achieve this. There are broadly two ways to approach polymorphic functions with specific behaviour:

Parameterise the traversal by type *and* function: You pass to the traversal 1) the type at which to apply 2) the function to apply, and leave it to do the rest of the work.

Apply a single function to *every* part of the tree: Such a function must do nothing unless it is being applied to a value of the right type (in this case salary). We need to write functions that can be applied to any value (i.e. have type $\forall a.a \rightarrow X$), but which do something at specific types (i.e. are *not* just constant functions).

We refer to both possible solutions as *polymorphic functions with specific behaviour*.

Function Extension

Function extension takes an existing function with the required polymorphic type and *extends it* with a more specific function. This creates a new function with the required polymorphic type, but which uses the specific function whenever it is applied to an appropriate type. You can either think of it as *specialising* the polymorphic function, or as *attaching a fail-safe* to the specific function. For example, the salary update snippet needs a function which increments a salary and does nothing for all other types.

This approach is used in SYB where there are a number of functions provided which can specialise a polymorphic function with a specific one. For example, `mkT` takes one function as an argument and uses it to specialise the polymorphic identity function. `mkQ` takes a function and uses it to specialise a polymorphic constant function (specified with its second parameter). For example, Listing 3.1 shows how to define a function with SYB that increments salaries and leaves all other values untouched (`incrementS`), and how to define a function which extracts integers from salaries while giving 0 when applied to something that is not a salary (`extractS`). We refer to this examples as *fixed* function extension since the functions which are specialised are only partly under the programmer's control.

Listing 3.1: Fixed function extension in SYB

```
1 incrementS = mkT (\S(i) -> S(i + 1))
2 extractS = mkQ 0 (\S(i) -> i)
```

However, `mkT` and `mkQ` are often too restrictive, we would like to define what polymorphic function to use for `mkT` and to define the constant as a function for `mkQ`. Thus SYB also defines `extT` and `extQ` which allow for this extra flexibility. Listing 3.2 shows how to define `incrementS` and `extractS` using these functions. We refer to this version as *flexible* function extension since the programmer defines all parts of the extension.

Listing 3.2: Flexible function extension in SYB

```
1 incrementS = extT (\a -> a) (\S(i) -> S(i + 1))
2 extractS = mkQ (\a -> 0) (\S(i) -> i)
```

The `extT` and `extQ` functions are not directly encoded in SYB. They are realised via a *type-safe cast* mechanism. This mechanism requires some kind of type parameter, telling it what type we wish to cast to, and a value to cast. It will return a boxed value (for example,

Maybe in Haskell, Option in Scala) which is empty if the cast failed and full with the cast value if the cast succeeds.

Achieving the type-safe cast is a problem in itself, but with type classes (as in Haskell) it can be achieved. SYB does this (shown in Listing 3.3), using the implicit type parameter that type-classes generate to direct the cast, and the overloading provided by type-classes to give a definition for each type.

Listing 3.3: increment encoded in with type-safe cast

```
mkT :: (Typeable a, Typeable b) => (b -> b)  -> a -> a
mkT f = case cast f of
                 Just g -> g
                 Nothing -> id

incS :: Int -> Salary -> Salary
incS k (Salary j) = Salary (j + k)

increment :: Int -> a -> a
increment k = mkT (incS k)
```

Flexible function extension (extT and extQ) does many of the things we require (as per the evaluation criteria in Section 3.1) but it won't be our final solution. The use of type-safe cast with fail-over means that there is no loss in type safety. SYB required only minimal changes to *GHC* and only small and commonly available extensions to the semantics of Haskell (such as higher ranked types). However this was done with a great deal of help from the type-class mechanism built into Haskell. A baseline functional compiler does not have type classes. Thus the SYB implementation of polymorphic functions with specific behaviour can not be used verbatim in DGEN. However, we will show that something very similar can be achieved without type-classes at all if we are willing to try a few other compiler extensions. These are simpler than type-classes and by moving expensive operations like type checks out of a library and into the compiler we will get much lower overhead for generic programs (see Section 8.2).

Another possible improvement from the point of view of a compiler writer is the above use of *two* function extension operations[1]. We will see in the next section a technique that requires only one function extension mechanism. If we could use one operation rather than two, we would not only reduce the implementation burden, we could get closer to emulating the technique we are going to see in the next section, extension typing, although that remains future work.

[1]We showed four, but mkT and mkQ can be written using extT and extQ, so you only need two.

Extension Typing

The SYB primitives `extT` and `extQ` extend one function with another based on *types*. The pattern calculus, and bondi, use a mechanism which extends functions based on *constructors*. This mechanism is called *extension* in the pattern calculus literature ([43], Chapter 4), but we call it *extension typing* because it requires particular typing rules. An extension in the pattern calculus is a triple, `p`, `s` and `r` written `p -> s | r`. This defines a computation which tries the pattern `p`. Upon success it returns expression `s`, with appropriate bindings. Upon failure to pattern match, it returns expression `r`.

The two main differences between extension typing and function extension are: extension typing is a single operation which can do the job of both `extT` and `extQ`, and extension typing mechanism is *constructor* focussed rather than *type* focussed. For example, Listing 3.4 shows how to encode the `increment` function in bondi and demonstrates this focus on constructors.

Listing 3.4: `increment` encoded in bondi

```
1  let (increment : Int -> a -> a) i =
2       S s  -> S (a + i)
3     | x    -> x
```

The `x -> x` case alternative is *not* capturing left-over salary constructors (as it would in a Hindley-Milner system); it is capturing *all* possible constructors, including those from other types. The type signature of `increment` captures this, it has a type variable (implicitly universally quantified) as its second argument.

This change in the semantics of pattern matching makes extension typing inappropriate for our needs. Furthermore, the type system required for full extension typing is a significant departure from standard Hindley-Milner. However, we are attracted by the single operator for all function extension and thus we use for DGEN a mechanism that is part-way between function extension and extension typing (see Section 3.3.2).

By-pass Static Type Checks

Listing 3.5 shows a possible encoding of the `increment` function in a (Java-like) statically typed object oriented language. It requires *manipulation of types* and *type cast*, both of which are mechanisms which allow the programmer by-pass the static type checking of the language.

Listing 3.5: `increment` encoded in a Java-like language

```
1  Object increment(Object x){
2     if (x.isIntanceOf().className("Integer"){
3        return ((Integer)x + 1)
4      } else {
5       return x
6  }
```

If the programmer always witnesses a cast with an `isInstanceOf`, then we won't get any problems, but this is *not* enforced by the type system. Most static OO languages allow encoding in a similar fashion.

Dynamic[2] OO languages can simply do without the type cast, but don't have static types at all, so are even further from our desired compiler.

Universal Representation of Data

Some systems use a value-level representation of types to get polymorphic functions with specific behaviour. LIGD [14], for example, gives a `Rep` α value for every type α, which is the *representation* of that type. This value must be passed as an extra parameter to functions that have specific behaviour at a specific type. Listing 3.6 shows how the increment function is encoded in LIDG [77]. The extra parameter which contains all the type information is the second (`Rep a`) one.

Listing 3.6: `increment` encoded in LIDG [77]

```
1   increase :: Float -> Rep a -> a -> a
2   increase f (RSum a b   ep)  t  =  case from ep t of
3                                      Inl x ->  to ep (Inl (increase f a x))
4                                      Inr y ->  to ep (Inr (increase f b y))
5   increase f (RPair a b  ep)  t  =  case from ep t of
6                                       x :*: y -> to ep (increase f a x :*: increase f b y)
7   increase f (RType e a  ep)  t  =  to ep (increase f a (from ep t))
8   increase f (RCon "S" a)     t  =  case a of
9                                           RFloat ep -> to ep (incS f (from ep t))
10  increase f (RCon s a)       t  =  increase f a t
11  increase f _                t  =  t
12
13  incS :: Integer -> Integer -> Integer
14  incS f s = (s + f)
```

Notice that a great deal of information about the type of the parameter is available and we must explicitly recurse through the value looking for the constructor that witnesses the type we are interested in. In `increase` we are looking for `S` for salary, which witnesses

[2] Where "dynamic" means either duck-typed or run-time typed. I.e., languages like SmallTalk, Ruby, Python, Objective-C.

the integer values in the system. This function is much more complicated than others in this comparison because it is doing both the traversal *and* the incrementing, which are separated in other systems. The fact that passing in a full type representation allows both specific/polymorphic functions *and* traversal means we will discuss it again in Section 3.3.3

Bringing types into the programming model and allowing functions to depend on them is a significant change to the semantics of standard functional languages, but introducing a value-level equivalent is not. It changes the programming style, but needs no new language features. However, achieving this value-level representation does require either type classes, or GADTs, or both. These are not features of our baseline functional compiler.

Implicit Type Manipulation

With type classes, one can make the explicit type parameter, that has been reflected from the type-level to value-level in Listing 3.6, an implicit one. Examples of this approach are EMGM [37, 10], `regular` [68] and Instant Generics [13]. This saves you from having to create the representation in the first place and can result in some very efficient generic programs [68]. However, it is completely dependent on type classes since it is this mechanism that passes around the implicit type parameter. Thus it requires advanced language features not present in a baseline functional compiler.

Explicit failure

We can do entirely without type representations and casts if we admit failure as a first-class citizen of the language. This is the approach normally taken in term rewriting systems. A "fail" value is the result of applying a function to input it cannot work over. Such fail values are also used to explain the way pattern matching in normal functional languages can drop from top to bottom looking for a case that matches, and in extension typing. Systems that take this approach make this failure mechanism a first-class citizen of the language. This means that any function could return a value *or* failure. Paired with this fail value is an operation, – sometimes called "left-choice"[21], sometimes called "fatbar"[54] – which we will denote <+[3], that will call its right argument if its left emits a fail value, and will do nothing otherwise. This allows us to try a specific function and to default to the polymorphic one if it fails. Listing 3.7 shows the `increment` function encoded

[3] The <+ operator is also used in Stratego, RhoStratego and Kiama

Table 3.1: Summary of appropriateness of various *polymorphic functions with specific behaviour* techniques to compilation by a baseline functional compiler

technique	semantic	type	changes	baseline
function extension	●	●	◐	○
extension typing	○	○	○	●
universal repr	●	●	○	○
implicit type manip	●	●	○	○
explicit failure	○	○	○	●

in Stratego [87] using this technique. The same technique also works in RhoStratego, ELAN [7], and Kiama [80].

Listing 3.7: `increment` encoded in Stratego

```
rules
  incr : S(x) -> S(<add>(x,234))
strategies
  main = io-wrap(topdown(incr <+ id))
```

The encoding looks very similar to our final approach (given below) because the semantics are very similar. There are two differences between explicit failure and the system we use, the first is that the system we use succeeds or fails on *type* where the explicit failure succeeds or fails on *constructors*, in the same way extension typing does. The second is that explicit failure requires that the mechanism which defers to the right hand function (fail) is a language primitive which can escape the function, in our mechanism the failure is encapsulated.

While explicit failure works very well for term-rewriting languages, it does not do well on our criteria. It is a large departure from standard functional semantics as any computation can result in a fail value. Furthermore it is not clear how to reconcile it with a Hindley-Milner type system. One can use a boxed type, but since all values could be fail, this is not very practical.

Summary

Table 3.1 summarises the discussion of each existing technique. We can see that none immediately matches all our criteria. Extension typing and explicit failure are a long way from being appropriate but there are only a few things we need to change about function extension or either type manipulation techniques. For example, if we could find a way to do therem without type classes, or we could find a simple type class implementation

that suffice, we could use either of these. In part because of the interaction with structure agnosticism (see next section) and in part because the implementation is *simplest*, we take the approach of modifying function extension to get a solution. We will show a version of function extension which does not require type classes (or type-safe casts) and works with a single operator for extension.

3.3.2 Our solution: Function Extension with a Single Operator

We define a *function extension operator*, $\triangleright$ which takes as its right argument, a function polymorphic in its first argument and as its left argument a non-polymorphic function which is consistent[4] with the right-hand function's type. Listing 3.8 shows how we can use this operation to write the *increment* function.

Listing 3.8: The `increment` function written in DGEN using function extension

```
1  def incS(amt, s) = case [s] of
2                        { [S(s)]    -> S(s + amt)
3                        } otherwise -> error "partial definition error in incS"
4  def id(x) = x
5  def increment(amt) = incS(amt) ▷ id
```

This function extension operation is able to do the task of both `extT` and `extQ` because its type inference rule is general enough. The right-hand argument is of a polymorphic type, either $a \rightarrow a$, or $a \rightarrow X$ for some specific type X. The left-hand argument must be a function that either a) takes one specific type and returns the same type as the right argument (for the specific type X), or b) a function that takes and returns the same type, so is consistent with $a \rightarrow a$. The return type of the extended function is the same as the type of the right hand argument. This type ensures that the functions being passed in can play the roles they are asked to. The right-hand argument is the fail-over, so it must be polymorphic in its first argument. The left-hand argument is the specific behaviour, so it must have a specific type as its first argument. The final function is applied to multiple input types, so it too is polymorphic in its first argument (i.e. `increment` has type $a \rightarrow a$).

The formulation of this operator, demonstration of how it works and *the algorithms for compiling it without any advanced language features like type classes* is one part of the technical contributions of this work (Chapter 5).

[4]We define what we mean by "consistent" in Section 5.4.

3.3.3 Structure Agnosticism

Our generic functions are *generic* because they work over any *structure*. Plainly then, we need a way to be agnostic of structure. The approaches in use are split broadly into two styles:

- A way to see inside data. Possibly *a single, universal representation of data* or *a reflection mechanism*.
- Defining a few functions that work over all data structures and building more complex algorithms from these. One could have a set of *multi-step functions* designed to cover all possibilities, or *single-step functions* designed to be built up themselves. Type classes are a useful way to create such functions, albeit requiring overloading to do so.

Universal Representation of Data

The universal representation of data which was a possible solution for polymorphic functions with specific behaviour can also be used to achieve structure polymorphism. Listing 3.9 shows the generic equality function encoded in LIDG [77][5].

Listing 3.9: Generic Equality in LIDG [77]

```
1  geq :: Rep tT -> tT -> tT  -> Bool
2  geq (RInt        ep) t1 t2 =  from ep t1 == from ep t2
3  geq (RChar       ep) t1 t2 =  from ep t1 == from ep t2
4  geq (RFloat      ep) t1 t2 =  from ep t1 == from ep t2
5  geq (RUnit       ep) t1 t2 =  case (from ep t1, from ep t2) of
6                                      (Unit, Unit) -> True
7  geq (RSum  rA rB ep) t1 t2 =  case (from ep t1, from ep t2) of
8                                      (Inl a1, Inl a2) -> geq rA a1 a2
9                                      (Inr b1, Inr b2) -> geq rB b1 b2
10                                     _                -> False
11 geq (RPair rA rB ep) t1 t2 =  case (from ep t1, from ep t2) of
12                                     (a1 :*: b1, a2 :*: b2) ->
13                                       geq rA a1 a2 && geq rB b1 b2
14 geq (RType e rA  ep) t1 t2 =  geq rA (from ep t1) (from ep t2)
15 geq (RCon s rA)      t1 t2 =  geq rA t1 t2
```

The first parameter to this function is a value which encodes the representation of the second argument. This allows the `geq` function to bind the sub-values in the value being inspected and branch based on the various forms that the second argument inhabits. The fact that one representation (in this case a sum of tagged products with special forms for

[5]This example was in the code bundle which accompanies the published work.

pairs and built-in types) can encode all values means that no matter what the type being worked on, a reasonably small set of case alternatives (or function definitions in this case) can describe any data that might come along. In LIDG the type representation is encoded at the value-level but meta-programming tools can also be used to achieve this, as with Generic Haskell [35] and Generic Clean [1]. Listing 3.10 shows generic equality written with the Generic Extension to Clean.

Listing 3.10: Generic Equality in Clean [1]

```
1  generic map a1 a2          :: a1 -> a2
2  instance map Int where
3    map x                    = x
4  instance map UNIT where
5    map x                    = x
6  instance map PAIR where
7    map mapx mapy (PAIR x y) = PAIR (mapx x) (mapy y)
8  instance map EITHER where
9    map mapl mapr (LEFT x)   = LEFT (mapl x)
10   map mapl mapr (RIGHT x)  = RIGHT (mapr x)
```

These meta-programming systems are amongst the most powerful of all the generic programming tools, capable of encoding more data-type generic functions than any others [36]. Indeed such tools are typically so advanced and well integrated into the underlying language that it can be difficult to discern them as meta-programming systems. For example, Generic Haskell is able to generate interface files to maintain separate compilation, a feature sometimes lost by pre-processors.

As was the case for polymorphic functions with specific behaviour, there are similar systems that use implicit type parameters (for example EMGM, `regular` and Instant Generics) and which require type classes and/or GADTs. Implicit and explicit universal data representations for structure agnosticism have the same pros and cons we identified when discussing polymorphic functions with specific behaviour on page 38.

A Set of Pre-defined Traversals

Term rewriting systems will typically use a set of pre-defined traversals to allow structure agnosticism. For example, the Stratego strategy `all` will apply a function to all immediate children of a given value. This can then be recursively applied to get a strategy that goes top-down in any value. These pre-defined traversals can either be one-step, designed to be built into more useful traversals; or multi-step, designed to cover all the traversals you are likely to need. Listing 3.11 shows the salary updating increment function encoded in Stratego where a single-step structure agnostic traversal `all` has been built into the

required multi-step traversal td[6].

Listing 3.11: increment encoded in Stratego showing how to derive top-down traversal

```
rules
  incr : S(x) -> S(<add>(x,234))
strategies
  td(s) = s; all(td(s))
  main = io-wrap(td(incr <+ id))
```

SYB, Stratego, ELAN, Tom [5], Kiama, S'_γ [58] and Uniplate [71] use the one-step traversal approach while ASF+SDF and XQuery[7] define multi-step traversals. Maude [17] takes things even further, defining traversal to occur in one way only and requiring the rewriting rules to bend to match that. Recent work by Lämmel [62] has demonstrated that the best approach of the three is not necessarily the most flexible one.

This approach does not require any changes to the semantics or the type system of the language, only the addition of new primitive operations. However, these operations are coarse grained and alternative approaches reveal more information during compilation. Furthermore, the explicit spine that we will use is at least as expressive.

Object Reflection

Object oriented (OO) languages typically expose a model of objects at run-time and allow the programmer to perform run-time manipulation via this model. This model is sometimes called the language's *reflection API*. This is very similar to the universal representation of data approach, but is typically done with very general types. For example, while a universal representation of data system will try to keep track of what the type of the arguments of a particular value are at compile time, a reflection system will not and will rely instead on run-time casts to get everything in the right form. Listing 3.12 shows the equality function written using an hypothetical reflection API in a Java-like language[8].

The representation of data associated with reflection is a very object oriented one. For example, you can get the fields from an object, and ask what its class name is. The data is also accessed in a very object oriented way, the data about a value is accessed by methods of that value. However, OO-style reflection is not restricted to OO languages. F# [23] is a functional language (ML-style) which interacts with objects via Microsoft's common

[6]Compare this listing to Listing 3.7 where the same function is encoded but using the library provided definition of top-down traversal.

[7]Working with XML *requires* structure agnosticism since XML is semi-structured by design.

[8]Writing the function in valid Java introduces unhelpful complexity.

Listing 3.12: Generic equality encoded using OO-style reflection.

```
1   public static boolean eq(Object one, Object two){
2     Field[] oneFields = one.getClass().getFields();
3     Field[] twoFields = two.getClass().getFields();
4     boolean sofar = true;
5     for(int i = 0; i < oneFields.length; i++){
6       Field onOne = oneFields[i];
7       Field onTwo = twoFields[i];
8       sofar = sofar &&  (  onOne.getType() == onTwo.getType()
9                         && eq(onOne.getValue(),onTwo.getValue())
10                        );
11    }
12    return sofar;
13  }
```

language runtime. In F# structure agnosticism can also be achieved with an OO-view on data via reflection [9].

This approach requires a full model of data to be added, but that can be done via a library so it does not necessarily modify the semantics of the language. However, this approach does require a significant weakening of the type system since the types of values can't be known at compile time and it is up to the programmer to include appropriate run-time type checks with fail-over code if the type found is not the desired one.

The Explicit Spine View

Consider again the universal representation of data approach. This approach requires defining a scheme by which any value in the language can be described. There are as many such approaches as there are programming languages, and even if we restrict ourselves to algebraic data-types, there are a number of different possibilities. However, there is one particular view of data which distinguishes itself by being very simple and applicable in many approaches. To view data as *nested tuples* associating to the left is simpler than any other view of data used in generic tools but still suffices to describe any algebraic data type. We now describe how this view works before exploring where and how it is used in existing generic tools[9].

This has been termed the *spine view* of data by Hinze [37, 38] but we call it the *explicit spine view* to distinguish it from approaches that use it implicitly in pre-defined traversal operations (like SYB)[10][37].

We define a mapping from algebraic data type values to tuples of constructors and

[9] We describe this approach in more detail since it is the one we will adopt in DGEN.

[10] Before Hinze's work was published we referred to it as the *tuple-view* for obvious reasons but we think Hinze has found a better name.

Table 3.2: Examples of the difference between the fully applied constructor view of data and the spine view of data

fully applied constructor view	spine view
`Just(5)`	`(Just,5)`
`Cons(5,Nil())`	`((Cons,5),Nil)`
`MkString(Cons('c',Nil()))`	`(MkString,((Cons,'c'),Nil))`

literals which allows us to see any data as a tuple. For example, Table 3.2 shows a spine view of three example constructed values (all in DGEN syntax). It compares the spine view against the view of data that functional programmers traditionally use, a fully applied constructor view.

A feature of this approach is that it takes no account of types - it is driven by the values themselves. Even though algebraic datatypes are sums of products, each individual value can only be a product. Consider the `List` algebraic datatype.

```
adt list(a) = Cons(a, list(a)) | Nil()
```

Each value of this type can only be either `Cons(a, b)` or `Nil()`, not both. Thus a universal representation of data need only deal with product values. All such data is headed by either a constructor or a primitive value (in which case there are no arguments). Thus, the following mapping suffices

(CONSTRUCTORS)

$$\frac{\forall i.(d_i \equiv_{spine} d'_i)}{C(d_1, d_2, \ldots, d_n) \equiv_{spine} (\ldots((C, d'_1), d'_2), \ldots), d'_n)}$$

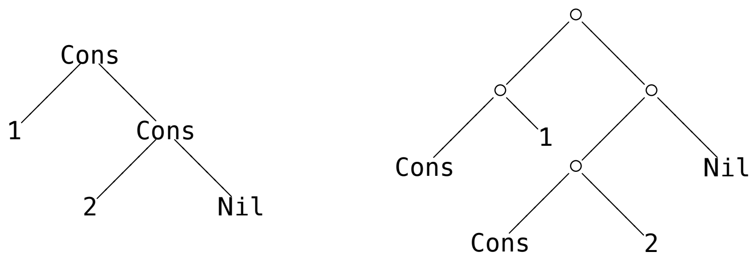
(LITERALS)

$$l \equiv_{spine} l$$

The difference between the fully applied constructor view and the spine view is perhaps easier to see in diagrammatic form. Figure 3.1 shows one value of a `list` datatype in both fully applied constructor view and explicit spine view

A system that explicitly uses the spine view will not only view data in this way, but will have facilities for the programmer to *access* data by this view. Three systems that do this are the pattern calculus, bondi and RhoStratego. In the pattern calculus and bondi, an explicit view of data by the spine view is one benefit of having first-class patterns. We will demonstrate with some bondi code. Listing 3.13 shows the generic equality function used in bondi's prelude (version 1.9)[44]. Line 2 demonstrates a pattern which can pull apart the spine view. The arguments to `equal` come in as a pair, but the pattern asks if

Figure 3.1: Fully Applied Constructor View (left) v.s. Spine View (right) for `Cons(1,Cons (2,Nil())` (i.e. `[1,2]`)



the first can match against `x1 x2` (and the second against `y1 y2`) where both `x1` and `x2` are variables. This will match against any structured value, i.e. any value that is not a constructor by itself or a primitive value, and will bind each half of the tuple to one of the variables.

Listing 3.13: Generic Equality in bondi

```
let ext (equal : a * b -> Bool) =
| (x1 x2,y1 y2) -> equal (x1,y1) && (equal(x2,y2))
| (x as _array,y as _array) ->
        let n = lengthv x in
          if  (lengthv y) eqcons n
            then
              let res = Ref True in
              ∀ 0 (n minusint 1)
                       (fun i -> res = !res &&
                                       equal(!(entry(x,i))
                                            ,!(entry(y,i))
                                            ));
              !res
            else False
| (x,y) -> x eqcons y
;;
```

RhoStratego uses the same technique. Listing 3.14 shows generic equality in RhoStratego [21].

Listing 3.14: Generic Equality in RhoStratego [21]

```
== =  c x -> d y -> ^((c == d) && (x == y))
   <+ x   -> y   -> primOp "p_primeq";
```

Again a pattern of two variables is used to access the spine view (`c x`), binding each half of the tuple to a variable which can be used in the body of the function. We will expand on exactly how this works when we discuss our own solution in Section 3.3.4.

Table 3.3: Summary of appropriateness of various *structure agnosticism* techniques to compilation by a baseline functional compiler

technique	semantic	type	changes	baseline
universal rep	●	○	○	○
defined traversals	●	●	●	●
reflection	●	○	●	●
spine view	○	○	○	○

Hinze shows how to encode the explicit spine view in Haskell, first using an encoding that does not need type classes [38] but which can't be abstracted in a library and requires GADTs, and another encoding which *can* be used as a generic library but which requires type classes [39].

At first glance, the explicit spine view looks like a poor candidate for us. It changes the semantics of the language because, as we will see in Section 3.3.4, constructors that have been stripped of their arguments are now values. It is also not clear how a pattern match such as `c x` should be typed in a Hindley-Milner system. Adding a whole new patten type brings each phase of the compiler into play because we have a new source language construct. However, despite our initial skepticism, we show that this approach fits very well in a baseline functional compiler.

Summary

Table 3.3 summarises the discussion of each existing technique. Object oriented reflection looks appealing in the table, but the loss of type safety is too great a price to pay. We would like very much to avoid requiring type classes. Not only do these not exist in all functional languages, but their interaction with existing features (like equality types in ML) is not clear. Changes that are orthogonal to other functional features would be more desirable. Thus we will not use a (non-spine) universal view of data.

Defined traversals seem to be a clear choice but there are significant benefits to using an explicit spine view. For example, Lämmel notes in [59] that pattern matching on two arguments is an alternative to the curried folds he uses in that paper. Hinze argues for the explicit spine view, despite the fact that the implicit view (i.e the defined traversals in SYB) is "equally expressive", on the basis that it "makes the definitions of some generic functions easier" [38]. Our experience is that the explicit spine view supports a useful style of generic programming (see Section 7.1). Thus, all other things being equal, we would prefer to use the explicit spine view over defined traversals. The summary table 3.3 seems to indicate that all other things *are not* equal and that defined traversals have a

significant advantage over the explicit spine view in terms of implementation effort and effects on the baseline language. However, in this thesis we show that relatively few and relatively simple changes to a baseline functional compiler *can* support the explicit spine view without large changes to the type inference mechanism or the semantics of the base language. Our contribution in this area is to show solutions to the apparent problems with the explicit spine view and to describe and demonstrate these algorithms.

3.3.4 Our Solution: Compiling Explicit Spine View

This thesis will show how to compile the explicit spine view. In all existing systems where it is used it is interpreted. Furthermore, the algorithms that do this will be simple and easy to include in a baseline functional compiler.

The explicit spine view is an alternative way to view data. Pattern matching in standard functional languages supports the fully applied constructor view and we extend this pattern matching to support the explicit spine view. More precisely, the fully applied constructor view of data is that, within a datatype, data is tagged with one of a known set of constructors and that each constructor is applied to a known number of arguments. The drop-down semantics of pattern matching supports the different possible constructors and each pattern is a constructor with patterns for each of its arguments. There is also a variable pattern for when the constructor does not need to be inspected.

The explicit spine view of data is that each datum is either an atom or a pair of datums (which we call *compound data*, *structured data* or *constructed data*), thus to support it we need one pattern corresponding to atoms and one pattern corresponding to compound data. The single variable pattern from the algebraic datatype view suffices for atoms, we only need to introduce a pattern for accessing compound data.

We add to the set of patterns, one variable applied to another, which matches against any compound data, we call these *application pattern matches*. All data that is not a literal or a zero-arity constructor is compound data. The first variable is bound to the constructor and all but one of its arguments (the last) and the second variable is bound to the last argument.

As an example, a `case` branch using this capability might be (where `f` is some function that can work on any data)

```
c(a) -> f(a)
```

Table 3.4 shows what value is bound to each variable for the examples we showed a little earlier.

Table 3.4: Binding data with application pattern matches

Data	c	a
`Maybe 5`	`Maybe`	`5`
`Cons 5 Nil`	`Cons 5`	`Nil`
`MkString (Cons 5 Nil)`	`MkString`	`Cons 5 Nil`

You will notice that this approach allows for *partially applied constructors* (like `Cons(5)`) to be bound to variables. This is a significant departure from ML-style languages but actually poses no great difficulty if we construct the type rules of the language appropriately. Even more interesting is that lonely constructors are now more like first-class values. There are two reasons these new semantics are relatively easy to deal with:

- Lonely (or partially applied) constructors are used only in accumulating values or traversing terms, they are not the end result of a computation.
- The type system we build keeps lonely (or partially compiled) constructors from appearing in any other place, for example, you don't need to perform algebraic datatype pattern matching against them.

Extending this work to make lonely constructors and partially applied constructors more extensively first-class citizens of the source language is an interesting idea, but not one we address here.

Listing 3.15: Generic Show Function

```
1  def gshow(a) :: (a) -> String
2              = case [a] of
3                   { [c(p)] -> gshow(c) ++ "(" ++ gshow(p) ++ ")"
4                   ; [z]    -> bishow(z)
5                   } otherwise -> error "partial definition error in gshow"
6
7  def bishow() :: (a) -> String
8              = let si(x) = show_int(x)
9                and sc(x) = show_char(x)
10               and sb(x) = show_bool(x)
11               and ss(x) = if (x s== "") then x else x
12               and ds(x) = show_constr(x)
13               in  si ▷ sc ▷ sb ▷ ss ▷ ds
```

Generic show, Listing 3.15, most clearly demonstrates this. The input parameter is checked to see if it is a compound or an atom. If it is a compound, recursive calls to `gshow` are made, if it is an atom then `bishow` is called. The `bishow` function must be able to show any of the atoms. This means all built in types *plus* lonely constructors. Thus you see a fail-over on line 12 which converts these lonely constructors to strings. This extra

ability must be built into the compiler as a new primitive (in this case called `show_constr`). At first glance this could be an onerous requirement but we are actually only adding one new primitive operation where there are already four. We need a show operation for all our primitives, we are just adding lonely constructors to the set of primitives. This turns out to be a very neat way to lift lonely constructor to more first-class values of the language. Once we have written these extra primitives, we don't need to make any other changes. Most notably, we don't make any (other than those already outlined for spine view) changes to the *semantics* of the language.

Note that this *view* of data does not require us to *encode* data in this way at run-time. We only need a sensible way to convert between this view and whatever we choose for encoding. In fact, since we will compile the explicit spine view to three primitive operations; `kar`, `kdr` and `ispair` (Chapter 6), we only need implementations of those three functions. It turns out to be quite easy to layer these on the data representation in our run-time and the same will be true for many other run-time data representations.

This approach first become known to us via the pattern calculus and RhoStratego. It is also used in SYB Reloaded and Revolutions [37, 38]. In Revolutions it is extended to add type information to the spine view, which we think could also be done with our system (see Section 10.4.1). Our contribution over these systems is that we show how to compile such a representation and give a new account of its typing (Chapter 6).

3.4 Summary

Of the myriad existing options for encoding generic functions, we have chosen a unique combination: function extension with a single primitive, and the explicit spine view. We have chosen these because they are sufficient to describe the generic functions we are concerned with, are very simple and, as we shall show later, they are most appropriate for compilation in a baseline functional language compiler.

Chapter 4

A Baseline Functional Language Compiler

In this chapter we define a baseline functional compiler. We describe the internal languages of the compiler and describe the algorithms used during compilation.

This thesis has (at least) two audiences:

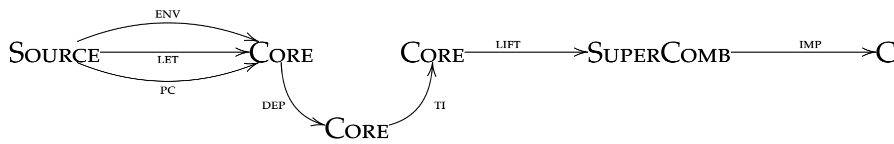
1. Those who want to build a compiler which can compile generic functions (or see how to do it, or be convinced it can be done).
2. Those who want to extend some other compiler with the ability to compile generic functions (or to see how to do it, or be convinced it can be done).

Those in this first group may ask to see a full compiler built from scratch. On the other hand, those in the second group may only be interested in understanding the difference between this generics compiler and a (mythical) standard functional language compiler.

To cater to both groups we start by presenting a full compiler *without* the ability to compile generic functions and then show how to extend it to have this ability. Thus in this chapter we ignore the presence of application pattern matches and the function extension operator; we cover them in Chapters 6 and 5 respectively.

We allow our compiler's architecture, as a series of transformations from one language to another, to determine the structure of our presentation here. Creating functional language compilers as a series of transformations is not new [3, 52, 56, 51, 74, 57] but we have achieved a particularly concise description of our whole compiler by being absolutely faithful to this idea. Figure 4.1 shows our compiler as a series of transformations from a source language to a target language, via two other languages Core and SuperComb, in five phases. Where there are concurrent translations they are defined on non-overlapping parts of the grammar so there is no ambiguity.

Figure 4.1: A Functional Language Compiler



Pattern Compilation (PC) Simplify all the patterns removing any nesting, literals and variables, etc, so that the resulting expression is essentially a switch statement. This translation is described in Section 4.1.

Dependency Analysis (DEP) Introduce as much polymorphism as possible by way of a dependency analysis. This translation is described in Section 4.2.

Type Inference (TI) The type inference step. We remain in the CORE language and fill the program with inferred types. This translation is described in Section 4.3.

Lambda Lifting (LIFT) Lift all lambdas to the top level removing any nested scopes. Also convert the "essentially switch statements" into actual switch statements. This translation is described in Section 4.4.

Imperative (IMP) Named for the target language, which is *imp*erative. Fix the evaluation order and perform other small tasks needed to make the code imperative. This translation is described in Section 4.5.

Let Lifting (LET) and Type Environment Creation (ENV) Are book-keeping transformations which reconcile the different abstract syntaxes of SOURCE and CORE. The LET translation converts the top-level definitions of SOURCE into a single `letrec` expression in CORE. The ENV translation converts the algebraic datatype definitions in SOURCE into an environment of constructor definitions in CORE. LET, PC and ENV are done in parallel and are independent of each other. ENV works only on algebraic datatype definitions, which neither of the other translations touch. The LET translation only works on the top-level of definitions, it is not recursive, and does not interact at all with PC.

Along the way we encounter three "simple applicative languages" [18]:

Source SOURCE which we met in Chapter 2, represents a desugared source language. It has all the standard features of applicative languages (function abstraction, function application, let binding, recursive lets), and also has case expressions which match

those in Haskell and ML for expressivity, notably working over a list of scrutinees and a list of patterns to emulate the equational style of writing functions. SOURCE is the abstract syntax which corresponds to the concrete syntax (DGEN) we introduced in Section 2.2.

Core CORE's task is to support type inference, thus it has much simpler `case` expressions and allows type annotations on any expression.

SuperComb SUPERCOMB is the final step before we convert to imperative form. Its primary characteristics are the lack of nested binding expressions and a `switch` statement rather than a `case` expression.

We also use one imperative language, which is C in our implementation.

4.1 Pattern Compilation

Pattern compilation translates the `case` expressions of SOURCE into the `case` expressions of CORE. Figure 4.2 shows the grammar of CORE. Any expression in CORE can be annotated with its inferred type, but we leave this out of the grammar for convenience, we only show the *programmer* type annotations that are passed through untouched from SOURCE. Literals remain unchanged between SOURCE and CORE (except that the expressions within them become CORE expressions). Note also that SOURCE and CORE restrict the scrutinee of a `case` expression to be a variable. This invariant is vital to the correctness of the final compiler, for example it is necessary for type inference of the explicit spine view (see Section 6.4).

We see the following differences in patterns compared to SOURCE (Figure 2.1), which pattern compilation must resolve:

- SOURCE has single variable patterns but CORE does not.
- SOURCE has literals in patterns but CORE does not.
- SOURCE has constructor patterns which can contain other patterns, in CORE, the constructor arguments must all be variables.
- SOURCE has a list of patterns for each alternative, CORE has only a single pattern.
- SOURCE has application pattern matches, CORE does not. We defer discussion of these until Chapter 6.

Figure 4.2: The abstract syntax of Core

$$
\begin{array}{rcll}
e & ::= & x & \text{(variable)} \\
& \mid & K\,\overline{e} & \text{(constructed value)} \\
& \mid & e\,e' & \text{(application)} \\
& \mid & \lambda x.e & \text{(anonymous function)} \\
& \mid & \texttt{let}\ \overline{x\ ot = e}\ \texttt{in}\ e' & \text{(let binding)} \\
& \mid & \texttt{letrec}\ \overline{x\ ot = e}\ \texttt{in}\ e' & \text{(recursive let binding)} \\
& \mid & \texttt{case}\ x\ \texttt{of}\ \overline{p \rightarrow e}\ \texttt{otherwise}\ e' & \text{(case expression)} \\
& \mid & l & \text{(literal value)} \\
& \mid & lo & \text{(built in operation)} \\
& \mid & \varnothing & \text{(error)} \\
p & ::= & K\,x_1 \cdots x_n & \text{(pattern)} \\
l & ::= & char & \text{(literal)} \\
& \mid & number & \\
lo & ::= & e\ (+ \mid - \mid = \mid \neq \mid < \mid > \mid \leqslant \mid \geqslant)\ e & \\
ot & ::= & \tau \mid \epsilon & \text{(optional type annotation)}
\end{array}
$$

Notice also that Core `case` expressions have only a single scrutinee, rather than the list of scrutinees in Source patterns.

Case expressions are pervasive and very common in functional code, thus for minimal run-time of the compiled executable it is very important is that they can easily be converted into a jump table (i.e. a C switch statement or assembly language jumps). Since we are compiling to C and the integers on which we are branching are densely populated, compiling to a `switch` statement is equivalent to a jump table. From this point on we will use the phrase "switch statement" to indicate a target language construct that results in a jump table. To convert a Core `case` expression to a switch requires converting each constructor pattern argument into a projection operation (which extracts the ith argument) on the scrutinee variable. In dgen this is done during lambda lifting (see Section 4.4). When we show how to compile structure agnosticism, we will use a new type of pattern. Being able to compile this pattern into a form of Core `case` expression that can still be easily converted to a switch is thus an important achievement.

The remainder of this section is concerned with describing each discrepancy between Source patterns and Core patterns in turn. For each one we describe the algorithm that resolves it. When taken as a whole they define the pattern compilation phase of the compiler. The pattern compilation algorithm, which we denote $se \Rightarrow_{pc} e$, converts a Source expression into a Core expression. The rules below only define its operation for Source

case expressions. For all other Source expressions $\Rightarrow_{pc}$ is recursively called on all subexpressions.

4.1.1 Variable Patterns

First, a little notation

head pattern Source case expressions have a *list* of scrutinees, and a *list* of patterns to match them against. The *head pattern* in a case alternative is the pattern at the head of that list, it is being matched against the head of the list of scrutinees.

If all the head patterns in a case expression are variables, then we can dispatch them all by replacing the variable in the right-hand-sides with the scrutinee corresponding to them, for example.

```
1 case [v1,v2] of
2      { [x,p1] -> foo(x)
3      ; [y,p2] -> foo(y)
4      } otherwise -> error "err"
```

$\equiv$

```
1 case [v2] of
2      { [p1] -> foo(v1)
3      ; [p2] -> foo(v1)
4      } otherwise -> error "err"
```

This only works if variables are unique within patterns. This is a constraint on the form of Source expressions that is not enforced by the grammar in Figure 2.1. Instead it is enforced here where a side condition prevents pattern compilation if there are repeated variables in a pattern. This leads to the following pattern compilation rule for variable headed patterns.

(Variable)

$$\frac{\begin{array}{r}\forall i.\exists x, se, sp_2 \cdots sp_n.(alt_i \equiv x\ sp_2 \cdots sp_n \rightarrow se \quad \wedge \\ alt'_i = sp_2 \cdots sp_n \rightarrow [x/x_1]se \quad \wedge \\ x \notin FV(sp_2, \ldots, sp_n)) \\ \texttt{case}\ x_2 \cdots x_n\ \texttt{of}\ alt'_1 \cdots alt'_m\ \texttt{otherwise}\ sf \Rightarrow_{pc} e'\end{array}}{\texttt{case}\ x_1\ x_2 \cdots x_n\ \texttt{of}\ alt_1 \cdots alt_m\ \texttt{otherwise}\ sf \Rightarrow_{pc} e'}$$

4.1.2 Constructor Patterns

If all the head patterns in a case expression are constructors, then we dispatch them all by:

1. Grouping them by constructor (no re-ordering allowed)

2. Taking each group at a time:

   (a) create one alternative for that constructor with all its arguments set to fresh variables, and

   (b) create a case expression to match these new variables with the previous constructor arguments (and all the others).

For example

```
1 case [v1,v2] of
2       { [K1(p_a,p_b),p1]  ->
            foo()
3       ; [K1(p_a,p_c),p2]  ->
            bar()
4       ; [K2(p_a), p3]     ->
            foo()
5       ; [K1(p_a,p_d), p4] ->
            bat()
6       } otherwise -> error "
           err"
```

≡

```
1  case [v1] of
2  { [K1(x_a,x_b)] -> case [x_a,x_b,v2] of
3                         { [p_a, p_b, p1] -> foo()
4                         ; [p_a, p_c, p2] -> bar()
5                         } otherwise -> error "err"
6  ; [K2(x_a)]     -> case [x_a,v2] of
7                         { [p_a,p3] -> foo()
8                         } otherwise -> error "err"
9  ; [K1(x_a,x_d)] -> case [x_a,x_d,v2] of
10                        { [p_a, p_d,p4] ->  bat()
11                        } otherwise -> error "err"
12 } otherwise -> error "err"
```

Notice that we have maintained the invariant that all the scrutinees of all the case expressions must be variables. We have also ensured that the outer case expression has only one scrutinee (i.e. is able to become a Source case expression once the inner cases are all compiled). This leads us to a rule which creates these inner case expressions, compiles them, then constructs the final Core expression. In this rule, an alternative group ag is the maximal group of patterns headed by the same constructor, i.e. the case alternatives are grouped into alternative groups partitioning them according to the value of the constructor in the head patterns. Each pattern in an alternative group is the head pattern ($K_i\ \overline{sp_i}$) and the remaining patterns ($\overline{sp'_i}$). For each alternative group we require a set of fresh variables $\overline{x}$ which is the same length as the argument list of the constructor in question ($\overline{sp}_j$).

(Constructor)

$$\frac{\begin{array}{l}\forall i,j.\exists \overline{sp}_j, \overline{sp}'_j, sf_j. \quad (ag \equiv (K\,\overline{sp}_1)\,\overline{sp}'_1 \rightarrow sf_1 \cdots (K\,\overline{sp}_p)\,\overline{sp}'_p \rightarrow sf_p \\ \qquad\qquad\qquad\qquad \wedge \quad \overline{x}\text{ new} \\ \qquad\qquad\qquad\qquad \wedge \quad \left(\begin{array}{l}\texttt{case}\ \overline{x}\ x_2 \cdots x_n\ \texttt{of} \\ \qquad \overline{sp}_1\ \overline{sp}'_1 \rightarrow sf_1 \\ \qquad \vdots \\ \qquad \overline{sp}_p\ \overline{sp}'_p \rightarrow sf_p \\ \qquad \texttt{otherwise}\ sdef\end{array}\right) \Rightarrow_{pc} f \\ \qquad\qquad\qquad\qquad \wedge \quad alt \equiv K\,\overline{x} \rightarrow f) \\ \qquad sdef \Rightarrow_{pc} def\end{array}}{\texttt{case}\ x_1\ x_2 \cdots x_n\ \texttt{of}\ ag_1 \cdots ag_m\ \texttt{otherwise}\ sdef \Rightarrow_{pc} \texttt{case}\ x_1\ \texttt{of}\ alt_1 \cdots alt_m\ \texttt{otherwise}\ def}$$

The resulting `case` expression is a Core case expression since it has only one scrutinee, and each of its `case` alternatives are a constructor with variables for arguments rather than possibly nested patterns.

4.1.3 Literal Patterns

Instead of looking at literals in a group, as we did for variables and constructors, we only consider how to compile a `case` with one alternative that has a literal as its head pattern. Code entering the pattern compiler is rarely in this form but the heterogeneous patterns rule we will soon see will create them for us.

To compile away literals, we pull them out of the `case` expression entirely and turn them into a surrounding `if` expression.

(Literal)

$$\frac{\begin{array}{c} alt \equiv l\ sp_2 \cdots sp_n \rightarrow se \\ alt' \equiv sp_2 \cdots sp_n \rightarrow se \qquad \texttt{case}\ x_2 \cdots x_n\ \texttt{of}\ alt'\ \texttt{otherwise}\ sf \Rightarrow_{pc} e' \qquad sf \Rightarrow_{pc} f'\end{array}}{\texttt{case}\ x_1\ x_2 \cdots x_n\ \texttt{of}\ alt\ \texttt{otherwise}\ sf \Rightarrow_{pc} \texttt{if}\ (l = x_1)\ \texttt{then}\ e'\ \texttt{else}\ f'}$$

Using $=$ to test the equality of the literal $(l = x_1)$ is a simplification since exactly what equality we use here is dependent on the type of the literal in question. However,

individual literals have only one possible type and that information is available in the parsed source, so no type inference is needed to make this work.

4.1.4 Heterogenous Patterns

The above rules all take away undesirable patterns *if they are in a suitable form*. Variables could only be removed if *all* head patterns were variables, similarly for constructors, and literals could only be removed when a literal-headed pattern was the only alternative present.

In the process though, each of those rules dispatched one element from the list of scrutinees. Thus, to remove all lists of scrutinees, converting them to single scrutinees, we need:

- one rule that will take alternatives in *any* form and make them suitable for one of the above rules, and
- rules to deal with an empty list of alternatives.

We describe the first in this section and the second in the next. Our first job is to define the *pattern class* relationship, which we denote $\overline{p} \equiv_c \overline{p'}$. Any pair of variable-headed patterns are in the same class, as are any pair of constructor-headed patterns (regardless of the equivalence of the constructors or the constructor arguments).

$$x\, sp_2 \cdots sp_n \equiv_c x'\, sp'_2 \cdots sp'_n \tag{4.1}$$

$$(K\, \overline{sp})\, sp_2 \cdots sp_n \equiv_c (K'\, \overline{sp'})\, sp'_2 \cdots sp'_n \tag{4.2}$$

Using this definition we can define a rule to split an heterogenous `case` expression into one that will fit one of the above rules. We do this by extracting from the top of the list of `case` alternatives, all those in the same pattern class (re-ordering is not permitted) and deferring all the others to the `case` default. For example,

```
1  case [v1,v2] of
2       { [K(p),K(p)]  -> foo
            ()
3       ; [K(p),y]     -> bar
            ()
4       ; [y, (K(p)]   -> foo
            ()
5       ; [K(p)), y]   -> bat
            ()
6       } otherwise -> error "
            err"
```

$\equiv$

```
1  case [v1,v2] of
2       { [K(p),K(p)]  -> foo()
3       ; [K(p),y]     -> bar()
4       } otherwise (case [v1,v2] of
5                        { [y, (K(p)]   -> foo()
6                        ; [K(p)), y]   -> bat()
7                        } otherwise -> error "err"
```

The first case expression is now ready for application of the Constructor rule and the remainder can be processed by another application of the Heterogenous rule.

A rule that performs this algorithm recursively is:

(Heterogenous)

$$\frac{ag \equiv (\overline{sp}_1 \rightarrow se_1) \cdots (\overline{sp}_m \rightarrow se_m) \qquad \overline{sp}_1 \equiv_c \cdots \equiv_c \overline{sp}_m \qquad \overline{sp}_m \not\equiv_c \overline{sp}_{m+1} \\ f' \equiv \texttt{case}\ x_1 \cdots x_n\ \texttt{of}\ \overline{alt}\ \texttt{otherwise}\ f \qquad \texttt{case}\ x_1 \cdots x_n\ \texttt{of}\ ag\ \texttt{otherwise}\ f' \Rightarrow_{pc} e'}{\texttt{case}\ x_1 \cdots x_n\ \texttt{of}\ ag\ \overline{alt}\ \texttt{otherwise}\ f \Rightarrow_{pc} e'}$$

4.1.5 Empty Patterns

To complete the removal of multiple scrutinees/patterns, we now need to define what to do for an "empty" case expression. In other words, one that has either no scrutinees or no patterns in each alternative, or has no alternatives at all. If we have corresponding empty lists (denoted []), the result will be the first body expression, since case alternatives are checked top to bottom and empty patterns match trivially. If this is not the case, then the result is the default expression.

(Empty Success)

$$\frac{se_1 \Rightarrow_{pc} e_1}{\texttt{case}\ []\ \texttt{of}\ ([] \rightarrow se_1) \cdots ([] \rightarrow se_m)\ \texttt{otherwise}\ f \Rightarrow_{pc} e_1}$$

(Missing Scrutinees)

$$\frac{sf \Rightarrow_{pc} f}{\texttt{case}\ []\ \texttt{of}\ \overline{alt}\ \texttt{otherwise}\ sf \Rightarrow_{pc} f}$$

(Missing Alternatives)

$$\frac{sf \Rightarrow_{pc} f}{\texttt{case}\ \overline{se}\ \texttt{of}\ []\ \texttt{otherwise}\ sf \Rightarrow_{pc} f}$$

(Missing Patterns)

$$\frac{sf \Rightarrow_{pc} f}{\texttt{case}\ \overline{se}\ \texttt{of}\ ([] \rightarrow se_1) \cdots ([] \rightarrow se_m)\ \texttt{otherwise}\ sf \Rightarrow_{pc} f}$$

4.1.6 Related Work

Pattern compilation is as old as equational style programming itself [12, 40]. Two approaches have most commonly been used, *decision trees* and *backtracking automata*. Decision tree algorithms convert the case expression into a series of tests, at each test one part of the scrutinee is inspected and all possible values for it are checked. Based on which value it matches with, you follow a branch to a subtree and repeat the process. Backtracking automata algorithms are very similar but do not guarantee that once you inspect one

part of the scrutinee, you won't have to inspect it again. Although they generate a tree, a certain part of the scrutinee might be checked at more then one level of the tree. The algorithm we present here is a backtracking automata.

The main advantage of backtracking automata is that the size of the resulting `case` is fixed and small, however the resulting code may inspect a value more than once before ultimately deciding which branch to take. The advantage of decision trees is that each value is examined only once, although the resulting tree can be unreasonably large [69] and the compilation algorithms are more complex. Compounding this complexity are questions regarding the heuristics which can guide the compilation [79].

The backtracking automata approach is most clearly given by Wadler and Barrett [88, 6] but LeFessant's treatment, which includes some very useful optimisations [63] is also an excellent description. There exists a very large number of papers discussing the decision tree approach. Maranget's paper describing some improvements to the standard approach [69] is a good place to start because it includes a nice description of the standard algorithm and an example of the kinds of heuristics employed to try and improve on it.

We have chosen a backtracking approach largely for simplicity. However, we also think the tradeoff between examining values more than once and unrestrained worst case behaviour falls on the side of backtracking automata when the "examination" is a C switch on a single tag (i.e. a very efficient operation).

4.2 Dependency Analysis

In later stages of the compiler, any set of definitions bound in the same `letrec` are treated as mutually recursive, even if they are not strictly so. Thus we need a phase (dependency analysis) that can ensure that any definitions left in a `letrec` expression are known to be mutually recursive, giving the type inferencer the best chance of doing its job.

The result is a nested sequence of `let` and `letrec` expressions where any definitions in a `let` are known not to be mutually recursive and any in a `letrec` are known to be mutually recursive.

While this phase takes no specific account of polymorphic recursion, it does minimise the chance it will exist by minimising the definitions that are treated as mutually recursive.

4.2.1 Polymorphic Recursion

We give a specific definition of polymorphic recursion because the existing literature is somewhat confused and contradictory. We contend that there are two factors creating

this confusion.

- In languages where dependency analysis is left to the programmer, functions that are not polymorphic recursive can look as if they are [30]. For example in ML, the compiler will treat all definitions in a `letrec` expression as if they are all mutually recursive when often they are not.

- We have found very few examples of necessarily polymorphic recursive code being used in non-experimental contexts. The only example we have found in standard functional programs is due to Peyton Jones [53] and even then the problem was resolved with some judicious refactoring. Only in generic functions (generic traversal in particular) do you regularly see programs which rely on polymorphic recursion. For example, the `apply2all` function in bondi [44], which is used for generic traversal, is a very small function which relies completely on polymorphic recursion. The function below is a slightly simplified version of the one found in bondi's prelude.

```
1  let ext (apply2all : (all a. a -> a) -> b -> b) f  z = f ((
2    | x y -> (apply2all f x) (apply2all f y)
3    | x -> x)
4  z)
5  ;;
```

The `apply2all f` function has type $b \to b$ and that function is applied to two different arguments in the body of the first pattern alternative. `x y` is bondi's syntax for application pattern matches, so `x` and `y` have different types. To apply the polymorphic function `apply2all f` to two different types in the body of that pattern alternative requires polymorphic recursion.

A definition of polymorphic recursion

> A function (with type $\forall a.T(a)$) is *polymorphic recursive* if calling it with instantiated type $T(A)$ results in another call to it with a different instantiated type $T(B)$, either from the body of that function or from some function with which it is mutually recursive.

where

> Two functions are *mutually recursive* if each can cause a call to the other, even if via some intermediary functions.

Figure 4.3: The SPLIT algorithm used in dependency analysis

(SPLIT)

$$\begin{aligned}&\text{given } a = \texttt{letrec}\ (x_1 = e_1) \ldots (x_n = e_n)\ \texttt{in}\ f\\&\quad\text{and } TS(SC(DG(x_1 = e_1, \ldots, x_n = e_n))) \text{ evaluates to } \{g_1, \ldots, g_m\}\\&\text{then } split(a) \equiv \texttt{let(rec)}\ g_1^\dagger\ \texttt{in} \ldots \texttt{let(rec)}\ g_m^\dagger\ \texttt{in}\ f\end{aligned}$$

4.2.2 Dependency Analysis Algorithm

Definitions are grouped using three standard algorithms. Firstly we create the dependency graph for each definition (DG) and from this dependency graph we calculate the strongly connected components (SC). Lastly, we topologically order these components (TS).

The dependency transformation, which we denote $e \Rightarrow_{dep} e'$, is the result of applying $split$ (Figure 4.3) to all `letrec` expressions in e.

The dependency graph for a `letrec` expression has nodes for each variable bound in the definitions and an edge to any variable whose body uses that variable. By calculating the strongly connected components of this graph, we are finding maximal mutually recursive sets of definitions. The remaining dependencies *between* these strongly connected components indicate when a binding must be defined *within* another `letrec` since it is dependant on its bindings. Thus we topologically sort the graph whose vertices are the strongly connected components where there are edges between two strongly connected components if there is a edge between any two of their members. The result is a sorted list of strongly connected components ($\{g_1 \ldots g_n\}$) where each variable in g_i is dependent on all the other variables in g_i *and* on at least one in $\{g_1 \ldots g_{i-1}\}$. We denote the set of *definitions* (binding variable with its right hand side) that corresponds to the strongly connected components g as $g^\dagger$. The SC algorithm tags groups as either singleton groups indicating that this definition is not mutually dependent on anything, components with more than one element indicating mutually recursive definitions, or self-referential, indicating a normal recursive function. Each component is then put in a `let` or `letrec` expression which binds all the definitions in the group and has the `let(rec)` resulting from the groups after it as its body. Singletons get a `let`, while multiple definition and self-referential components get `letrec` expressions.

4.2.3 Related Work

Dependency analysis is relatively simple and, as such, this is a process which doesn't support competing implementation decisions in practice. Our algorithm is derived from, and equivalent to, the algorithm presented in [54].

It was Henglein who first discovered that type inference for polymorphic recursion was undecidable [33]. Henglein also explained why you can still achieve type inference in practice most of the time. This is something that compilers can exploit, for example Mercury [32] uses iteration with an upper limit to calculate polymorphic recursive types [75, 84]. We will discuss in Section 6.5 our own approach to dealing with the polymorphic recursion that remains after dependency analysis.

4.3 Type Inference

4.3.1 FCP

Our type inference algorithm is based on FCP [48] which is, in-turn, based on algorithm W [19]. This is not a common choice for language implementors, but we have greatly enjoyed working with this particular extension of basic Hindley-Milner type inference. FCP has a number of characteristics that recommended it to us:

- FCP has explicit construction and destruction of data. This makes it easy to add algebraic data types and pattern matching functions on them. We have extended the FCP destruction expression to a multi-branched `case` expression without having to alter the underlying FCP inference algorithm.
- Its support for existential types has been invaluable. The very simple way in which existential types are provided in FCP is used in Section 6.4 to infer types for the explicit spine view.
- Its support for universally quantified type variables has been sufficient to supported higher ranked types and polymorphic recursion, two type system features we require.

FCP extends Hindley-Milner by having constructors witness higher rank types (i.e. types with universal and existential quantifiers within them). It does this by extending unification to respect a set of constant variables (we will call these *fixed for unification* variables) which can't be substituted for (V), and adding construction and destruction rules

Figure 4.4: Unification in FCP (and in DGEN)

$$\tau \overset{id}{\sim} \tau \bmod V \tag{id}$$

$$\left.\begin{array}{l} \alpha \overset{[\tau/\alpha]}{\sim} \tau \bmod V \\ \tau \overset{[\tau/\alpha]}{\sim} \alpha \bmod V \end{array}\right\} \quad \alpha \notin V \cup TV(\tau) \tag{var}$$

$$\frac{\tau \overset{U}{\sim} v \bmod V \qquad U\tau' \overset{U'}{\sim} Uv' \bmod V}{(\tau \to \tau') \overset{UU'}{\sim} (v \to v') \bmod V} \tag{fun}$$

for data which include side conditions to ensure quantified type variables don't escape their scope. FCP uses the modified unification algorithm shown in Figure 4.4. It checks the fixed for unification variables when it performs the occurs-check, but beyond that it is standard unification.

4.3.2 Algorithm

Figure 4.5 shows *our* FCP type inference rules. They differ from those in [48] in four ways:

1. Rather than break data with a "pattern matching lambda", we use a single-alternative case expression. The only difference is that the case expression includes the expression to which a pattern matching lambda would be applied (the scrutinee). We do this because CORE has `case` expressions rather than pattern matching lambdas.
2. Our MAKE rule includes a different side condition which corrects an error in the original paper.
3. We introduce a `letrec` expression and an associated type inference rule.
4. We provide a set of inference rules for primitive operations, exemplified here by two specific rules PLUS and INTEQ. The type inference rules for $-$ and $*$ follow the same pattern as PLUS and the rules for $<$ and $>$ follow the same pattern as EQ.

You will notice that the term language in Figure 4.5 does not match CORE exactly, it is something approaching a subset of CORE. In particular constructors in this system have only one argument. Presenting the type inference rules for all of CORE is prohibitively expensive in terms of pages and our reader's attention. We present here all the features that are needed to understand the type inference algorithm as it grows throughout the thesis. In Chapter 9 we provide a soundness proof for the type system underlying this in-

ference algorithm. We present here only the type *inference* algorithm (not the type *checking* relation) because we are focussed on the implementation of the type system.

Our type inference algorithm, denoted $TA \vdash e \colon \tau \bmod V$, takes A (the environment), e (the expression we are inferring the type for) and V (the set of fixed for unification variables) as input and returns T (the set of newly discovered substitutions) and τ (the calculated type for e) as outputs. We use $A_x, x : \tau$ to represent the environment where any existing binding for x has been replaced with a binding of x to τ. The notation $\sigma_K = \forall\gamma.(\forall\alpha.\exists\beta.\tau) \rightarrow \tau'$ denotes the process of looking up the constructor K in a constructor environment (populated from abstract datatype definitions) and substituting all quantified variables with new variables. $[\alpha/\tau]$ indicates the type variable α is substituted with the type τ. $Gen(A, \tau)$ is the generalisation of τ in the environment A, that is, all variables free in τ and not free in the environment are explicitly universally quantified.

When types are calculated for a let binding, they are put into the environment in a fully generalised form. When they are extracted from the environment with the Var rule, we create fresh instances of these type schemes by substituting all quantified type variables with fresh type variables. The Abs rule guesses a type for the bound variable and puts that guess in the environment (un-generalised) before calculating the type of e. The App rule first calculates the type of each expression, then calculates the unification necessary to get them in the right shape, using a new variable (α) for the unknown type.

The Let rule first calculates the type of e before fully generalising it. The environment is updated with this type scheme and the type of f is calculated under this updated environment. The LetRec rule is very similar but it first guesses a type for the binding since it may be present in its own definition. The guess is unified with the calculated type before calculating the type of f. The Let and LetRec rules are the source of polymorphism in this system. Notice that the optional type annotations are ignored at this stage. We will consider them in Sections 5.7 and 6.5.1.

The Make rule tells us how to create programmer defined data types from information in a constructor environment. We look up this environment and get a fresh instance of the type for K. We calculate the type of e and ensure it unifies with the type we obtained from the constructor environment. The type variable α denotes the universally quantified variable obtained from the constructor environment and thus these must become fixed for unification and can't be allowed to escape this instance of the Make rule. We have corrected an error in the original FCP algorithm by adding $U\tau'$ to the set of types to check for free instances of α. The original FCP algorithm had the side condition $\alpha \notin FV(UTA)$

Figure 4.5: The heart of DGEN's type inference algorithm

Type Language

$$\sigma ::= \forall\alpha.\sigma \mid \tau \qquad \tau,\rho ::= \alpha \mid \tau \to \rho \mid \mathrm{Int} \mid \mathrm{Bool}$$

Term Language

$$e, f ::= x \mid e\ f \mid \lambda x\ ot.e \mid K\ e \mid \texttt{let}\ x = e\ \texttt{in}\ f \mid \texttt{letrec}\ x\ ot = e\ \texttt{in}\ f \mid \texttt{case}\ y\ \texttt{of}\ (K\ x) \to e$$

Type Inference

(VAR)
$$\frac{(x : \forall\alpha.\tau) \in A \qquad \beta \text{ new}}{A \vdash x : [\alpha/\beta]\tau \text{ mod } V}$$

(ABS)
$$\frac{T(A_x, x : \alpha) \vdash e : \tau \text{ mod } V \qquad \alpha \text{ new}}{TA \vdash \lambda x\ ot.e : T\alpha \to \tau \text{ mod } V}$$

(APP)
$$\frac{TA \vdash e : \tau \text{ mod } V \qquad T'TA \vdash f : \tau' \text{ mod } V \qquad T'\tau \overset{U}{\sim} (\tau' \to \alpha) \text{ mod } V \qquad \alpha \text{ new}}{UT'TA \vdash e\ f : U\alpha \text{ mod } V}$$

(LET)
$$\frac{TA \vdash e\colon \tau \text{ mod } V \qquad \sigma = Gen(TA, \tau) \qquad T'(TA_x, x\colon \sigma) \vdash f\colon \rho \text{ mod } V}{T'TA \vdash (\texttt{let}\ x = e\ \texttt{in}\ f)\colon \rho \text{ mod } V}$$

(LETREC)
$$\frac{\begin{array}{c} T(A_x, x\colon \alpha) \vdash e\colon \tau \text{ mod } V \\ T\alpha \overset{U}{\sim} \tau \text{ mod } V \qquad \sigma = Gen(UTA, U\tau) \qquad T'(UTA_x, x\colon \sigma) \vdash f\colon \rho \text{ mod } V \end{array}}{T'UTA \vdash (\texttt{letrec}\ x\ ot = e\ \texttt{in}\ f)\colon \rho \text{ mod } V}$$

(MAKE)
$$\frac{\begin{array}{c} \sigma_K = \forall\gamma.(\forall\alpha.\exists\beta.\tau) \to \tau' \\ \alpha, \beta, \gamma \text{ new} \qquad TA \vdash e\colon \rho \text{ mod } V \qquad \rho \overset{U}{\sim} \tau \text{ mod } (V \cup \{\alpha\}) \qquad \alpha \notin TV(UTA, U\tau') \end{array}}{UTA \vdash (K\ e)\colon U\tau' \text{ mod } V}$$

(BREAK)
$$\frac{\begin{array}{c} \sigma_K = \forall\gamma.(\forall\alpha.\exists\beta.\tau) \to \tau' \qquad \alpha, \beta, \gamma \text{ new} \qquad T(A_x, x\colon \tau) \vdash e\colon \tau_e \text{ mod } (V \cup \{\beta\}) \\ \beta \notin TV(TA, \tau_e, T\alpha) \qquad T'TA \vdash y\colon \tau_y \text{ mod } V \qquad T'(T\tau' \to \tau_e) \overset{U}{\sim} \tau_y \to \delta \qquad \delta \text{ new} \end{array}}{UT'TA \vdash (\texttt{case}\ y\ \texttt{of}(K\ x) \to e)\colon U\delta \text{ mod } V}$$

(PLUS)
$$\frac{TA \vdash e : \tau \text{ mod } V \qquad \tau \overset{U}{\sim} \mathrm{Int} \qquad T'UTA \vdash e' : \tau' \text{ mod } V \qquad \tau' \overset{U'}{\sim} \mathrm{Int}}{U'T'UTA \vdash e + e' : \mathrm{Int} \text{ mod } V}$$

(INTEQ)
$$\frac{TA \vdash e : \tau \text{ mod } V \qquad \tau \overset{U}{\sim} \mathrm{Int} \qquad T'UTA \vdash e' : \tau' \text{ mod } V \qquad \tau' \overset{U'}{\sim} \mathrm{Int}}{U'T'UTA \vdash e = e' : \mathrm{Bool} \text{ mod } V}$$

which is not enough to prevent ill-typed expressions from being allocated types.

The original FCP Break rule uses a pattern matching lambda expression which breaks its input if and only if it is the right constructed value. We replace this with a single alternative `case` expression which has the same semantics. A single branch like this is not sufficient for Core since it precludes pattern compilation, but the type inference rules for the multi-branched `case` expression are too complex to include here and the full rule contains nothing of substance beyond the content of the single branch rule presented here. For our single branch version, we look up the constructor environment and create a fresh instance of this constructor's type. We use its input type as the assumed type for x in the environment when calculating the type of e. To ensure that any existentially quantified variables don't escape from the scope of this case clause, which they might since this rule pulls the potentially higher rank type from within a constructed value, β is added to the set of fixed for unification variables when calculating the type of e. Any existentially quantified variables remaining free in TA, τ_e or $T\alpha$ are an error.

4.3.3 Related Work

There were any number of type inference algorithms we could have used in dgen. Starting from the original Damas-Milner algorithm W [19], through versions with better error messages [64] to algorithms able to implement common functional programming extensions [82]. In fact there exists dozens of different W-inspired type inference algorithms in the literature. We have chosen FCP because it occupies common ground between all modern functional compilers, i.e. *algorithm W extended with type constructors*. By building from this starting point we make our work as widely applicable as possible. Furthermore, FCP is a particularly simple algorithm compared to other inference algorithms capable of higher ranked types, which we will require.

4.4 Lambda Lifting

Executing a program in the presence of nested environments is more complicated than executing one without them. Thus we would like to get rid of any language characteristic that causes this. In Core it is the presence of nested λs, `let`s and `letrec`s that create nested environments.

To remove these, we translate to a language (SuperComb) where there is only a single, top-level `letrec`-equivalent. For this to work, each of the definitions in the `letrec`-equivalent need to contain *no free variables*. Thus lambda lifting is the process of taking

λs, `lets` and `letrecs`, making their free variables into bound variables and *lifting* them to a single top-level `letrec`-like expression.

4.4.1 Lifting Cases

The other feature we compile away at this stage is the `case` expression. Although there is an equivalent expression in SuperComb, it is a `switch` statement which is semantically equivalent to a C `switch` statement. In pattern compilation we removed most of the complexity in the Source `case` and in lambda lifting we complete the job.

Case alternatives operate somewhat like λ-bindings in that they bind variables and cause the execution of the body with those bound variables in scope. Thus it is natural to ask whether we need to lift the alternatives in `case` expressions as well. In fact we do not. We could rely on the surrounding λ-binding (as there must be one) to do the hard work and then use a simple scheme to ensure the newly bound variables are where we expect them to be. It is difficult to determine exactly the algorithm used in other compilers, but at least one similar functional compiler [67] takes this approach and it is also the solution given in [55].

In this work however, we take an alternative approach. We choose to lift each `case` alternative to the top-level `letrec`. We do this for two reasons:

- Combined with our algorithm for converting to C it makes the passing of parameters bound in the pattern more explicit.
- It allows the `switch` in SuperComb to be *exactly* a C `switch` statement. We feel this makes the transformation from functional to imperative code more explicit. In particular, it makes the compilation of pattern matching into a `switch` statement absolutely clear and this is of prime concern for us. An early potential casualty of the addition of generic capabilities is this compilation to `switch` statements. One of our claims is that we support generic code while maintaining the compilation to `switch` for normal patterns, something we want to make as clear as possible.

Once we are convinced that removing nested λs, `lets` and `letrecs` plus further simplifying `cases` is a good idea, the best way to see where we are going is to look at a language with these characteristics. In this compiler, that is SuperComb. The grammar for SuperComb is presented in Figure 4.6

A program is a list of mutually recursive definitions (super-combinators) plus a "main" expression. This is the (implicit) top-level `letrec`. Each super-combinator is a name (kn),

Figure 4.6: The abstract syntax of SuperComb

$$
\begin{aligned}
kp &::= \overline{kd}\ \texttt{in}\ ke \\
kd &::= kn\ \overline{x} = ke \\
ke &::= x \\
&\mid\ K\ \overline{ke} \\
&\mid\ ke\ ke \\
&\mid\ kn\ \overline{x} \\
&\mid\ ke^{i} \\
&\mid\ \texttt{switch}\ x\ \texttt{of}\ \overline{K \to ke}\ \texttt{otherwise}\ ke \\
&\mid\ l \\
&\mid\ lo \\
&\mid\ \varnothing
\end{aligned}
$$

a set of formal parameters ($\overline{x}$) and a definition (ke) All defined names must be defined here and none of these can have free variables (although this is not enforced in the above grammar). This leaves us with only variables (x), constructors ($K\ \overline{ke}$), applications ($ke\ ke'$), literals (l), literal operations (lo), errors ($\varnothing$), super-combinator names ($kn\ \overline{x}$, which we explain below), and `switch` expressions. Each `switch` alternative is a *constructor symbol only* and an expression to run if that alternative is fired, plus a distinguished default expression to run if none of the others others work. For all these switch alternatives, the expression to run must be a super-combinator name applied to all its arguments (although that is not enforced in the grammar either). Finally, we add a projection expression (ke^{i}) which allows us to pull the arguments from a constructed value, which is necessary since Source lacks binding operations that can pull constructor expressions apart.

The differences between Core and SuperComb are:

switch rather than case As discussed above.

No lets, letrecs or λs As discussed above

New projection expression. We add a new type of expression, projection ke^{i}. This pulls the ith argument from the expression if it is a constructed value and is undefined otherwise.

Program is a list of expressions Actually a list of super-combinator name/expression pairs. This is the top-level `letrec`-like expression we have referred to.

Super-combinator name expression We need a way to identify a call to a super-combinator.

Since all super-combinators are named, we introduce a new expression for calling super-combinators and passing in their arguments, $kn\ \overline{x}$.

We will show how to reconcile all these differences with the lambda lifting algorithm. Note that the constructor environment used in typing is passed through lambda lifting, we need it when we add the extension operation, but it is not part of SuperComb.

4.4.2 Algorithm

The lambda lifting algorithm, $\overline{kd} \mid e \Rightarrow_l kp$, converts a Core expression into its equivalent SuperComb program. It accepts as input a so-far accumulated list of super-combinators $\overline{kd}$ plus a Core expression e and returns a complete SuperComb program kp. To lift a Core expression we seed the algorithm with an empty list of so-far accumulated super-combinators. Note that we omit the optional type annotations from the Core expressions for simplicity.

Lambdas represent anonymous functions, we want to promote each to a top-level definition. In its place we will put the newly created super-combinator, applied to all the arguments it needs.

$$\text{(Lambda)}\quad \frac{\overline{kd_0} \mid e \Rightarrow_l \overline{kd}\ \texttt{in}\ ke \qquad \overline{v} \equiv fv(ke) \qquad kn\ \text{new}}{\overline{kd_0} \mid \lambda x.e \Rightarrow_l \overline{kd}\ (kn\ \overline{v}\ x = ke)\ \texttt{in}\ kn\ \overline{v}}$$

We would also like to promote any name defined in a `let` to a top-level definition. We don't need to discover its free variables because the bottom-up application of other rules will ensure there are none. However, the name may be used in the body to refer to this definition, so we need to change all such instances to the name of the new super-combinator. The Let rule works on one definition at a time and is applied to the remaining `let` until there are no definitions left.

$$\text{(Let)}\quad \frac{\begin{array}{c}\overline{kd_0} \mid \texttt{let}\ (x_2 = e_2) \cdots (x_n = e_n)\ \texttt{in}\ [x_1/kn]e \Rightarrow_l \overline{kd}\ \texttt{in}\ ke \\ \overline{kd} \mid e_1 \Rightarrow_l \overline{kd'}\ \texttt{in}\ ke' \qquad kn\ \text{new}\end{array}}{\overline{kd_0} \mid \texttt{let}\ (x_1 = e_1) \cdots (x_n = e_n)\ \texttt{in}\ e \Rightarrow_l \overline{kd'}\ (kn = ke')\ \texttt{in}\ ke}$$

`letrec`s are lifted in exactly the same way as `let`s except that the substitution of the new super-combinator name for the old definition name needs to occur on the bodies of

all sibling definitions (those in the same `letrec`).

(LetRec)

$$\frac{\overline{kd_0} \mid \texttt{let}\ (x_2 = [x_1/kn]e_2) \cdots (x_n = [x_1/kn]e_n)\ \texttt{in}\ [x_1/kn]e \Rightarrow_l \overline{kd}\ \texttt{in}\ ke \qquad \overline{kd} \mid [x_1/kn]e_1 \Rightarrow_l \overline{kd'}\ \texttt{in}\ ke' \qquad kn\ \text{new}}{\overline{kd_0} \mid \texttt{let}\ (x_1 = e_1) \cdots (x_n = e_n)\ \texttt{in}\ e \Rightarrow_l \overline{kd'}\ (kn = ke')\ \texttt{in}\ ke}$$

Our aim in lifting `case` expressions is to remove the binding that occurs in the pattern match. Since `case` expressions only have very simple patterns (a constructor and one variable for each of its arguments) our job is to pass each of these constructor arguments to the body of an alternative and to transform the body into a form that expects arguments. We could do this in two steps, first converting the body to a λ expression that explicitly takes in each argument bound in the pattern and then applying each projected argument to that λ expression. The second step would then be to lift this λ expression. Instead we do both steps in one. The projection expression introduced in SuperComb is applied to the scrutinee to pull out each argument (the pattern match is gone). Each of these is applied in-turn to a super-combinator generated from the body. If we use the binding variable names from each alternative as the parameter names for the corresponding super-combinator, free variables in the body of each alternative will still refer to the exact same values as they originally did.

When all `case` branches are done, the scrutinee and the default branch are lifted. The result is a `switch` statement that has the new branches plus the lifted scrutinee and default branch.

(Case)

$$\frac{\begin{array}{l} \forall i.(alt_i \equiv K_i\ \overline{x_i} \rightarrow e_i \quad \wedge \\ \quad alt'_i \equiv K_i \rightarrow kn_i\ (\forall j.e_s{}^j) \quad \wedge \\ \quad kn_i\ \text{new} \quad \wedge \\ \quad \overline{kd_{i-1}} \mid e_i \Rightarrow_l \overline{kd_i}\ \texttt{in}\ ke_i) \\ \overline{kd_n} \mid e_s \Rightarrow_l \overline{kd_s}\ \texttt{in}\ ke_s \qquad \overline{kd_s} \mid e_d \Rightarrow_l \overline{kd_d}\ \texttt{in}\ ke_d \end{array}}{\begin{array}{c} \overline{kd_0} \mid \texttt{case}\ e_s\ \texttt{of}\ alt_1 \cdots alt_n\ \texttt{otherwise}\ e_d \Rightarrow_l \\ \overline{kd_d}\ (kn_1\ \overline{x_1} = ke_1) \cdots (kn_n\ \overline{x_n} = ke_n)\ \texttt{in switch}\ ke_s\ \texttt{of}\ alt'_1 \cdots alt'_n\ \texttt{otherwise}\ ke_d \end{array}}$$

4.4.3 Related Work

There are a few variants of lambda lifting, the most celebrated of which is fully lazy lambda lifting [54]. Our ground rules (page 9) state that we are building on a baseline

functional compiler, which calls for a relatively simple lambda lifter. However it is clear that all of the other lifters of which we are aware could be extended to handle the lifting of case alternatives and thus could act as a suitable replacement for our lifter. The algorithm we present above was derived from the descriptions in [47, 54, 55] extended to handle the lifting of case alternatives. While we have not found articles describing the lifting of case alternatives in this situation, it would be surprising to find it has never been done.

4.5 Conversion to Imperative

Translating SuperComb programs to C programs is yet another relatively complex job. We will *not* describe this phase in the same detail as the others, because:

Less-standard There exists far less common-ground amongst the approaches to this phase described in the literature than for the other phases. This makes any standard algorithm we might try to present less convincing and less applicable to existing compilers.

Simpler changes The modifications we will make to this phase are relatively simple.

For these reasons there is little to be gained from adopting the approach we have taken to describing the earlier phases of the compiler. Instead we describe this phase in general terms and do the same for the modifications we will require to it.

We take as our starting point, the super-combinator compiler *Epic*, which was created as part of the Idris project [8]. Epic (suitably customised) takes each super-combinator definition in SuperComb and converts it into a C function. Each primitive-operation is converted into a call to a function defined in a C run-time. Epic links the run-time with the compiled code to create the final program. Epic does not use any particular reduction machine and is strict by default, although it does provide a mechanism for lazy evaluation. Epic is a reasonably efficient and relatively simple super-combinator compiler that we have found simple to use, appropriate for our needs and easy to extend.

4.6 Summary

In this chapter we have described a complete functional language compiler. It contains valuable features like an inferencing type system, compilation to C via super-combinators plus compilation of pattern matching. This compiler could compile an ML-like language

as it is, and known techniques could be used to expand it to compile Haskell-like languages.

We have described in detail pattern compilation, dependency analysis, type inference and lambda lifting. The next two chapters will add novel compilation techniques for polymorphic functions with specific behaviour and structure agnosticism, modifying pattern compilation and type inference to do so. We *will not* need to make any modifications to dependency analysis or lambda lifting. It is an important characteristic of this work that these algorithms are unaffected by our changes.

Chapter 5

Compiling and Typing Polymorphic Functions with Specific Behaviour

In this chapter we show how to compile the function extension operation, $\triangleright$, our solution to polymorphic functions with specific behaviour. We will add a new phase to the compiler which converts it (using type information) to a simpler primitive operation (`typeOf`). We then show how to implement this simpler operation in C. To effectively *use* $\triangleright$, we need rank-2 types, which we describe. We also give the type inference rule for $\triangleright$.

When extending existing compilers, it is simplest to implement new features in a way that does not interact with any other parts of the compiler. For example, we can add a new built-in type and some operators for it with small atomic additions through the compiler. However, $\triangleright$ has a few features which prevent us from implementing it in this way. s

It is type-driven The choice of which function to call is dependent on the type of both the specialising function and of the data it will eventually work on.

It generates a function `a` $\triangleright$ `b` is a function built from two other functions. It takes the argument that would have gone to `a` or `b` and inspects it before passing it on to one of them.

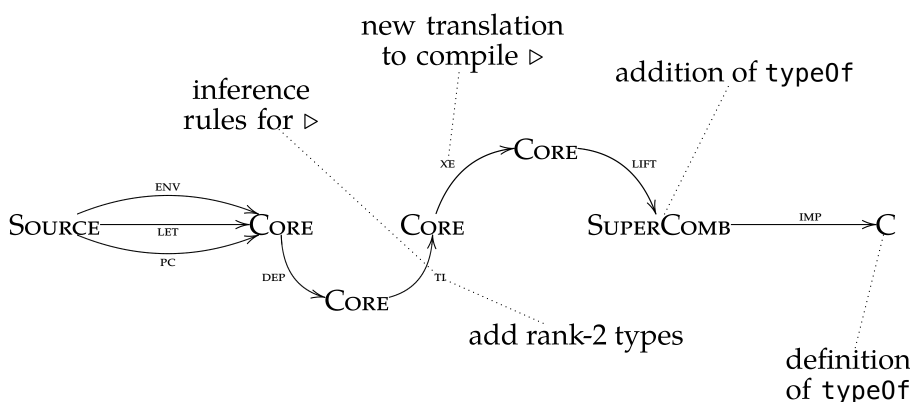
This limits our options. There are no types before type inference and we have converted functions to super-combinators by the time we get to SuperComb. One could work out what the equivalent super-combinator for the function is and then ensure generated type information is passed around to the right super-combinators, but this approach is unnecessarily complex. If we need types and functions, we should deal with $\triangleright$ in the place where we have both of these. This place is between Core and lambda lifting, thus we need to add a new phase to the compiler that sits in here and which compiles $\triangleright$ into features

that *do* exist later in the compiler.

As an aside, the lambda lifting algorithm we presented in Section 4.4 works from the bottom up and thus can't be extended to do this extra step. By the time the lambda-lifter sees `a ▷ b`, `a` and `b` are both super-combinators and sufficiently divorced from the code that originally defined them as to make dealing with ▷ more complex than we would like. A top-down lambda-lifter would solve this problem but to our knowledge no top-down lambda lifter has been described in the literature.

5.1 Overall Journey

We have an expression (▷) in SOURCE, which we have ignored to this point. It is also present in CORE and it we will compile it away before we reach SUPERCOMB, i.e. we compile it away as soon as we have calculated types. We compile it into an expression that uses a low-level primitive called `typeOf` which simply returns the type of whatever is passed to it. The bulk of the work dealing with ▷ is in inferring its type and describing how to convert it into `typeOf`. The transformation which does this (extension elimination, XE) is a precursor to lambda lifting, creating an CORE expression with no extension operations and with a new primitive operation `typeOf`. CORE without ▷ and with `typeOf` is strictly a new internal language but we don't include this slight variant of CORE in our diagram for the sake of simplicity. In the actual implementation of DGEN, CORE contains both ▷ and `typeOf` in its definition. Up until XE there are no instances of `typeOf` and after XE, there are no instances of ▷.



The functions built with the extension operator are often used in situations where they are applied to two different types in the body of one expression, thus they are less useful without rank-2 types. Hence we add rank-2 types to the type inference algorithm of Chapter 4.

There are two forms of extension we want to capture with ▷, *type preserving transformations* and *accumulators*. In SYB for example, there are two separate functions to achieve the two forms of extension, `extT` and `extQ`. We will show that there is a very simple and natural type inference algorithm that captures both behaviours.

Extension Terminology

The right-hand function in the extension operation can be thought of as either a *fail-over* which prevents errors or a *default* that performs the normal behaviour, depending on the context in which it is used. The left-hand function can be thought of as a *special case* of the default or as a function which can fail and hence needs a fail-over. From this point on we will refer to the left-hand function as the *specific function* and to the right-hand function as the *general function*. When we need to refer to the result of the extension operation, it will be called the *synthesised generic function*.

5.2 Type Preserving Transformations

The `increment` function from our salary update snippet (shown in Listing 5.1) is a type preserving transformation.

Listing 5.1: Part of the salary update snippet

```
1  def incS(amt, s) = case [s] of
2                         { [S(s)]    -> S(s + amt)
3                         } otherwise -> error "partial definition error in incS"
4  def id(x) = x
5  def increment(amt) = incS(amt) ▷ id
```

Its general function is the trivially type preserving function `id` and its specific behaviour is a function which takes in a salary and returns a salary.

5.3 Accumulators

If the general function has a specific return type, we can use ▷ to build up an accumulator. The `check_it` function (shown in Listing 5.2) from the name analysis snippet is an example of an accumulator.

The general function will return a pair of string and boolean and thus the synthesised generic function will also. In this case *two* specific functions are added, the right-most

Listing 5.2: Part of the name analysis snippet

```
1
2 def check_it(strbool)  :: (pair(list(string),bool), a) -> pair(list(string),bool)
3      = check_comm(strbool) ▷ check_intexp(strbool) ▷ fun(a) = strbool
```

Table 5.1: Types of `extT` and `extQ` as reported by GHC version 6.21.1

`extT`	:	`(Typeable a, Typeable b)=> (a -> a)-> (b -> b)-> (a -> a)`
`extQ`	:	`(Typeable a, Typeable b)=> (a -> q)-> (b -> q)-> (a -> q)`

first, each also having a return type of a pair of string and boolean. We can chain together as many of these as we like. Function extension must be right associative for this to work since the general function (the right-hand argument) needs to always be a polymorphic function.

5.4 Deriving a type inference rule for ▷

The function extension operator is a hybrid of the two function extension operations in SYB (`extT` and `extQ`) and the extension typing of the pattern calculus (see Section 3.3.2). In this section we show how to derive a single type rule for ▷ from these existing mechanisms. In fact we derive our rules entirely from the equivalent SYB rules because the constructor focus of the pattern calculus extension typing pushes it too far from our final implementation, which is *very* concerned with what it can know about *types*.

5.4.1 `extT` and `extQ`

We claim that a single operator, ▷ can substitute for the two SYB operators `extT` and `extQ`. Thus ▷ should possess a type rule that works identically to the type rules for the two SYB operations. Table 5.1 shows the types of `extT` and `extQ` reported by GHC version 6.21.1

The *type constraints* `Typeable a, Typeable b` tell us that the general function and the specific function must both take in an argument that is a member of the `Typeable` type class. In SYB, this type class contains the definition of `typeOf`, thus it is equivalent to saying "all types that can have `typeOf` called on them". Putting aside this constraint for now – we will return to it soon – let's see what assuming these typing rules would do to ▷. We will do a derivation for ▷ assuming it has the same type as `extT`, then repeat the process assuming instead it has the type of `extQ`. We will see that the resulting rules (modulo the type constraint above) can be combined. After we have done this we will return to the type constraint and resolve how to emulate it in our rule. Note that the SYB

operations take their arguments in the reverse order to $\triangleright$; we will base the derivation on SYB type rules with the argument types reversed.

Recall that our type inference algorithm $TA \vdash e\colon \tau \text{ mod } V$ takes A (the environment), e (the expression we are inferring the type for) and V (the set of fixed for unification variables) as input and returns T (the set of newly discovered substitutions) and τ (the calculated type for e) as outputs. We will often subscript the returned values (T and τ) to indicate which inference rule they were calculated in. For example, calculating the type of f will result in a set of substitution T_f and a type τ_f.

If we take the existence of a prefix function $\triangleright$ with type $(\alpha \to \alpha) \to (\beta \to \beta) \to (\beta \to \beta)$, i.e. the type of `extT`, as assumptions and otherwise only use our existing type inference rule we get the following type derivation fragment for `((▷f)g)` (i.e. the prefix version of our previously infix $\triangleright$ operator).

$$\dfrac{\dfrac{\dfrac{}{A \vdash \triangleright : (\alpha \to \alpha) \to (\beta \to \beta) \to (\beta \to \beta) \text{ mod } V}\,(\textsc{Assum}) \quad T_f A \vdash f : \tau_f \text{ mod } V \quad T_f((\alpha \to \alpha) \to (\beta \to \beta) \to (\beta \to \beta)) \overset{U_f}{\sim} \tau_f \to \alpha_1 \quad \alpha_1 \text{ new}}{U_f T_f A \vdash \triangleright f : \alpha_1 \text{ mod } V}\,(\textsc{App}) \quad T_g(U_f T_f A) \vdash g : \tau_g \text{ mod } V \quad T_g(\alpha_1) \overset{U_g}{\sim} \tau_g \to \alpha_2 \quad \alpha_2 \text{ new}}{U_g T_g U_f T_f A \vdash ((\triangleright f)\ g) : U_g T_g U_f T_f \alpha_2 \quad \text{mod } V}\,(\textsc{App})$$

There is some redundant information we can remove from this derivation. α_1 must be $(\beta \to \beta) \to (\beta \to \beta)$ and (given α_1 cannot be in τ_f), it is the unification of τ_f and $\alpha \to \alpha$ which gives the substitution U_f. Applying these insights once removes α_1 from the derivation, replacing it with $(\beta \to \beta) \to (\beta \to \beta)$. We can follow the same process a second time to remove α_2, giving the following derivation.

$$\text{(Assum)}\ \frac{}{A \vdash \triangleright : (\alpha \to \alpha) \to (\beta \to \beta) \to (\beta \to \beta) \text{ mod } V}$$

$$\text{(App)}\ \frac{\begin{array}{c} T_f A \vdash f : \tau_f \text{ mod } V \\ T_f((\alpha \to \alpha)) \overset{U_f}{\sim} \tau_f \end{array}}{U_f T_f A \vdash \triangleright f : U_f(\beta \to \beta) \to (\beta \to \beta) \text{ mod } V}$$

$$\text{(App)}\ \frac{U_f T_f A \vdash \triangleright f : U_f(\beta \to \beta) \to (\beta \to \beta) \text{ mod } V \qquad T_g(U_f T_f A) \vdash g : \tau_g \text{ mod } V \qquad T_g U_f T_f(\alpha \to \alpha) \overset{U_g}{\sim} \tau_g}{U_g T_g U_f T_f A \vdash ((\triangleright f)\ g : U_g T_g U_f T_f(\beta \to \beta) \quad \text{mod} \quad V}$$

If we remove our assumptions and all intermediate steps we get the final rule for $\triangleright$ (emulating only `extT`)

$$\text{(extT only)}\ \frac{T_f A \vdash f : \tau_f \text{ mod } V \qquad T_f((\beta \to \beta)) \overset{U_f}{\sim} \tau_f \qquad T_g U_f T_f A \vdash g : \tau_g \qquad T_g U_f(\alpha \to \alpha) \overset{U_g}{\sim} \tau_g}{U_g T_g U_f T_f A \vdash ((\triangleright f)\ g : U_g T_g U_f T_f(\beta \to \beta) \quad \text{mod} \quad V}$$

If we repeat the whole of this process with `extQ` in place of `extT` we get a very similar rule

$$\text{(extQ only)}\ \frac{\begin{array}{c} T_f A \vdash f : \tau_f \text{ mod } V \\ T_f((\beta \to \gamma)) \overset{U_f}{\sim} \tau_f \qquad T_g U_f T_f A \vdash g : \tau_g \text{ mod } V \qquad T_g U_f(\alpha \to \gamma) \overset{U_g}{\sim} \tau_g \end{array}}{U_g T_g U_f T_f A \vdash ((\triangleright f)\ g : U_g T_g U_f T_f(\beta \to \gamma) \quad \text{mod} \quad V}$$

In fact, we are happy to be even more prescriptive about what the type of τ_g can be, by insisting that it must syntactically *be* either $\alpha \to \alpha$ or $\alpha \to \gamma$. This removes a little complexity from our rules and allows us to combine them into one rule. After this change, the only differences between the two rules are what we unify τ_f with and what the final type is. However, in both cases that type is the same as τ_g (if we ensure α is freshened). Thus the following single rule is able to emulate the behaviour of *both* `extT` and `extQ`.

$$\text{($\sim$FunExt)}\ \frac{\begin{array}{c} T_f A \vdash f : \tau_f \text{ mod } V \qquad T_g U_f T_f A \vdash g : \tau_g \text{ mod } V \\ (\alpha \to \gamma) = \tau_g \qquad \tau_g' = [\alpha/\alpha']\tau_g \qquad \alpha' \text{ new} \qquad T_f(\tau_g') \overset{U_f}{\sim} \tau_f \end{array}}{U_g T_g U_f T_f A \vdash ((\triangleright f)\ g : U_g T_g U_f T_f(\tau_g) \quad \text{mod} \quad V}$$

5.4.2 Emulating the Type Constraints

We now consider the type constraints on `extT` and `extQ` which come from the use of Haskell's type classes. The above inference rule is not actually sufficient for our purposes because it ignores those constraints. For example, we could extend one identity function by another identity function and the ∼FunExt type inference algorithm would work. To fully understand why this is a problem we need to jump ahead and look at what $\triangleright$ is compiled into. We cover this more fully in Section 5.8 but for now we can say that, at run time, the input type of each specific function is inspected and that type is checked against incoming data to decide if it is appropriate for this application. If the specific function is the identity there is no canonical type to check against. We store only datatype constructor and built-in type information at run time. There are no type variables in the stored value and arrow types are labelled "unmatchable". This means that the argument to all specific functions must be a non-arrow type with no free variables[1]. In SYB, the fact that there are no arrow types in `Typeable` takes care of the second restriction and the first is enforced by the behaviour of type constraints [88]. Each type constraint is representative of a dictionary of functions. Which dictionary is used in any function is determined by the type that function eventually gets used at. However, if type variables in type constraints can't be resolved into types, the right dictionary can't be chosen. In all instances where a free type variable remains in the type of the specific function, the type constraint for that variable can never be resolved. This occurs because the type variable is left "dangling", in other words it does not occur in either the input or output types of the resulting function, so can't ever be resolved to anything else.

Consider an attempt to extend one identity with another. Using the type for `extT` shown in Table 5.1 we will unify $\alpha \rightarrow \alpha$ (the type of the first identity) to `b -> b` and we will unify $\beta \rightarrow \beta$ (the type of the second identity) to `a -> a`. The result of the extension `` id `extT` id `` is then `Typeable a => a -> a`. The type variable `a` is still able to be resolved later on, but the type variable `b` has been obliterated without ever being resolved to a ground type. Haskell does not allow this and gives an error if we try to compute `` id `extT` id ``.

It is exceptionally neat that the (very general) type class mechanism can fulfil this quite specific requirement. We must encode this into our rule for $\triangleright$. Doing so is not particularly onerous we simply need a side condition that does not allow type variables in the final type of τ_f. Since we have not considered what it means for a specific function to work on

[1]We consider the possibility of relaxing these restrictions in Chapter 10.

Figure 5.1: Type inference rules for $\triangleright$

(FunExt)

$$\frac{\begin{array}{c} T_f A \vdash f : \tau_f \text{ mod } V \qquad T_g(T_f A) \vdash g : \tau_g \text{ mod } V \\ (\alpha \to \gamma) = \tau_g \qquad \tau_g' = [\alpha/\alpha']\tau_g \qquad \alpha' \text{ new} \qquad T_f(\tau_g') \overset{U_f}{\sim} \tau_f \\ U_f \tau_f = \tau_{f1} \to \tau_{f2} \qquad FTV(\tau_{f1}) = \emptyset \qquad \tau_{f1} \text{ does not include arrow types} \end{array}}{A \vdash ((\triangleright f)\ g : T_g U_f T_f(\tau_g) \quad \text{mod} \quad V}$$

arrow types, we disallow occurrences of arrow types anywhere in the type of the input parameter of τ_f. Adding these constraints as side conditions to our $\sim$FunExt rule gives us our final type inference rule for function extension, shown in Figure 5.1.

5.5 Discussion

Rule FunExt infers the type of an extension function that is able to create the two types of polymorphic functions with specific behaviour that we require: *type preserving transformations* and *accumulators*. The type inference algorithm is a minimal common generalisation of the rules for `extT` and `extQ` and we prove the soundness of the underlying type relation in Chapter 9.

By ensuring that the types of each argument to $\triangleright$ must unify, we ensure that a type preserving general function can only be extended with a type-preserving specific function. The type of the general function constrains the type of any specialising function sufficiently to ensure run-time type safety. There is only one type preserving general function, the identity function. This can be successfully extended with any other function that transforms a value to another value *of the same type*.

The type inference algorithm only requires the first argument of the general function to be polymorphic. Thus accumulator general functions will pass through the same rule. The unification of the types of the arguments to $\triangleright$ ensures that any specific function must return the same type as the general function, ensuring that regardless of whether the specific function or the general function is used, the result is the same type.

5.5.1 Comparison to **extQ** and **extT**

The `extT` and `extQ` functions from SYB are more specific versions of $\triangleright$. With $\triangleright$ we can emulate both of these combinators. Thus $\triangleright$ is a more general mechanism and we benefit by having only one operation to compile rather than two. This would be a false economy if

the type inference algorithm for the combined extension operator was very complex, but it is very simple and falls naturally from the desired behaviour of the operator. We have been able to achieve this because we have more control over our type system than SYB does. SYB is restricted to using Haskell's type system which prevents a single operator from having a type flexible enough to do both jobs.

5.5.2 Comparison to Extension Typing

Consider a function that will increment a salary (as an integer), increments any other integer and returns `0` for any other value. Figure 5.3 shows how to encode this function with ▷ and Listing 5.4 shows how to encode it in bondi (using extension typing).

Listing 5.3: A strange salary incrementer with ▷

```
1 def increment(i) =
2     (fun i = case [i] of {[S(s)] -> s + 600} otherwise error "e1")
3   ▷ (fun i = case [i] of {[i] -> i + 600} otherwise error "e2")
4   ▷ (fun x = 0)
```

Listing 5.4: A strange salary incrementer in bondi

```
1 let increment i  =  S(s)  -> s + 600
2                  |  i:Int -> i + 600
3                  |  x     -> 0
```

Leaving aside the syntactic conciseness of the bondi encoding, it looks as if there is a one-to-one translation between the two encodings. However, they actually work differently. The bondi encoding is discriminating on *constructors* while the ▷ encoding is choosing the specific branch to take by inspecting the type of the value to which it is applied. The error clauses in Figure 5.3 cannot ever be triggered because the patterns for each case are exhaustive, there is only one `salary` constructor and the pattern `i` will match on all input. However, if `salary` were to have another constructor, say `W(int)` for wages as in Listing 5.5, an error expression would be the result of applying `increment` to a salary built with that constructor, i.e. `increment(W(5))` would return `error "e1"`. Contrast this with the bondi version which would return `0`. bondi's approach is a neat mechanism, however it brings two downsides with it. Firstly you can't factor your specific functions out of your extension definition. So we can't encode `check_it` in bondi, for example, without having every possible specific function (and the general function) defined within the one

Listing 5.5: Expanded salary datatype

```
1  data salary = S(int)
2              | W(int)
```

function definition.

More importantly for us though, the bondi approach is a significant departure from the semantics of existing functional languages. A function such as that in Listing 5.6 is no longer a function over a single data-type. The presence of a catch-all pattern makes it a function over all data-types. Thus it would have type `a -> int` in bondi, but `salary -> int` in existing functional languages.

Listing 5.6: A strange salary and wage incrementer in bondi

```
1  def increment(i) =  S(s) -> s + 600
2                   |  W(w) -> w + 600
3                   |  x    -> 0
```

These semantics are both fascinating and useful, but they are also too great a departure from, for example the FCP algorithm, for us to claim such changes are "largely orthogonal to the original compiler" as we do in this thesis.

Many functions we can encode this way in bondi can also be encoded with $\triangleright$ but this translation is not completely trivial since we have to convert a demarcation by constructors into a demarcation by types.

5.6 Higher Rank Types

Polymorphic functions with specific behaviour are often used as higher-ranked polymorphic arguments. For example, in the bottom-up generic query (repeated here in Listing 5.7 for convenience), the first argument needs to be applied at two different types in the body of the function. Thus it needs to have a rank-2 type, as the type signature for `generic_query_bu` clearly shows. In practice, it is very likely that it will be a polymorphic function with specific behaviour that will be the actual parameter to match this formal one.

Consequently we need to add higher rank types to our type system before we can actually apply our polymorphic functions with specific behaviour to good effect.

Higher rank types cannot be inferred in a classical Hindley-Milner type system, so

Listing 5.7: A bottom-up (left-to-right) generic query

```
1 def generic_query_bu(f,start,dat) :: (∀ a . (r,a) -> r, r, b) -> r =
2     case [dat] of
3         { [c(z)]    -> f(generic_query_bu(f,generic_query_bu(f,start,z),c),dat)
4         ; [o]       -> f(start,o)
5         } otherwise -> error "partial definition error in generic_query_bu"
```

we must use a certain amount of type annotation to assist the type inference mechanism. The primary difference between the various kinds of higher ranked Hindley-Milner algorithms is the level of annotation required. Less onerous annotation burden is bought at the expense of more complex inference algorithms. We use the type annotation facility that we have in Source and Core to collect this information from the programmer and pass it on to type inference. We ignored these type annotations in Chapter 4, it is here that we show how to deal with them.

Higher rank types are explicitly quantified, and so far annotations can only be monomorphic types. We fix that by modifying the grammar for type annotations in Figures 2.1 and 4.2 (recall that σ denotes universally quantified types, also known as type schemes, and was defined in Figure 4.5).

$$ot = \epsilon \mid \sigma$$

We will derive our final type inference rules for higher-ranked types using only the facilities already in our type system.

5.7 Deriving a Higher Rank Mechanism from FCP

FCP is capable of encoding System F [48] which it does by replacing System F type abstraction with (possibly higher-ranked) data constructors and replacing System F type applications with *selector functions*. A selector function uses a `case` expression[2] to pull arguments from their constructors (i.e. a selector reverses the effect of a constructor). In effect, FCP uses constructors to witness types which are in System F but not in the Hindley-Milner system. Since FCP can encode all of System F, we know we can encode any higher ranked functions in our existing type inference system (Figure 4.5) *if* we are willing to add extra witnessing constructors. In this thesis we are trying to work "with the grain" of common functional compiler techniques and thus would prefer to use *type*

[2]Actually, a pattern matching lambda binding in the original paper. Recall we have replaced these with `case` expressions (page 63).

annotations to witness higher ranked types. In this section we show how to derive higher ranks types with annotations from higher rank types with witnessing constructors (standard FCP). Throughout the derivation we will denote values that we are in the process of calculating as $\circ$. For example, if we part way through a derivation of a type and we don't yet know the final substitution and type, we denote them $\circ$ until their values are known.

Consider a higher ranked lambda abstraction applied to its argument (with annotation)

$$(\lambda(x : \forall\alpha.\exists\beta.\tau).e)\ f$$

This is equivalent to the FCP expression

$$(\texttt{let}\ unK = \lambda(K\ x).x\ \texttt{in}\ \lambda f.[f/(unK\ f)]e)\ (K\ f)$$

if $K : \forall\emptyset.(\forall\alpha.\exists\beta.\tau) \to \tau_K$. Notice that we have an empty set of universally quantified variables in the outer scope. This is because higher-ranks only require generalisation of the inner quantified types. If we quantify all free variables in the equivalent FCP expression we will get more polymorphism than we expect. The constructor K is introduced to witness the higher rank type and it is extracted from such constructed values in a let binding to ensure it is fully generalised in the body of the original function. We can generalise this to definitions of higher ranked functions separate from uses of that function with the following equivalence

$$\lambda(x : \forall\alpha.\exists\beta.\tau).e \equiv \texttt{let}\ unK = \lambda(K\ x).x\ \texttt{in}\ \lambda f.[f/(unK\ f)]e \quad \text{if } K : \forall\emptyset.(\forall\alpha.\exists\beta.\tau) \to \tau_K$$

$$e\ f \equiv e'\ (K\ f) \quad \text{if } e' : \tau_k \to \rho \text{ and } K : \forall\emptyset.(\forall\alpha.\exists\beta.\tau) \to \tau_K$$

Thus the FCP inference rule for $\texttt{let}\ unK = \lambda(K\ x).x\ \texttt{in}\ \lambda f.[f/(unK\ f)]e$ given $K : \forall\emptyset.(\forall\alpha.\exists\beta.\tau) \to \tau_K$ is the same as the rule we want for higher ranked functions witnessed by annotations, and the FCP inference rule for $e\ (K\ f)$ given $e : \tau_k \to \rho$ and $K : \forall\emptyset.(\forall\alpha.\exists\beta.\tau) \to \tau_K$ is the rule we want for higher ranked function application.

The inference rule for $\lambda(x\colon \forall\alpha.\exists\beta.\tau).e$ needs to be derived since it is made up of a number of FCP terms. Lets work backwards from the FCP expression until we reach inference rules that don't include unK or K in their assumptions.

$$\text{(Break)}\ \frac{\begin{array}{c}(x:\tau)\vdash x:\tau \bmod (V\cup\{\beta\})\\ \beta\notin TV(A,\tau,\alpha)\end{array}}{A\vdash \lambda(K\,x).x:\tau_K\to\tau \bmod V}$$

$$\text{(Let)}\ \frac{\sigma = Gen(\tau_K\to\tau)\qquad \circ(unX:\sigma)\vdash \lambda f.[f/(unK\,f)]e:\circ \bmod V}{\circ A\vdash \texttt{let}\ unK=\lambda(K\,x).x\ \texttt{in}\ \lambda f.[f/(unK\,f)]e:\circ \bmod V}$$

At this point it looks like we might have exhausted our options, but some careful reconsideration will allow us to advance. The premise of the Break rule requires that x get the type τ but restricted to those types that do not have β in their free variables. The only way this can happen is if β is the empty set. So we will modify the side condition to say this more clearly. The free variables of $(\tau_K\to\tau)$ *must be* those that were originally bound in α, so we can replace $Gen(\tau_K\to\tau)$ with $\forall\alpha.(\tau_K\to\tau)$ and now we can expand things further

$$\text{(Break)}\ \frac{\begin{array}{c}(x:\tau)\vdash x:\tau \bmod V\\ \beta=\emptyset\end{array}}{A\vdash \lambda(K\,x).x:\tau_K\to\tau \bmod V}$$

$$\text{(Let)}\ \frac{\sigma=\forall\alpha.(\tau_K\to\tau)\qquad \text{(Lam)}\ \dfrac{T_e(unK:\forall\alpha.\tau_K\to\tau, f:\alpha_2)A\vdash [f/(unK\,f)]e:\circ}{T_e(A, unK:\forall\alpha.\tau_K\to\tau)\vdash \lambda f.[f/(unK\,f)]e:\circ}}{T_eA\vdash \texttt{let}\ unK=\lambda(K\,x).x\ \texttt{in}\ \lambda f.[f/(unK\,f)]e:\circ \bmod V}$$

Now have an expression for which there is no FCP inference rule ($[f/(unK\,f)]e$). However, with some careful consideration we can go further. The substitution tells us that all instances of f will become $unK\ f$. We know that unK has type $\forall\alpha.\tau_K\to\tau$ so as long as the type of f (which is α_2) can unify with τ_K we can use τ for the type of $unK\ f$ - i.e. for f in the unmodified e. Well, the type for f *will be* τ_K since our transformation converts all arguments to this function from e' to $K\ e'$. So lets allow ourselves a new rule

$$\text{(Spec)}\ \frac{(unK:\forall\alpha.\tau_K\to\tau, f:\forall\alpha.\tau)A\vdash e:\tau \bmod V}{(unK:\forall\alpha.\tau_K\to\tau, f:\alpha_2)A\vdash [f/(unK\,f)]e:\tau \bmod V}$$

With this new rule, and removing the redundant $\sigma=\forall\alpha.(\tau_K\to\tau)$, we can complete

Figure 5.2: Type inference algorithm for lambda abstractions with type annotations

$$\text{(HigherLam)}\ \frac{T(A, f : \forall\alpha.\tau) \vdash e : \tau_e \bmod V}{TA \vdash \lambda(f : \forall\alpha.\tau)e : \forall\alpha.\tau \to \tau_e \bmod V}$$

our derivation back to expressions that we can otherwise process.

$$\text{(Let)}\ \frac{\text{(Break)}\ \dfrac{(x : \tau) \vdash x : \tau \bmod V \quad \beta = \emptyset}{A \vdash \lambda(K\,x).x : \tau_K \to \tau \bmod V} \qquad \text{(Lam)}\ \dfrac{\text{(Spec)}\ \dfrac{T_e(A, unK : \forall\alpha.(\tau_K \to \tau), f : \forall\alpha.\tau) \vdash e : \tau_e \bmod V}{T_e(A, unK : \forall\alpha.(\tau_K \to \tau), f : \alpha_2) \vdash [f/(unK\,f)]e : \tau_e \bmod V}}{T_e(A, unK : \forall\alpha.(\tau_K \to \tau)) \vdash \lambda f.[f/(unK\,f)]e : \tau_K \to \tau_e \bmod V}}{T_e A \vdash \texttt{let}\ unK = \lambda(K\,x).x\ \texttt{in}\ \lambda f.[f/(unK\,f)]e : \tau_K \to \tau_e \bmod V}$$

If we remove all the intermediate and trivially correct steps we get a simple rule for inferring the type of $\texttt{let}\ unK = \lambda(K\,x).x\ \texttt{in}\ \lambda f.[f/(unK\,f)]e$ under the assumptions that $K : \forall\emptyset, (\forall\alpha, \exists\beta.\tau) \to \tau_K$ and $f : \tau_K$.

$$\text{(Simple)}\ \frac{T(A, unK : \forall\alpha.(\tau_K \to \tau), f : \forall\alpha.\tau) \vdash e : \tau_e \bmod V \qquad \beta = \emptyset}{TA \vdash \texttt{let}\ unK = \lambda(K\,x).x\ \texttt{in}\ \lambda f.[f/(unK\,f)]e : \tau_K \to \tau_e \bmod V}$$

This is a very nice, simple rule which can easily be translated into the rule for the annotation witnessed version. With the introduction of annotations we dispense with unK terms, so we remove that from the environment; τ_K is acting as the proxy for $\forall\alpha.\exists\beta.\tau$, so we replace it with that type, in which β must be empty. Applying these considerations to the above rule gives the final rule for lambda abstractions with a higher-ranked argument, shown in Figure 5.2.

Of course we must also derive the rule for applying such functions to an argument. The derivation of $e\ (K\,f)$ under the constraints that $e : \tau_k \to \rho$ and $K : \forall\emptyset.(\forall\alpha.\exists\beta.\tau) \to \tau_K$ is

Figure 5.3: Type inference algorithm for function application with type annotations

(HigherApp)

$$\frac{T_eA \vdash e\colon (\forall\alpha.\tau) \to \rho \bmod V \qquad T_f(T_eA) \vdash f : \tau_f \bmod V \qquad \tau_f \overset{U_f}{\sim} \tau \bmod V \cup \{\alpha\} \qquad \alpha \notin TV(U_fT_fT_eA)}{UU_fT_fT_eA \vdash e\ f : U_fT_fT_e\rho \bmod V}$$

$$\text{(App)}\ \frac{T_eA \vdash e : \tau_K \to \rho \bmod V \qquad \text{(Make)}\ \dfrac{T_f(T_eA) \vdash f : \tau_f \bmod V \quad \tau_f \overset{U_f}{\sim} \tau \bmod V \cup \{\alpha\} \quad \alpha \notin TV(U_fT_fT_eA)}{U_fT_fT_eA \vdash (K\ f) : U_f\tau_k \bmod V} \qquad U_fT_f(\tau_K \to \rho) \overset{U}{\sim} (\tau_K \to \delta) \qquad \delta \text{ new}}{UU_fT_fT_eA \vdash e\ (K\ f) : U\delta \bmod V}$$

Removing the intermediate steps gives us a single rule for $e\ (K\ f)$, which we call K-App, under the given constraints

(K-App)

$$\frac{T_eA \vdash e : \tau_K \to \rho \bmod V \qquad T_f(T_eA) \vdash f : \tau_f \bmod V \qquad \tau_f \overset{U_f}{\sim} \tau \bmod V \cup \{\alpha\} \qquad \alpha \notin TV(U_fT_fT_eA) \qquad U_fT_f(\tau_K \to \rho) \overset{U}{\sim} (\tau_K \to \delta) \qquad \delta \text{ new}}{UU_fT_fT_eA \vdash e\ (K\ f) : U\delta \bmod V}$$

We can remove the τ_Ks from the unification leaving us with only one τ_K. In this place τ_K is acting as the proxy type for our higher-ranked type so we replace it with that type, giving the rule for $e\ f$ with annotations instead of witnessing types

(Almost)

$$\frac{T_eA \vdash e : (\forall\alpha.\tau) \to \rho \bmod V \qquad T_f(T_eA) \vdash f : \tau_f \bmod V \qquad \tau_f \overset{U_f}{\sim} \tau \bmod V \cup \{\alpha\} \qquad \alpha \notin TV(U_fT_fT_eA) \qquad U_fT_f\rho \overset{U}{\sim} \delta \qquad \delta \text{ new}}{UU_fT_fT_eA \vdash e\ f : U\delta \bmod V}$$

We can further simplify by removing the unification of $U_fT_f\rho$ with the fresh variable δ since that unification can't tell us anything we don't already know and removing β since we know that must be empty. This gives us the final rule for higher ranked function application shown in Figure 5.3.

Figure 5.4: Eliminating the function extension operator from Core

$$\frac{a \equiv e : \tau \to \rho \qquad TV(\tau) = \emptyset}{a \triangleright b \Rightarrow_{xe} \lambda d.\texttt{if}\ (\texttt{typeOf}\, d = \tau)\ \texttt{then}\ a\ d\ \texttt{else}\ b\ d}\ (\textsc{ExtensionElimination})$$

5.7.1 Discussion

Our solution for rank-2 types is extremely simple. In this respect it compares favourably with other systems like the "practical" system of Jones et. al. [50]. That algorithm is more complex but also more general and more useful because it results in better error messages. In our case the simplicity has been achieved by re-using the first-class polymorphism mechanism in FCP. Instead of witnessing higher-rank types with constructors, we witness them with type annotations but use the same mechanisms from that point on. Comparing our algorithm with Leijen's HMF [65] is also interesting. Although that algorithm is able to provide impredicativity, which we don't address here, the mechanisms share some interesting similarities. Both use a restriction on the process of unification to deal with quantified types.

5.8 Translating ▷ to `typeOf`

The description of ▷ leads us directly to its definition

> Function extension, ▷, creates a function which is identical its left-hand argument if applied to a value of the type the left-hand argument is defined over, and identical to its right-hand argument otherwise.

Up to this point we have ignored the fact that Core expressions are all tagged with a type. Before type inference this tag is empty and after type inference it is populated with an inferred type. We denote an expression and its inferred type tag as $e : \tau$. We require an operation, `typeOf`, which can determine the type of a value. Once we have that, the translation (Figure 5.4) is straightforward.

The simplicity of this rule testifies to the neatness of our approach. We use a single operator (▷) to take care of *all* the work needed to extend polymorphic functions with specific cases and compiling it to a lower level form is still exceedingly simple. The type system ensures our side condition is never triggered, but we include it here for clarity.

5.9 Definition of `typeOf`

`typeOf` is a function that extracts a canonical representation of the type of a value from that value at run time. This means that at run time we need;

1. to tag all values with their type, but that type is never used, except by
2. an operator, `typeOf` which can extract the type from a value.

The implementation of `typeOf` is shown in Figure 5.8. All values in the runtime are of type `VAL` and there are numerous macros defined which allow us to extract information from a `VAL`. The `GETTY` macro extracts what kind of value this is; either a constructor (`CON`), one of the built-in types (`INT`, `STRING`, `CHAR`, `BOOL`), or a closure. To support `typeOf`, the definition of `VAL` is supplemented with a new field for type information, making all run-time values slightly larger, and the `TYCON` macro created to extract this string representation of the type of a constructed value. The new built-in operation, `typeOf`, is created to pull this type from any value. Both these tasks are very easy in our run time system, Epic, and we expect they are relatively easy in many other run-times. The value extracted with `typeOf` will be checked for equality with the input type of the specific function, so there cannot be any type variables in that type. If there were, we would need something more complicated than mere equality. Happily, this restriction is assured by the type system. Note that this is naive encoding used for clarity and widespread applicability. We have chosen an unoptimised approach because we want our benchmarks of this system (see Chapter 8) to be indicative of the maximum cost that would be encountered when these techniques are use to extend existing compilers.

Listing 5.8: C definition of `typeOf`

```
1  void* typeOf(VAL one){
2      switch(GETTY(one)){ // GETTY will get the type field of a VAL
3          case CON:
4            return MKSTR(TYCON(one));
5          case INT:
6            return MKSTR("#int");
7          case STRING:
8            return MKSTR("#string");
9          case CHAR:
10           return MKSTR("#char");
11         case BOOL:
12           return MKSTR("#bool");
13         default:
14         return MKSTR("#unmatchable");
15     }
16     assert(0); // we should not have been given any other type of closure
17 }
```

`typeOf` is not a SOURCE operation and can only be created in the compilation of ▷. Thus runtime types are a space cost for all code, but only a time cost for code that uses extensions. As we will show below, this time-cost is unavoidable and the memory cost is minimal.

5.9.1 Discussion of the Semantics of `typeOf`

By-passing static type checks can only be safely done if there is some equivalent to, for example, Java's `instanceOf` which allows one to inspect the run-time type of a value. This is an identical mechanism to `typeOf` but making it available in the source language means it can appear anywhere. Type-safe casting can be done in Haskell98 (which has no run-time type information) with type classes, but these replace the type information with a dictionary of functions.

5.9.2 Discussion of Time and Space Cost

Of the approaches to polymorphic functions with specific behaviour that we saw in Chapter 3 only explicit failure and extension types do not use type classes or run-time types. Instead they require per-program unique constructor names because you need to be able to check any two values from the whole program to see if they have the same constructor. In traditional functional languages, the run-time only needs to differentiate between two constructors of the *same type*. We now explain why per-program constructors will have an equivalent memory cost and a greater run-time cost than the solution we are suggesting[3]. To store a constructor and a type, as we do, requires the minimum bits to distinguish this constructor from the others in its type plus a longer, per-program unique, identifier for the type, perhaps using namespaces to ensure uniqueness in the presence of separate compilation. The alternative of storing the per-program unique constructor requires that the constructors themselves use the longer form, again using something like namespaces if necessary. This long form must be at least as long as the form you would use to store just the pre-program unique *type*. Hence one can, at most, save a few bits of memory by not storing the constructor separately. However, constructor analysis will now need to be done on a large value (the per-program unique constructor) instead of a small value (a few bits for the constructor). When we compile to C for example, using per-program unique constructors means we can't compile our `case` expressions into `switch` statements. This

[3]Unfortunately there are no system on which we can run empirical tests to show this, so we must content ourselves with a theoretical investigation.

will potentially cost huge amounts of run-time[4].

We have not had the opportunity to benchmark all the relevant approaches to storing type information at run-time, it is far beyond the scope of this work. However, we we can see that, without type classes, some run-time cost needs to be paid either by keeping type information at run-time or by moving to per-program unique constructors. In this way we have argued that adding type information to each value and a function to inspect it is at least competitive with alternative techniques. In fact, it is likely to be the most efficient since we are using the run-time representation, keeping exactly what we need, and nothing more, to do the job.

5.10 Summary

We have shown that $\triangleright$ can be compiled into a simple primitive operation. We have shown a small extension of Hindley-Milner which can infer a type for this operator and we have shown that implementing `typeOf` requires only modest changes to the language runtime.

With $\triangleright$ we have achieved an version of function extension which, unlike SYB's version, does not need type-safe casts or type classes. We will later show that it is also significantly faster. We have also achieved a version of extension typing that is compatible with existing functional languages. It does not require any new types and does not change the existing semantics or the inferred type of existing code. We also demonstrated a simple way to add higher-rank types to a baseline functional compiler using only existing FCP features.

[4]We say *potentially* because we have not been able to test this empirically.

Chapter 6

Compiling and Typing Structure Agnosticism

Recall that in Chapter 3 (page 45) we described how structure agnosticism can be achieved with an explicit spine view of data such that:

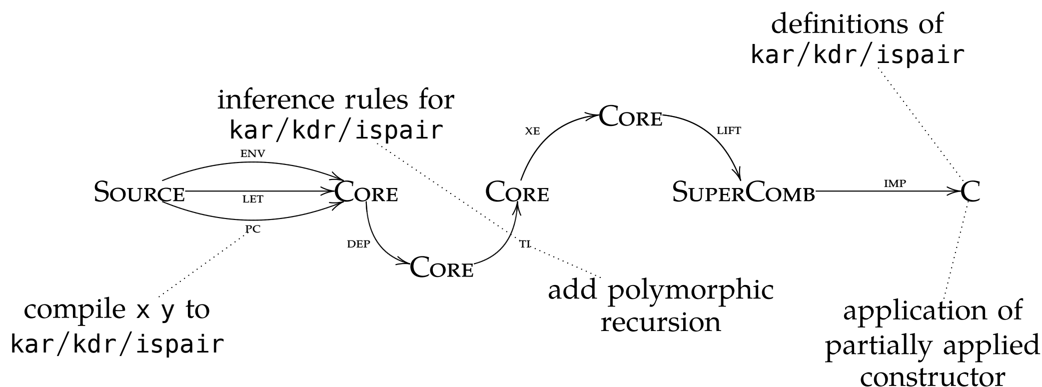
1. All data is a tuple of: the constructor and all the arguments but the last, and the last argument to the constructor.
2. Tuples can be pulled apart by *application pattern matches* which are `case` patterns `x(y)` where `x` is bound to the constructor and all arguments but the last and `y` is bound to the last argument.

The examples in Chapter 2 show how this explicit spine view of data can encode generic functions. In this chapter we show how we can remove these patterns in the pattern compilation translation, converting them to simple primitive operations. We also show how to attribute types to these primitive operations and how to encode them in C. We include an implementation of polymorphic recursion since none of our generic examples can work without it.

The types we attribute to `x` and `y` require no underlying extensions to FCP, only careful use of existing features. Thus this chapter shows exactly how application pattern matches can be incorporated into a baseline functional compiler. Application pattern matches also give an identity to partially applied constructors and in this chapter we describe the small extensions to the run-time which are required to take advantage of this. Note that we restrict ourselves to a very small set of additions that don't amount to making partially applied constructors first-class values, but which provide enough capability to support our generic functions.

6.1 Overall Journey

We describe how to resolve application pattern matches into simple, typed language primitives, `kar/kdr/ispair`. Application pattern matches are already present in SOURCE although we ignored their existence in Chapter 4. We use the pattern compilation translation to compile these into new primitive operations, thus application pattern matches are not present in CORE.



However, these new primitive operations *are* in CORE, and we need to describe the type inference rules for them. Furthermore, application pattern matches are very commonly used in conjunction with polymorphic recursion, so we modify our type inference algorithm to support this feature. Extension elimination and lambda lifting are not effected by these new primitives, we only need to define their operation at run-time with new C functions.

Application pattern matching can give an identity to partially applied constructors. Our type rules ensure that they are always re-applied to their missing arguments but this does not prevent us from requiring an implementation of this re-application in the run-time.

6.2 An Unresolved SOURCE Detail

The discussion of pattern compilation presented in Chapter 4 is complete in all but one respect. Specifically it fails to account for the compilation of one significant feature of the pattern system of SOURCE, that being the *application patterns* which we introduced on page 22.

$$p ::= \cdots$$
$$\mid \quad x(y)$$

It is important to note that in order to implement the compilation of application patterns we make no changes to the underlying analysis and translation of the `case` expressions of Core themselves. In particular they maintain all the characteristics that made them simple to convert into `switch` statements.

6.3 Converting `x(y)` to `kar`, `kdr` and `ispair`

As we saw in Chapter 4 (page 69), pattern matching is compiled to `switch` statements, whose chief characteristic is that they branch on the integer representing the constructor of the input. An `x(y)` pattern cannot be compiled directly into this form simply because it provides the compiler with no specific constructor information. While the variable `x` may become bound to some constructor it may also become bound to a compound subexpression (as shown in Table 3.4). Furthermore, even if it is bound to a constructor, this pattern alone will not determine the identity of that constructor. For this reason, `x(y)` patterns are not compiled into Core `case` expressions. They are instead compiled into three primitive operations which suffice to describe their semantics. These primitives are:

`kar` *d* which retrieves the left part of the datum `d` when the datum is viewed as a tuple, and

`kdr` *d* which retrieves the right part of the datum `d` when the datum is viewed as a tuple, and

`ispair` *d t e* which evaluates to the expression `t` if the datum `d` is a tuple and evaluates to the expression `e` if it is not.

This approach to encoding the spine view is well known in untyped functional languages, indeed we use the name `kar` for pulling off the front of a value since this operation is very much like the lisp `car` operation. Similarly, we name `kdr` for `cdr`. However, statically inferring types for expressions like these is not done in any languages we are aware of. Barry Jay [43] has shown that `x(y)` patterns can be encoded with statically typed versions of the `kar/kdr/ispair` primitives and our solution is an extension of his work. We

have been able to achieve simpler and more practical type rules than Jay because our compiler pipeline maintains a very useful invariant, that `kar` and `kdr` can only appear within a witnessing `ispair` expression.

To implement this functionality we add the following primitive functions to those available in CORE

$$
\begin{aligned}
lo ::= &\ \cdots \\
\mid &\ \texttt{kar}\ e \\
\mid &\ \texttt{kdr}\ e \\
\mid &\ \texttt{ispair}\ e\ e\ e \\
\mid &\ \cdots
\end{aligned}
$$

The actual conversion from `x(y)` patterns to these new built-in operations occurs in the pattern compilation translation phase ($\Rightarrow_{pc}$). The addition of these primitives and the associated rules for compiling application pattern matches does not require us to change any of the existing pattern compilation rules discussed in Chapter 4. So, in principle, all we need to do in order to process application patterns is to add a single new rule to our pattern compilation algorithm which processes these new patterns to `kar`/`kdr`/`ispair` expressions. However, before we do that we should ask ourselves two questions:

1. Is it better to process such patterns in batches or on their own?
2. Exactly what expressions should we generate when we process each batch of application patterns?

Batch or Singleton?

Application pattern matches can be compiled in batches like variables and constructors. Thus we need to tell our pattern compilation algorithm to slurp up as many as it can from the top of our set of patterns when applying the HETEROGENOUS rule (page 59).

In fact, any time there are more than one in a row, all but the first are ignored. It is impossible to get a failure on one application pattern match but a success on a later one since the only way they can vary is in the names of the variables they bind. By processing them in batches, the pattern compilation algorithm removes dead code for us, with no extra effort. If we break them up into singletons then we will end up generating nested `ispairs` within which it is much harder to find such dead code. Indeed, were we to do

Figure 6.1: Pattern compilation for application pattern matches

$$\frac{\begin{array}{c}(\textsc{Application Patterns})\\ \forall i.(alt_i \equiv x_i(y_i)\ \overline{sp_i} \rightarrow se_i \wedge alt'_i \equiv \overline{sp_i} \rightarrow [x_i/\texttt{kar}(x_1), y_i/\texttt{kdr}(x_1)]se_i)\\ \texttt{case}\ \overline{z}\ \texttt{of}\ \overline{alt'}\ \texttt{otherwise}\ d \Rightarrow_{pc} e' \qquad d \Rightarrow_{pc} d'\end{array}}{\texttt{case}\ z_1\ \overline{z}\ \texttt{of}\ \overline{alt}\ \texttt{otherwise}\ d \Rightarrow_{pc} \texttt{ispair}\ z_1\ e'\ d'}$$

this, we would probably need to introduce a separate phase simply to analyse and remove this superfluous code.

To support batch processing of these patterns, we need to modify the definition of pattern classes ($\overline{p} \equiv_c \overline{p'}$) we gave in Section 4.1. So we extend the relation definition in equation 4.1 (page 58) to the one given by the following set of equations, in which the last line is new.

$$\begin{aligned} x\ p_2 \cdots p_n &\equiv_c x'\ p'_2 \cdots p'_n \\ (K\ \overline{p})\ p_2 \cdots p_n &\equiv_c (K'\ \overline{p'})\ p'_2 \cdots p'_n \\ x(y)\ p_2 \cdots p_n &\equiv_c x'(y')\ p'_2 \cdots p'_n \end{aligned}$$

This indirectly changes the definition of the Heterogeneous rule (page 59), but this is the *only* change to the rules in the original pattern compilation algorithm we described in Chapter 4.

Figure 6.1 shows the pattern compilation algorithm which applies when all head patterns are application pattern matches. All application pattern matches in the head pattern resolve to a single `ispair` check. If this succeeds, then the remaining `case` (with one less scrutinee and one less pattern in each alternative), where x_i is replaced by `kar` of the scrutinee and y_i is replaced by `kdr` of the scrutinee, is the result. If it fails, the default is the result.

Notice that this compilation scheme results in the `kar`/`kdr` of a matched term being re-evaluated at every point where the variables bound in the application pattern are used. If might be better to replace the term $[x_i/\texttt{kar}(x_1), y_i/\texttt{kdr}(x_1)]se_i$ with $\texttt{let}\ xkar = \texttt{kar}(x_1)\texttt{and}\ xkdr = \texttt{kdr}(x_1)\ \texttt{in}\ [x_i/xkar, y_i/xkdr]se_i$. However, we have not implemented this in DGEN so the rule in Figure 6.1 is a more accurate representation of what DGEN actually does. Furthermore, none of the examples in this thesis would benefit greatly from the change. We can see that DGEN is quite naive with regards to optimising the generated executable, something we discuss more fully in Chapter 8.

So, we have managed to completely remove `x(y)` patterns in pattern compilation, replacing them with three carefully chosen primitives. Lets now see how to get these primitives through the rest of the compiler, eventually formalising their behaviour in C. We will start off looking at the type rules for `kar`, `kdr` and `ispair` and then see why polymorphic recursion is necessary and how to implement it.

6.4 Type Inference Rules for kar, kdr and ispair

Our starting point is the following question

> Given `x` has type `X`, and assuming `x` is a tuple, what are the types of `kar(x)` and `kdr(x)`?

If `x` is a tuple, it is a constructor applied to some arguments, `kar(x)` is the constructor with all the arguments but the last, so it could have the type of a function that if given that last argument, will return a value of the original type, i.e. `argtype -> X`. `kdr(x)` is exactly the argument that was peeled off, so its type is `argtype`. However, we don't know anything about `argtype`. Since we only know `x` has type `X`, we can say nothing about its last argument, we only know that there is one. We might be tempted then to make it a variable, giving `kar x` the type α`-> X` and `kdr(x)` the type α. However, this variable would be implicitly universally quantified and since `kar(x)` and `kdr(x)` will appear separately, we lose the fact that the two αs represent the same *something*. The solution to this problem is existential types.

The addition of pattern matching for application patterns amounts to proposing the existence of the retraction function $break_\tau$ for all types τ where ($\tau \mid \rho$ is notation for a sum type with τ "in-left" and ρ "in-right")

$$break_\tau : \tau \to (\exists \rho.(\rho \to \tau, \rho) \mid \tau)$$

$$make_\tau : (\exists \rho.(\rho \to \tau, \rho) \mid \tau) \to \tau$$

$$make_\tau \; (break_\tau \; e) = e$$

In other words, we propose that, for every type τ, we have some operation $break_\tau$ which can separate compound values of that type into two parts and which returns atoms unchanged. Furthermore, for each type τ, there is a function $make_\tau$ which can put the separated parts back together. We cannot claim the other half of the *make*/*break* isomorphism, i.e. we can't say that

$$break_\tau \; (make_\tau \; e) = e$$

because the choice of existentially quantified type is lost by $make_\tau$ and cannot be recovered. In functional programming languages, $make_\tau$ is a function we can build for any type since its definition is fixed for all types. For the left-hand side of the input sum-type, $make_\tau$ takes a triple $(\rho, f : \rho \to \tau, g : \rho)$ and its body is always $f\ g$. For the right-hand side of the input sum type, $make_\tau$ is the identity function. Function application takes the role of $make_\tau$ in DGEN, though a special syntax is often used to aid parsing it (see page 17). By re-using an existing mechanism we reduce the distance between existing functional language compilers and the solution we present in this thesis, making our techniques easier to implement.

The definition of $break_\tau$ can be informally stated using `ispair`, `kar` and `kdr` (where *inl* injects a value into the left-hand side of a sum-type and *inr* injects a value into the right-hand side of a sum-type)

$$break_\tau\ x = \texttt{ispair}\ x\ (inl(\rho, \texttt{kar}(x), \texttt{kdr}(x)))\ (inr(x)) \quad \text{where } \rho \text{ is unique}$$

We will show in Section 6.6 how to write `ispair`, `kar` and `kdr` in our run-time, i.e. how to write one definition for each primitive that works on all τ. All that is left is to devise a type inference algorithm for these primitives which reflects our observations about $break_\tau$.

Previous attempts to give types to `kar` and `kdr` in this context, for example the compound calculus [43], have failed because `kar` and `kdr` are language primitives in those systems. We are in a better situation. The `kar` and `kdr` primitives *are not* language primitives which can appear anywhere, they are the result of pattern compilation. Our pattern compilation algorithm ensures that *all instances of `kar` and `kdr` are inside the left branch of an `ispair` expression*. Since every instance of `kar` or `kdr` is associated with a specific enclosing `ispair`, this latter construct becomes the ideal site at which to introduce a fresh existentially quantified type variable to constrain the use of these unpairing operations.

The addition of existentially quantified variables to an existing functional language compiler might seem like it would require significant type system changes. In our case we are using a variant of FCP which already comes equipped with existential type quantification. However, it turns out that we can provide typing rules for `kar`/`kdr`/`ispair` without having to provide a complete implementation of existential typing. Specifically, all we need to do is ensure that any new existential variables introduced by `ispair` are treated like constants for the purposes of unification. To do this we need to keep track of which variables in the environment must be handled in this way and to use the simple

Figure 6.2: Type Inference for `ispair`

(IsPair)

$$\frac{TA \vdash c\colon \tau_c \text{ mod } V \qquad T'T(A, c : (\beta \to \tau_c, \beta) \vdash t\colon \tau_t \text{ mod } (V \cup \beta) \qquad T''T'TA \vdash e\colon \tau_e \text{ mod } V \qquad \tau_t \overset{U}{\sim} \tau_e \text{ mod } V \qquad \beta \text{ new} \qquad \beta \notin TV(UTA, U\tau_t)}{UT''T'TA \vdash \texttt{ispair}\ c\ \texttt{then}\ t\ \texttt{else}\ e\colon U\tau_e \text{ mod } V}$$

variant of the usual unification algorithm deployed by Jones in his FCP paper [48].

$$\tau \overset{id}{\sim} \tau \text{ mod } V \tag{id}$$

$$\left.\begin{array}{l} \alpha \overset{[\tau/\alpha]}{\sim} \tau \text{ mod } V \\ \tau \overset{[\tau/\alpha]}{\sim} \alpha \text{ mod } V \end{array}\right\} \qquad \alpha \notin V \ \cup \ TV(\tau) \tag{var}$$

$$\frac{\tau \overset{U}{\sim} v \text{ mod } V \qquad U\tau' \overset{U'}{\sim} Uv' \text{ mod } V}{(\tau \to \tau') \overset{UU'}{\sim} (v \to v') \text{ mod } V} \tag{fun}$$

This version of unification reduces to normal unification when the set V is empty, so only a few extra empty sets are needed in other parts of the compiler.

Figure 6.2 shows the type inference rule for `ispair`. This `ispair` rule first calculates the type of the expression being tested (c), this is used as the basis for the type of the conditional in the first branch. We store two types for c in the environment. The first ($\beta \to \tau_c$) is the type for `kar(c)` anywhere in the body of the branch and the second (β) is the type for `kdr(c)`. When we are typing the first branch, we ensure that β is treated as an existential type variable by adding it to the set of fixed for unification variables. In this environment we calculate the type for the first branch. This is all the hard work done and we use standard techniques to get the type for the second branch and to unify that type with the type we got for the first branch.

Notice that when we add the new mapping for `c` to the environment, we leave the old one in. The new mapping is to a *pair of types* and so can be distinguished from the other mapping, which is to a single type scheme. Only the type rules for `kar` and `kdr` will ever go looking for this pair. Note also that `c` needs to be a variable if we are to put *anything* in the environment for it. You will notice that Core requires scrutinees to be variables and since the conditional of the `ispair` always comes from such a scrutinee, it *will* always be a variable. The restriction of scrutinees to variables in Core is there for other reasons, for example it allows scrutinees to be let-bound to avoid recalculating them, so this technique is likely to work in many core languages. We have had to extend the capabilities of our type

Figure 6.3: Type Inference Rules for `kar` and `kdr`

(Kar)
$$\frac{(x\colon(\beta\to\tau,\beta))\in A}{A\vdash \texttt{kar}(x)\colon\beta\to\tau \bmod V}$$

(Kdr)
$$\frac{(x\colon(\beta\to\tau,\beta))\in A}{A\vdash \texttt{kdr}(x)\colon\beta \bmod V}$$

environment, but the changes are minor. Previously, variables mapped to type schemes, now they map to a type scheme and an optional pair of type schemes which is completely ignored in all other type inference rules.

With this done, the type inference rules for `kar` and `kdr` are very simple (Figure 6.3). Both these rules just look in the type environment for the type that the `ispair` rule left for them. The type variable β is already in the set of fixed for unification variables.

These type inference rule ensure that the existentially quantified type variable only exists within the first branch of an `ispair` operation. Because the existentially quantified type variable can't be accessed in the second branch of an `ispair` and the types of the two branches need to be unified, there is no way the existentially quantified type variable can escape the scope of its quantification.

6.5 Polymorphic Recursion

Although we have now done enough to ensure appropriate types for `kar` and `kdr`, we are still in no position to do much with these operations. The vast bulk of generic traversals do their work by calling themselves recursively on the `kar` or `kdr` (or both) of their inputs. To see how this works in practice consider the bottom-up generic traversal we saw in Chapter 2.

Listing 6.1: A bottom-up generic traversal

```
1 def apply_to_all(f,g) :: (∀ a . (a) -> a, b) -> b =
2     case [g] of
3          { [c(a)]    -> f(@apply_to_all(f,c)(apply_to_all(f,a)))
4          ; [o]       -> f(o)
5          } otherwise -> error "partial definition error in apply_to_all"
```

The first recursive call to `apply_to_all` instantiates the type variable `a` to `b -> a'` while the second recursive call instantiates it to `b`. Thus we are recursively calling a function at different types in its own body. This is the purest form of polymorphic recursion and one that dependency analysis has no hope of removing for us. Thus we need to add support for polymorphic recursion to our type system.

As we mentioned in Section 4.2, while it *can* be possible to infer the type of a polymorphic recursive function, it is not *always* possible. Pure un-annotated type inference is formally undecidable for a language that requires polymorphic recursion. The most commonly used solution, and the one we will adopt, is to require type-annotations on polymorphic recursive functions [49], and to use these annotations to guide the inference. We used the same technique for rank-2 types in Section 5.6. Since we already have type annotations on terms from our work in Section 5.6, we are left with the job of using these annotations effectively for `letrec`-bound expressions.

Type inference for polymorphic recursion thus amounts to type inference for the following form of `letrec` expression

$$\texttt{letrec}\ x_1\ ot_1 = e_1\ \texttt{in}\ e$$

where ot designates that the type annotation is optional. This means that each binding is either x_1 or $x_1\ \sigma$.

6.5.1 Deriving Polymorphic Recursion from FCP

FCP allows first class polymorphism for constructor arguments, witnessing all higher ranks by equivalent constructors (page 85). Since Jones describes a mapping from System F types (which includes all polymorphic recursive types) to FCP [48], we must be able to encode polymorphic recursion in our existing type system if we are willing to use witnessing constructors. As was the case with higher ranked types, our starting point of a baseline functional compiler encourages us to work with the grain of current functional languages. Thus we want polymorphic recursion using *type annotations* instead of witnessing constructors. In this section we explain how to achieve polymorphic recursion using type annotations from FCP's polymorphic recursion with witnessing constructors.

Our first job is to take the DGEN/CORE expression (with type annotations) which we wish to infer the type of and to formulate an equivalent FCP (with witnessing construc-

tors) expression. For single-binding `letrec`, our task is to calculate the type of

$$\texttt{letrec}\ x \colon \forall\alpha.\exists\beta.\tau = e\ \texttt{in}\ f \tag{6.1}$$

where x may occur free in e and f. The equivalent FCP expression witnesses the type $\forall\alpha.\exists\beta.\tau$ with a constructor K and extracts it with a selector unK. Informally speaking, the corresponding FCP expression eliminates the need for type annotations by $\tau' = \forall\alpha.\exists\beta.\tau$ by replacing *definitions* of expressions e with type τ' with $K\ e$ and replaces *uses* of expressions of this type x with $unK\ x$. We define the constructor K in the standard way for FCP, in other words we give it the type $\sigma_K = \forall\gamma.(\forall\alpha.\exists\beta.\tau) \to \tau^K$ with new α, β, and γ. We need to include the definition of the selector function unK in a let-binding to ensure the unpacked value is given a polymorphic type. An FCP expression using witnessing constructors which is equivalent to (6.1) is

$$\texttt{let}\ unK = \lambda(K\ f).f\ \texttt{in}\ \texttt{letrec}\ x' = K\ ([x/unK\ x']e)\ \texttt{in}\ [x/unK\ x']f \tag{6.2}$$

Now our approach will be to apply the FCP type inference algorithm (from the original FCP publication [48]) to this term and elaborate its operation by hand just far enough to eliminate all references to K and unK in the assumptions at the leaves of the resulting inference proof. Once we have done that, we can summarise that entire calculation in the form of a corresponding single inference rule couched entirely in term of the type annotation expression (6.1). In other words, we can calculate a type inference rule for the type annotation expression (6.1) using only the rules required for the constructor witnessing expression (6.2).

This derivation is far too large to fit into one diagram, instead we need to look at small parts of the derivation tree at any one time. We will label all calculations which require exposition with a double-stroke character. Where there are labeled calculations in the conclusion of an inference rule, it means that entire inference rule should be substituted for the labelled calculation in a larger inference calculation.

We first apply the Let rule to remove the outermost `let` binding.

$$(\text{Let})\ \frac{\begin{array}{c}(\mathbb{A})\ T_{unK}A \vdash \lambda(K\ f).f \colon \tau_{unK}\ \text{mod}\ V \qquad \sigma_{unK} = Gen(T_{unK}A, \tau_{unK}) \\ (\mathbb{B})\ T_i T_{unK} A_{unK}, unK \colon \sigma_{unK} \vdash \texttt{letrec}\ x' = K\ ([x/unK\ x']e)\ \texttt{in}\ [x/unK\ x']f \colon \tau_i\ \text{mod}\ V\end{array}}{\begin{array}{l}T_i T_{unK} A \vdash \texttt{let}\ unK = \lambda(K\ f).f \\ \qquad\qquad \texttt{in}\ \texttt{letrec}\ x' = K\ ([x/unK\ x']e)\ \texttt{in}\ [x/unK\ x']f \colon \tau_i\ \text{mod}\ V\end{array}}$$

We proceed by trying to calculate all the substitutions (T, U) and types (τ, ρ) in the premises. Firstly, we need to calculate T_{unK} and τ_{unK} via $(\mathbb{A})$.

$$\text{(Break)}\ \frac{\sigma_K = \forall\gamma_1.(\forall\alpha_1.\exists\beta_1.\tau_1) \to \tau_1^K \qquad A_f, f\colon \tau_1 \vdash f\colon \tau_1 \text{ mod } V \cup \{\beta_1\} \qquad \beta_1 \notin TV(A, \tau_1, \alpha')}{(\mathbb{A})\ A \vdash \lambda(K\, f).f\colon \tau_1^K \to \tau_1 \text{ mod } V}$$

The only way to satisfy $\beta_1 \notin TV(A, \tau_1, \alpha')$ is if β_1 is empty (which only happens if β is empty). The other two assumptions in this rule are trivially true, so the result of this calculation are that the modified side condition $\beta = \emptyset$ plus the results, $T_{unK} = \emptyset$ and $\tau_{unK} = \tau_1^K \to \tau_1$, replace $(\mathbb{A})$.

$$\text{(Let)}\ \frac{\beta = \emptyset \qquad \sigma_{unK} = Gen(A, \tau_1^K \to \tau_1) \\ (\mathbb{B})\ T_i A_{unK}, unK\colon \sigma_{unK} \vdash \texttt{letrec}\ x' = K\ ([x/unK\ x']e)\ \texttt{in}\ [x/unK\ x']f\colon \tau_i \text{ mod } V}{T_i A \vdash \texttt{let}\ unK = \lambda(K\, f).f\ \texttt{in}\ \texttt{letrec}\ x' = K\ ([x/unK\ x']e)\ \texttt{in}\ [x/unK\ x']f\colon \tau_i \text{ mod } V}$$

We know that $Gen(A, \tau_1^K \to \tau_1)$ must be $\forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1)$, so we can replace σ_{unK} with that type.

$$\text{(Let)}\ \frac{\beta = \emptyset \\ (\mathbb{B})\ T_i A_{unK}, unK\colon \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1) \vdash \texttt{letrec}\ x' = K\ ([x/unK\ x']e)\ \texttt{in}\ [x/unK\ x']f\colon \tau_i \text{ mod } V}{T_i A \vdash \texttt{let}\ unK = \lambda(K\, f).f\ \texttt{in}\ \texttt{letrec}\ x' = K\ ([x/unK\ x']e)\ \texttt{in}\ [x/unK\ x']f\colon \tau_i \text{ mod } V}$$

We now calculate the values of T_i and τ_i via $(\mathbb{B})$. This requires an application of the LetRec rule.

$$\text{(LetRec)}\ \frac{(\mathbb{C})\ T_{x'} A_{unK,x}, unK :: \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1), x'\colon \alpha_2 \vdash K\ [x/unK\ x']e\colon \tau_{x'} \text{ mod } V \\ T_{x'}\alpha_2 \overset{U_{x'}}{\sim} \tau_{x'} \qquad \sigma_{x'} = Gen(T_{x'}A, U_{x'}\tau_{x'}) \\ (\mathbb{D})\ T_f U_{x'} T_{x'} A_{unK,x}, unK\colon \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1), x'\colon \sigma \vdash [x/unK\ x']f\colon \tau_f \text{ mod } V}{(\mathbb{B})\ \begin{array}{l} T'UTA_{unK}, unK\colon \forall\gamma_1.(\forall\alpha_1.\tau_1) \to \tau_1^K \\ \qquad \vdash \texttt{letrec}\ x' = K\ ([x/unK\ x']e)\ \texttt{in}\ [x/unK\ x']f\colon \tau_i \text{ mod } V \end{array}}$$

Again we need to calculate the premises individually. Lets first calculate $T_{x'}$ and $\tau_{x'}$ via

$(\mathbb{C})$.

$$\text{(Make)}\ \frac{\begin{array}{c}\sigma_K = \forall\gamma_2.(\forall\alpha_3.\tau_3) \to \tau_3^K \\ (\mathbb{E})\ T_e A_{unK,x}, unK\colon \forall\gamma_1.(\forall\alpha_1.\tau_1) \to \tau_1^K, x'\colon \alpha_2 \vdash [x/unK\ x']e\colon \tau_e \bmod V \\ \tau_e \overset{U_e}{\sim} \tau_2 \bmod V \cup \{\alpha_3\} \qquad \alpha_3 \notin TV(U_e T_e A_{unK}, unK\colon \forall\gamma_1.(\forall\alpha_1.\tau_1) \to \tau_1^K)\end{array}}{(\mathbb{C})\ T_{x'} A_{unK,x}, unK :: \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1), x'\colon \alpha_2 \vdash K\ [x/unK\ x']e\colon \tau_{x'} \bmod V}$$

We use the Spec rule we created for deriving higher rank types (page 87) to calculate $(\mathbb{E})$

$$\text{(Spec)}\ \frac{T_e A_{unK,x}, unK\colon \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1), x\colon \forall\alpha_4,\gamma_4.\tau_4 \vdash e\colon \tau_e \bmod V}{(\mathbb{E})\ [\alpha_2/\tau_4] T_e A_{unK}, unK\colon \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1) \vdash [x/unK\ x']e\colon \tau_e \bmod V}$$

We have converted $(\mathbb{E})$ into a premise without K or unK in the expression for which we are calculating the type. We replace $(\mathbb{E})$ with this premise and propagate the calculated values to the Make rule in which $(\mathbb{E})$ is a premise.

$$\text{(Make)}\ \frac{\begin{array}{c}\sigma_K = \forall\gamma_2.(\forall\alpha_3.\tau_3) \to \tau_3^K \\ [\alpha_2/\tau_4] T_e A_{unK,x}, unK\colon \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1), x\colon \forall\alpha_4,\gamma_4.\tau_4 \vdash e\colon \tau_e \bmod V \\ \tau_e \overset{U_e}{\sim} \tau_2 \bmod V \cup \{\alpha_3\} \qquad \alpha_3 \notin TV(U_e T_e A_{unK}, unK\colon \forall\gamma_1.(\forall\alpha_1.\tau_1) \to \tau_1^K)\end{array}}{(\mathbb{C})\ T_{x'} A_{unK,x'}, unK\colon \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1), x'\colon \alpha_2 \vdash K\ [x/unK\ x']e\colon \tau_3^K \bmod V}$$

We can simplify this a little since we know unK can't be in e and α_3 can't be in $\forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1)$. Plus we can calculate the value of $T_{x'}$

$$\text{(Make)}\ \frac{\begin{array}{c}\sigma_K = \forall\gamma_2.(\forall\alpha_3.\tau_3) \to \tau_3^K \qquad [\alpha_2/\tau_4] T_e A_x, x\colon \forall\alpha_4,\gamma_4.\tau_4 \vdash e\colon \tau_e \bmod V \\ \tau_e \overset{U_e}{\sim} \tau_2 \bmod V \cup \{\alpha_3\} \qquad \alpha_3 \notin TV(U_e T_e A)\end{array}}{(\mathbb{C})\ [\alpha_2/\tau_4^K] T_e U_e A_{unK,x'}, unK\colon \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1), x'\colon \alpha_2 \vdash K\ [x/unK\ x']e\colon \tau_3^K \bmod V}$$

The unresolved premises of this rule now replace $(\mathbb{C})$ in the LetRec rule

$$\text{(LetRec)}\ \frac{\begin{array}{c}\sigma_K = \forall\gamma_2.(\forall\alpha_3.\tau_3) \to \tau_3^K \\ [\alpha_2/\tau_4^K] T_e A_x, x\colon \forall\alpha_4,\gamma_4.\tau_4 \vdash e\colon \tau_e \bmod V \qquad \tau_e \overset{U_e}{\sim} \tau_2 \bmod V \cup \{\alpha_3\} \\ \alpha_3 \notin TV(U_e T_e A) \qquad [\alpha_2/\tau_4^K] T_e \alpha_2 \overset{U_{x'}}{\sim} \tau_3 \qquad \sigma_{x'} = Gen([\alpha_2/\tau_4] T_e A, U_{x'} \tau_3^K) \\ (\mathbb{D})\ T_f U_{x'} [\alpha_2/\tau_4] T_e A_{unK,x}, unK\colon \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1), x'\colon \sigma \vdash [x/unK\ x']f\colon \tau_f \bmod V\end{array}}{\begin{array}{l}(\mathbb{B})\ T' U T A_{unK}, unK\colon \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1) \\ \qquad \vdash \texttt{letrec}\ x' = K\ ([x/unK\ x']e)\ \texttt{in}\ [x/unK\ x']f\colon \tau_i \bmod V\end{array}}$$

$[\alpha_2/\tau_4]T_e\alpha_2$ and τ_3 differ only in the names of their free variables, so we can disregard this unification and continue using either (lets choose τ_3^K). $\sigma_{x'}$ must then be $\forall\gamma_3.\tau_3$.

$$\text{(LetRec)}\ \frac{\begin{array}{c}\sigma_K = \forall\gamma_2.(\forall\alpha_3.\tau_3) \to \tau_3^K \qquad [\alpha_2/\tau_4^K]T_eA_x, x\colon \forall\alpha_4,\gamma_4.\tau_4 \vdash e\colon \tau_e \bmod V \\ \tau_e \overset{U_e}{\sim} \tau_2 \bmod V \cup \{\alpha_3\} \qquad \alpha_3 \notin TV(U_eT_eA) \\ (\mathbb{D})\ T_f[\alpha_2/\tau_4^K]T_eA_{unK,x'}, unK\colon \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1), x'\colon \forall\gamma_1.\tau_3^K \vdash [x/unK\ x']f\colon \tau_f \bmod V\end{array}}{(\mathbb{B})\ \begin{array}{l}T'UTA_{unK}, unK\colon \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1) \\ \qquad \vdash \texttt{letrec}\ x' = K\ ([x/unK\ x']e)\ \texttt{in}\ [x/unK\ x']f\colon \tau_i \bmod V\end{array}}$$

We apply the Spec rule to resolve the expression-level substitution in $(\mathbb{D})$ and calculate T_f and τ_f.

$$\text{(Spec)}\ \frac{T_f[\alpha_2/\tau_4^K]T_eA_{unK,x}, unK\colon \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1), x\colon \forall\alpha_5,\gamma_5.\tau_5 \vdash f\colon \tau_f \bmod V}{(\mathbb{D})\ T_f[\alpha_2/\tau_4^K]T_eA_{unK,x'}, unK\colon \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1), x'\colon \forall\gamma_1.\tau_3^K \vdash [x/unK\ x']f\colon \tau_f \bmod V}$$

We can now calculate $T'UT$ and τ_i from the LetRec rule

$$\text{(LetRec)}\ \frac{\begin{array}{c}\sigma_K = \forall\gamma_2.(\forall\alpha_3.\tau_3) \to \tau_3^K \qquad [\alpha_2/\tau_4]T_eA_x, x\colon \forall\alpha_4,\gamma_4.\tau_4 \vdash e\colon \tau_e \bmod V \\ \tau_e \overset{U_e}{\sim} \tau_2 \bmod V \cup \{\alpha_3\} \qquad \alpha_3 \notin TV(U_eT_eA) \\ T_f[\alpha_2/\tau_4^K]T_eA_{unK,x}, unK\colon \forall\gamma_1.(\forall\alpha_1.\tau_1) \to \tau_1^K, x\colon \forall\alpha_5,\gamma_5.\tau_5 \vdash f\colon \tau_f \bmod V\end{array}}{(\mathbb{B})\ \begin{array}{l}T_f[\alpha_2/\tau_4]T_eA_{unK}, unK\colon \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1) \\ \qquad \vdash \texttt{letrec}\ x' = K\ ([x/unK\ x']e)\ \texttt{in}\ [x/unK\ x']f\colon \tau_f \bmod V\end{array}}$$

And finally we can return to the Let rule to calculate T_i and τ_i. They are $T_f[\alpha_2/\tau_4^K]T_e$ and τ_f respectively. Because we have removed K and unK from all expressions in the premises, we can also remove any K and unK bindings from the environments in those premises. Taking all remaining assumptions and non-trivial side conditions we get our final rule for the polymorphic recursive emulating FCP term with witnessing constructors (6.2).

(∼PolyRec)

$$\frac{\begin{array}{c}\sigma_K = \forall\gamma_1.(\forall\alpha_1.\tau_1) \to \tau_1^K \\ T_eA_x, x\colon \forall\alpha_4,\gamma_4.\tau_4 \vdash e\colon \tau_e \bmod V \qquad \tau_e \overset{U_e}{\sim} \tau_1 \bmod V \cup \{\alpha_3\} \qquad \alpha_3 \notin TV(U_eT_eA) \\ \sigma_K = \forall\gamma_2.(\forall\alpha_2.\tau_2) \to \tau_2^K \qquad T_f[\alpha_2/\tau_2^K]T_eA_x, x\colon \forall\alpha_5,\gamma_5.\tau_1 \vdash f\colon \tau_f \bmod V \qquad \beta = \emptyset\end{array}}{\begin{array}{c}T_f[\alpha_2/\tau_4]T_eA_{unK}, unK\colon \forall\gamma_1.\tau_1^K \to (\forall\alpha_1.\tau_1) \\ \vdash \texttt{letrec}\ x' = K\ ([x/unK\ x']e)\ \texttt{in}\ [x/unK\ x']f\colon \tau_f \bmod V\end{array}}$$

Figure 6.4: Type inference for polymorphic recursion by type annotations

$$\frac{\begin{array}{c} T_e A_x, x\colon \forall\alpha.\tau \vdash e\colon \tau_e \text{ mod } V \qquad \tau_e \overset{U_e}{\sim} \tau_1 \text{ mod } V \cup \{\alpha_3\} \\ \alpha_3 \notin TV(U_e T_e A) \qquad T_f[\alpha_2/\tau_2]T_e A_x, x\colon \forall\alpha\tau \vdash f\colon \tau_f \text{ mod } V \qquad \beta = \emptyset \end{array}}{T_f[\alpha_2/\tau_4]T_e A \vdash \texttt{letrec}\ x\colon \forall\alpha.\exists\beta.\tau = e\ \texttt{in}\ f\colon \tau_f \text{ mod } V}\ (\textsc{PolyRec})$$

Since the type annotation expression (6.1) and the constructor witnessing expression (6.2) are equivalent, this rule can be translated into a type inference rule for polymorphic recursion via type annotations by swapping (6.1) for (6.2) in the conclusion (shown in Figure 6.4).

6.6 Definitions of `kar`, `kdr` and `ispair`

As we have seen, the matching of application patterns compiles to expressions which involve the primitives `kar`, `kdr` and `ispair`. However, this this point we've said little about how these constructs are to be evaluated at run-time. So in this section we will consider how their behaviour is implemented in the DGEN runtime. Later, in Chapter 9, we will complete this story by giving a more formal presentation of their behaviour as a component of the overall formal operational semantics of our language. Recall the informal definitions we gave earlier:

`kar` *d* which retrieves the left part of the datum `d` when the datum is viewed as a tuple, and

`kdr` *d* which retrieves the right part of the datum `d` when the datum is viewed as a tuple, and

`ispair` *d t e* which evaluates to the expression `t` if the datum `d` is a tuple and evaluates to the expression `e` if it is not.

6.6.1 The Epic Runtime

Recall that we use Epic [8] as a super-combinator compiler and a runtime. To understand our implementation of `kar/kdr/ispair` we need to know a little bit more about how Epic works.

1. All runtime values are designated as `VAL`s and constructed data is a special kind of `VAL`.

2. Constructed data is stored as a structure of

    - a tag for the constructor (extracted with the `TAG` macro),
    - a type string used by `typeOf` (extracted with the `TYCON` macro),
    - the arity of this constructor (extracted with the `ARITY` macro), and
    - an array of values for each argument to this constructor (extracted with the `ARGS` macro).

3. We construct data with the `CONSTRUCTOR` macro, which takes all the required information for a constructed value and builds a `VAL`.

Support for `kar`, `kdr` and `ispair` requires no change at all to the way data is *stored* in the run-time. These primitive operations act as an abstraction over the actual run-time data representation to make it look like the spine view when we require it.

Listing 6.2: `kar` in the Epic/DGEN runtime

```
1 VAL kar(VAL one){
2     return (CONSTRUCTOR(TAG(one), TYCON(one), ARITY(one)-1, ARGS(one)));
3 }
```

Listing 6.2 shows `kar` in the DGEN runtime. The `CONSTRUCTOR` macro will only take as many args from `ARGS(one)` as it needs to, based on the arity we passed in, which is one less that the original. Thus this code is creating a new constructed datum with all the original arguments except the last.

Listing 6.3: `kdr` in the Epic/DGEN runtime

```
1 VAL kdr(VAL one){
2   con cc = (con)one->info;
3   return (cc->args[ARITY(one) - 1]);
4 }
```

Listing 6.3 shows `kdr` in the DGEN runtime. This function returns the last argument in the argument array. It is already a `VAL` so there is nothing more to do.

Listing 6.4 shows `ispair` in the DGEN runtime. The `ispair` function is capable of working on any data whatsoever, so it first looks up the `VAL`-type of its input (which is an efficient `switch` statement). If it has a constructed datum, it returns true for arity > 0 and false otherwise. It will return false for any input that is not a constructed datum (such as integers and other primitive values).

Listing 6.4: `ispair` in the Epic/DGEN runtime

```
1  bool isPair(VAL one){
2    switch(GETTY(one)){
3          case CON:
4              return (ARITY(one) > 0)
5              break;
6          default:
7              return false;
8    }
9  }
```

6.7 Application of Partially Applied Constructors to Values

We now need to define what happens when a partially applied constructor is applied to something, i.e. we need to define $make_\tau$ from page 100. This application can occur because the `x(y)` pattern gives the partially applied constructor `x` an identity (it is bound to a variable which we can then use in the body of the `case` alternative). As in `apply_to_all` (page 117), we most often want to re-attach it to its missing argument (suitably modified). This is not difficult change in Epic, but you may find things work differently in your runtime. The code for applying one thing to another already has a `switch` in it, so we simply add a new branch that tells us what to do if the left argument to the application is actually constructed data. Listing 6.5 shows the code that runs for that `switch` case.

Listing 6.5: Applying a datum to an argument in Epic/DGEN

```
1  int arityxin = ARITY(xin);
2  int arityout = arityxin + 1;
3  void** argspace = EMALLOC(arityout*sizeof(VAL));
4  memcpy((void*)(argspace), (void*)(ARGS(xin)), arityxin*sizeof(VAL));
5  memcpy((void*)(argspace + arityxin), (void*)block, sizeof(VAL));
6  VAL x = CONSTRUCTORn( TAG(xin)
7                      , arityout
8                      , argspace
9                      );
```

This code will create a new constructor with all the arguments of the first argument, `xin`, and with the second argument (which must be a single `VAL`) added to the end. The arity is updated appropriately.

The mechanisms described in this chapter do cause generic code to use more memory than the equivalent polymorphic code, we measure this and discuss this in more detail in Chapter 8.

Figure 6.5: Type Inference for Lonely Constructor Primitives

(ConstrEq)
$A \vdash e \texttt{===} f : \alpha \to \beta \to Bool$

(ConstrShow)
$A \vdash \texttt{constr_show}\, e : \alpha \to String$

6.8 A Prescription for Lonely Constructors in the Runtime

Regardless of the value of `x`, `kar(x)` is not a data value in the traditional functional setting. By giving it the type $\rho \to \tau$ where ρ is existentially quantified, our type system ensures we only use it in ways that are safe. We are able to define most of our generic functions without giving partially applied constructors first-class status. However, as yet, we can't write generic show or generic equality (page 24). To complete the capabilities we require, we need to lift lonely constructors closer to "first-class" status. To be clear, we will only reify *lonely* constructors in this section, *partially applied* constructors (with at least one attached argument) will not be given any extra affordances. It is a simple matter to differentiate lonely constructors from partially applied constructors since lonely constructors are atoms. In effect, we will show how to treat lonely constructors like any other built-in value (integers, characters, etc).

In the same way we have built-in operations for built-in types (like `ord` to get the value of a character), we should have built-in functions on lonely constructors. So, we want to add two new built-in operations on lonely constructors: === which is constructor equality, and `show_constr` which converts a constructor to a string. It is a simple matter to define new literal operations in our internal languages, they pass through the compiler without effecting any of the compilation algorithms except for type inference.

We must define type inference rules for these new operations, but it is a straightforward affair. Firstly we note that lonely constructors can have many types, for example `Cons :: (a, list(a))-> list(a)` while `Pair :: (a, b)-> pair(a,b)`. Thus these new primitive operations need to have polymorphic input types. The result type for each is self-evident. This we want to infer the following types for these two new primitive operations

$$\texttt{===} : \alpha \to \beta \to Bool$$
$$\texttt{constr_show} : \alpha \to String$$

We don't require any new techniques, we only need to formulate type inference rules which reflect the desired types. Figure 6.5 shows these type inference rules.

It is only left to define these operations in Epic (DGEN's runtime system). Both these functions can take in *any* input, thus we need to build some run-time checks into the definitions to cope with input of undesirable values. It is up to the compiler writer to define the operation of these functions on input that is not lonely constructors, we have chosen to return false and the empty string for equality and show respectively. Listings 6.6 and 6.7 show our definitions of constructor show and equality.

Listing 6.6: Constructor Show in the DGEN Runtime

```
1  char* constrToStr(Closure* cl){
2    if(ISCON(cl)){
3      con* c = (con*)(cl->info);
4      char* str = c->tycon;
5      char* buf = EMALLOC((4+strlen(str))*sizeof(char));
6      sprintf(buf,"[%d,%s]", c->tag & 65535, c->tycon);
7      return buf;
8    }
9    else
10     char* str = "";
11     char* buf = EMALLOC((4+strlen(str))*sizeof(char));
12     sprintf(buf,"%s", str);
13     return buf;
14 }
```

Listing 6.7: Constructor Equality in the DGEN Runtime

```
1  int constreq(Closure* one, Closure* two){
2      switch(GETTY(one)) {
3      case CON:
4          if(  (GETTY(two) == CON)
5                   && (TAG(one) == TAG(two))
6                   && (strcmp(TYCON(one),TYCON(two)) == 0)
7            ){
8                return MKINT(1);
9          }
10         else{
11               return MKINT(0);
12         }
13         break;
14     }
15     return MKINT(0);
16 }
```

Arbitrary values in Epic are `Closures` which we can inspect to find out what form of closure (either with `GETTY` or with a specific `IS*` macro). Boolean values are stored as integers in the runtime, hence string equality returns an integer. Constructor show builds a string representation for the constructor from its type and its tag. This is not the "pretty printing" that Haskell's `show` function performs, it is a simpler method but it

still gives a unique identifying string for each constructor. Constructor equality needs to check both the equality of the constructor's tags (its identity *within* its type) and that the two values come from same type. We could test only the tag if constructor equality had type $\alpha \to \alpha \to Bool$ rather than the type it has, $\alpha \to \beta \to Bool$. We address this matter in more detail in Section 7.3.1.

The downside of this approach is that the built-in functions on atoms must be provided by the compiler. However, we consider this a small burden since compiler writers would already do this for other atoms (integers, characters, etc). The only extra work is to include constructors without arguments as one of these atoms, a relatively small addition to creating all the required built-in functions. It is easy to determine what built-in operations should be provided for lonely constructors. Any built-in that is defined for all other primitive types should be provided for lonely constructors as well.

This raises the status of lonely constructors to values of the language rather than just tags for other values. Note that we have found a very simple way to give them this status. As we have seen, we have not had to make wholesale changes to the run-time to achieve this.

In summary, our prescription for lonely constructors is to include polymorphic operations which work on lonely constructors, one for each of the operations that are overloaded for all the other built-in types. Performing this relatively simple task, combined with function extension and application pattern matching, allows one to expand a function with definitions for all the built-in types into one that works for *all types in the language*, as we have done with generic equality and generic show.

6.9 Summary

We have shown that structure agnosticism can be achieved with modest changes to a baseline functional compiler via application pattern matches. Application pattern matches can be removed with a small extension to the pattern compilation algorithm. The resulting primitives can be typed using FCP plus a small extension to the type environment. Finally, the new primitives can be realised in the run-time without any changes to how values are *stored* at run-time. We also described how to supply built-in primitive operations on lonely constructors to expand the set of generic functions to include operations like equality and show.

We have now described everything required to compile generic functions. These techniques have been implemented in DGEN and we now use that implementation to evaluate them.

Chapter 7

Evaluating Expressiveness

In this chapter we examine the extent to which our language, DGEN, provides enough of a foundation upon which to build generic functional compilers. Specifically we do this by demonstrating that it is expressive enough to support a very wide repertoire of admissible generic program types. We look well beyond our initial group of eight example snippets to see the full capabilities (and limitations) of using the explicit spine view and a single function extension operator. We also show that these two mechanisms are powerful enough to emulate the slightly different spine view used in SYB and the computation style of strategic rewriting [85, 87]. The latter of these topics encompasses a data transformation style which is often (although not always [80]) supported by specifically designed domain specific languages. In our case, however, we demonstrate that DGEN is expressive enough to support full strategic rewriting in a library.

To show the full capabilities of the mechanisms we have developed, and to demonstrate that our type system is accepting and rejecting the right programs we:

- revisit the five kinds of generic functions we introduced in Chapter 2,
- describe variants of the generic functions which illuminate the extent of the capabilities we have introduced,
- encode strategic rewriting in DGEN, and
- explain the limitations of these techniques.

We have placed the full source of the compiler online and have exposed a working instance of the compiler at the URL `dgen.science.mq.edu.au`. Interested readers can compile and run any tests at that URL.

7.1 Five Kinds of Generic Programs

Out first task is to demonstrate that we can compile and run all the generic functions outlined in Chapter 2. When we introduced these functions we, in effect, *assumed* the presence of appropriate function extension and application pattern match capabilities. In the chapters since, we have shown how these capabilities can be added to a baseline functional compiler. In this section we show that the mechanisms we built in Chapters 5 and 6 are capable of performing the roles we intended for them. We will do this by highlighting the relevant part of each snippet from Chapter 2 and giving a brief commentary explaining how our function extension and application pattern matches fulfil the roles required of them.

7.1.1 Generic Update

Listing 7.1: Part of the salary update snippet

```
1 def incS(amt, s) = case [s] of
2                         { [S(s)]    -> S(s + amt)
3                         } otherwise -> error "partial definition error in incS"
4 def id(x) = x
5 def increment(amt) = incS(amt) ▷ id
6 def generic_update(func, dat) :: (∀ a . (a) -> a, b) -> b = apply_to_all(func,dat)
```

Listing 7.1 shows the relevant part of the salary update snippet from Listing 2.3. The `incS` function is defined by `case` analysis on values of type `salary`. The type inference mechanism can infer this because the `case` pattern is a constructor of the `salary` type. Thus the inferred type for `incS` is `(int, salary())-> salary()`. We want this function to be passed into `generic_update`, which will apply its first argument throughout the structure of its second argument. Thus we need to construct a function with type ∀ `a . (a)-> a`, but which has the *behaviour* of `incS`.

Function extension allows us to specialise a generic function with `incS` in exactly the way we require. The synthesised generic function `incS(amt)▷id` will have type ∀ `a . (a )-> a` and will behave exactly as `incS` does when applied to values of type `salary`.

The type system will accept this extension because it allows the type of `id` (i.e. `(a) -> a`) in the right hand side of the function extension operation and it will successfully unify that type against the type of `incS(amt)` (i.e. `(salary())-> salary()`). Furthermore, `salary()` is a ground, arrow-less type so the side conditions in the type inference rule for function extension (see page 82) are not triggered. At runtime, function extension has

been converted into a `typeOf` operation (see page 90) in such a way that any value with type `salary()` (which is exactly those values constructed with the `S` constructor) will be applied to `incS` as desired and all other values will be applied to `id` (see page 91).

7.1.2 Generic Query

Listing 7.2: Part of the name analysis snippet

```
1  def check_it(strbool)  :: (pair(list(string),bool), a) -> pair(list(string),bool)
2       = check_comm(strbool) ▷ check_intexp(strbool) ▷ fun(a) = strbool
3
4  def decl_before_use(comm) = snd(generic_query(check_it,Pair([],true),comm))
```

Listing 7.2 gives the relevant part of the name analysis snippet from Listing 2.4. The `check_it` function requires the ability to string together specific functions with the function extension operator and have whichever is appropriate apply when the synthesised generic function is applied to a value. Furthermore, it requires that these functions can have a known return type (in this case `pair(list(string),bool)`), values of which will be threaded through the `generic_query` function.

Function extension is able to support both these requirements. Firstly, function extension is right associative (see page 78), for example `f ▷ g ▷ h`, is more explicitly stated as `f ▷ (g ▷ h)`. Thus we can string together as many extension operations as we require.

Function extension supports known types as the result type of synthesised generic functions because it only checks the input type for polymorphism (see page 82). Furthermore, the unification of the types of the specific and generic functions ensures that the known return type is the same for all functions in the chain.

7.1.3 Generic Traversal

Listing 7.3: A bottom-up generic traversal

```
1  def apply_to_all(f,g) :: (∀ a . (a) -> a, b) -> b =
2      case [g] of
3           { [c(a)]    -> f(@apply_to_all(f,c)(apply_to_all(f,a)))
4           ; [o]       -> f(o)
5           } otherwise -> error "partial definition error in apply_to_all"
```

We will use the bottom-up generic update, repeated for convenience in Listing 7.3, to represent all the generic traversal snippets since it includes all the relevant requirements.

The `apply_to_all` function assumes the presence of:

1. Polymorphic recursion to allow two calls to `apply_to_all` at two different types in its own body.

2. Rank-2 types to allow two different calls to `f`, with arguments of different types, in the body of `apply_to_all`.

3. The ability to pull anonymous (i.e. we can't identify how it was constructed) values into two pieces.

4. The ability to transform these extracted pieces using a type preserving function.

5. The ability to "re-assemble" the transformed pieces.

We have extended our type system to support polymorphic recursion (see page 109) and rank-2 types (see page 88) so the first two requirements are satisfied. Application pattern matches allow us to disassemble any value into two pieces, so the third requirement is satisfied. The values that come from this extraction are given the types $\rho \to \tau$ and ρ where τ is the type of the original value and ρ is a unique, existentially quantified type variable. The only restriction on these types is that ρ can only unify with itself. Specifically, we can pass values of type ρ to functions that expect type ρ (of which there are none since ρ does not exist outside of the existential scope) or which are polymorphic. The type preserving function in this example is polymorphic in its input type, so we can pass both parts of the original value to it for transformation, satisfying our fourth requirement. Furthermore, since the transformation is type preserving, the resulting values have the same types as they did when they were originally split. Since ρ *can* be unified with itself, we can re-assemble the transformed parts, satisfying the final requirement.

7.1.4 Generic Equality and Generic Show

These two snippets required only one extra capability, polymorphic built-in functions on lonely constructors. In Section 7.2.3 we will expand upon DGEN's support for the nested function extension used in generic equality so we will defer discussion of that feature until then. On page 112 we gave a prescription for adding primitive functions on lonely constructors to a run-time and giving the required polymorphic types to them. Because these primitives have polymorphic types, they can be the generic function in a function extension. In other words, we can safely use them as the final function in a chain of function extension operations, as shown in Listings 7.4 and 7.5.

Listing 7.4: Part of Generic Equality

```
def g_str_eq() :: (a,b) -> Bool =
       (fun(a) = (fun(b) = a s== b)
                  ▷
                  (fun(b) = false)
       )
       ▷ (fun(a) = (fun(b) = a === b))

def g_int_eq() :: (a,b) -> Bool =
       (fun(a) = (fun(b) = a i== b)
                  ▷
                  (fun(b) = false)
       )
       ▷ g_str_eq()

def g_char_eq() :: (a,b) -> Bool =
       (fun(a) = (fun(b) = a c== b)
                  ▷
                  (fun(b) = false)
       )
       ▷ g_int_eq()

def g_bool_eq() :: (a,b) -> Bool =
       (fun(a) = (fun(b) = a b== b)
                  ▷
                  (fun(b) = false)
       )
       ▷ g_char_eq()

def bi_eq(x,y) :: (a,a) -> Bool
               = g_bool_eq(x,y)
```

Listing 7.5: Part of Generic Show

```
def bishow() :: (a) -> String
             = let si(x) = show_int(x)
               and sc(x) = show_char(x)
               and sb(x) = show_bool(x)
               and ss(x) = if (x s== "") then x else x
               and ds(x) = show_constr(x)
               in  si ▷ sc ▷ sb ▷ ss ▷ ds
```

7.2 Variants

We now turn our attention to exploring the extent of what we have achieved with the explicit spine view and function extension.

7.2.1 Structure Modification (with Type Preservation)

As long as we maintain the type of each node, we can do *anything* with generic updates. Listing 7.6 shows an alternative to updating salary in the same datatype as the salary update snippet (and Listing 7.7 shows the output of this program). In this example we

wish to “flatten” a sub-unit, identified by its name. It is a port to DGEN of an example in [60].

Listing 7.6: Program for flattening sub-units

```
def flatten(n,c) = let g_flatD(n) = flatD(n) ▷ id
                   in  everywhere(g_flatD(n),c)

def flatD(n,c) = case [c] of
                      { [D(n',m,us)] -> D(n',m,concatMap(unwrap(n),us))
                      } otherwise   -> error "partial definition error in flatD"

def unwrap(q,su) :: (string, sub_unit()) -> list(sub_unit())
                 = case [su] of
                        { [DU(D(d',M(m),us))] -> if (q s== d')
                                                    then Cons(PU(m),us)
                                                    else [su]
                        ; [u]                 -> [u]
                        } otherwise           -> error "partial definition error in unwrap"

// setup
def gen_com() = let ralf()     = E(P("Ralf", "Amsterdam"), S(8000))
                and joost()    = E(P("Joost", "Amsterdam"), S(1000))
                and marlow()   = E(P("Marlow", "Cambridge"), S(2000))
                and blair()    = E(P("Blair", "London"), S(100000))
                and terrence() = E(P("Terrence", "Ottowa"), S(3000))
                and phillip()  = E(P("Phillip", "Montreal"), S(3000))
                and dion()     = E(P("Dion","Quebec"), S(10000))
                in  C([ D("Research", M(ralf), [ PU(joost)
                                               , PU(marlow)
                                               , DU(D("Hijinx", M(dion), [ PU(terrence)
                                                                         , PU(phillip)]
                                                     ))
                                               ])
                      , D("Strategy", M(blair), [])
                      ]
                     )

main = do_all([ put_string("--- Before ---\n")
              , put_string(show_company(gen_com))
              , put_string("\n--- Can't flatten top level department ---\n")
              , put_string(show_company(flatten("Research", gen_com)))
              , put_string("\n")
              , put_string("--- Can flatten sub units ---\n")
              , put_string(show_company(flatten("Hijinx", gen_com)))
              , put_string("\n")
              ])
```

We are able to use the same `generic_update` function to traverse the datatype but instead of looking for salaries to update, we look for departments to flatten. When we find a department (any department), we check its sub units to see if the department we want to flatten is in there (`unwrap`). If it is, we demote its manager to a normal employee and move all its normal employees (and sub-units) to the surrounding department. We are changing the structure of the data, but as long as we are maintaining the type, it is

safe (and passes DGEN's type inference).

Listing 7.7: Output of flattening sub-units

```
--- Before ---
Company[ Department: Research, Ralf<Amsterdam, 8000>,
          [ Joost<Amsterdam, 1000>
          , Marlow<Cambridge, 2000>
          , Department: Hijinx, Dion<Quebec, 10000>, [ Terrence<Ottowa, 3000>
                                                     , Phillip<Montreal, 3000>
                                                     ]
          ]
    and  Department: Strategy, Blair<London, 100000>, []
]
--- Can't flatten top level department ---
Company[ Department: Research, Ralf<Amsterdam, 8000>,
          [ Joost<Amsterdam, 1000>
          , Marlow<Cambridge, 2000>
          , Department: Hijinx, Dion<Quebec, 10000>, [ Terrence<Ottowa, 3000>
                                                     , Phillip<Montreal, 3000>
                                                     ]
          ]
    and  Department: Strategy, Blair<London, 100000>, []
]
--- Can flatten sub units ---
Company[ Department: Research, Ralf<Amsterdam, 8000>,
          [ Joost<Amsterdam, 1000>
          , Marlow<Cambridge, 2000>
          , Dion<Quebec, 10000>
          , Terrence<Ottowa, 3000>
          , Phillip<Montreal, 3000>
          ]
    and  Department: Strategy, Blair<London, 100000>, []
]
```

However, the traversal order can affect the correctness of an operation like this one. Once we start changing the structure of the data we are working over, we need to ensure that the traversal we are employing to find elements is working in the order we expect. Otherwise we might find that a structure we are looking for has been removed further up (or down) the tree. In this case, the top-down traversal of `generic_query` is perfect.

7.2.2 Datatype Aware Traversals with no "Boilerplate"

The name analysis snippet is only possible because `generic_query` does traversal in the way that the problem requires. The ability to write our own generic traversals means that if the right traversal is not available in the libraries, we can always write our own. However, splitting the job of *traversal* from the *computation* required can be very awkward. Sometimes it is just simpler to do the traversal with monomorphic functions on the types in question. If you do this though, you are normally committed to update these functions

every time you extend the datatypes. In this section we will see how application pattern matches free you from this burden and as such makes another small dent in the *expression problem* [90].

Listing 7.8: An alternate algorithm for name analysis

```
1  def check_comm(lst, expr) :: (list(string),comm()) -> bool =
2      case [expr] of
3           { [CDecl(v,z,e)] -> decl_before_use(Cons(v,lst),e)
4           ; [CAssign(v,z)] -> elem(fun(p,q) = p s== q,v,lst)
5           ; [c(a)]         -> decl_before_use(lst,c) & decl_before_use(lst,a)
6           ; [z]            -> true
7           } otherwise      -> error "partial function error in check_comm"
8
9  def check_intexp(lst,expr) :: (list(string), int_exp()) -> bool =
10     case [expr] of
11          { [IUse(v)]      -> elem(fun(p,q) = p s==q,v,lst)
12          ; [c(a)]         -> decl_before_use(lst,c) & decl_before_use(lst,a)
13          ; [z]            -> true
14          } otherwise      -> error "partial function error in check_intexp"
15
16
17 def check_other(lst, a) :: (list(string), a) -> bool =
18     case [a] of
19          { [c(a)]         -> decl_before_use(lst,c) & decl_before_use(lst,a)
20          ; [z]            -> true
21          } otherwise      -> error "partial function error in check_other"
22
23 def decl_before_use(lst) :: (list(string), a) -> bool = check_comm(lst) ▷ check_intexp(lst)
       ▷ check_other(lst)
```

Listing 7.8 shows a version of the name analysis code which is less generic than Listing 2.4 (on page 21) because it is "datatype aware". We have written one method for each interesting type (`comm()` and `int_exp()` as before) but this time we traverse into the children explicitly instead of leaving that job to `generic_query`. This means that each function needs to have `case` alternatives for all possible constructors. Without application pattern matches, this would mean enumerating all constructors (since we need to recurse into constructor arguments). This would force an update to the function every time the associated datatype changed. With application pattern matches we can cover all "uninteresting" constructors with one application pattern, and all atoms with one variable pattern. This not only makes the initial definition more concise, it means extensions to `comm()` or `int_exp()` only force changes to `check_comm` and `check_intexp` if the new constructor has some interesting behaviour in that function.

For example, Listing 7.9 shows expanded algebraic datatype definitions that were created *with no corresponding changes to the code for name analysis*. We were able to add new

Listing 7.9: An expanded `comm()` algebraic datatype

```
1  adt comm() = CAssign(string, int_exp())
2             | CDecl(string, int_exp(), comm())
3             | CSkip()
4             | CSeq(comm(), comm())
5             | CWhile(bool_exp(), comm())
6             | CPut(int_exp())
7             | CUntil(bool_exp(), comm())
8             | CFor(int_exp(), int_exp(), int_exp(), comm())
9
10 adt int_exp() = IUse(string)
11               | ILit(int)
12               | IPlus(int_exp(), int_exp())
13               | IMinus(int_exp(), int_exp())
14               | IMultiply(int_exp(), int_exp())
15               | IDiv(int_exp(), int_exp())
16               | IMod(int_exp(), int_exp())
17
18 adt bool_exp() = BTrue()
19                | BFalse()
20                | BEq(int_exp(), int_exp())
21                | BNEq(int_exp(), int_exp())
22                | BAnd(bool_exp(), bool_exp())
23                | BOr(bool_exp(), bool_exp())
```

integer primitive operations (`IMinus`, `IMultiply`, `IDiv` and `IMod`) and new control flow structures (`CUntil` and `CFor`) without changing any of the code to perform the name analysis check.

The style of traversal we have described in this section is similar to the type safe traversal functions of Asf+Sdf [85]. Asf+Sdf is a term-rewriting system that supports traversal functions with which one can define the traversal scheme for a particular rewrite rule. Normal rewrite rules are used to define the interesting behaviour of the traversal while a more generic traversal function defines how the traversal should visit the tree. Thus Asf+Sdf supports a split of duties similar to that in Listing 7.8. Furthermore, Asf+Sdf and dgen support a similar level of type safety for the defined traversals. One important difference between the two systems is that Asf+Sdf uses traversal parameters, which are passed to traversal functions, to customise the traversal behaviour whereas in dgen the developer does the traversal by hand using the explicit spine view of data. The Asf+Sdf approach is simpler while the dgen approach is more flexible.

7.2.3 Function Extension with Multiple Arguments

Although each instance of the function extension operator ($\triangleright$) chooses which function to run based on only a single argument, we can nest them to discriminate on multiple arguments. Listing 7.10 shows a polymorphic function that returns `None()` for input that

is not two integers, and Some(a+b) for two integer inputs a and b.

Listing 7.10: Choosing functions based on two arguments

```
1  def plus(x,y) = Some(x+y)
2  def noneInt(b) = if (true) then None() else Some(5)
3  def noneIntInt(a,b) = if (true) then None() else Some(5)
4
5  def plus_or_none() :: (a,b) -> some(int) =
6         (fun(a) = (fun(b) = plus(a,b))
7                   ▷
8                   (noneInt)
9         )
10        ▷ (noneIntInt)
```

The plus_or_none function defines an inner, single-argument function, which discriminates on the second argument, adding it to the first if it is an integer. This is nested inside a similar function which runs the inner one if the first argument is an integer and defers to the default (None()) if not. The definitions of these defaults are complicated by the need to get the types right without the benefit of general type annotations. However, it is a simple task to extend DGEN with such annotations.

7.2.4 Generic Zip-With

Implicitly using the spine view (as is done in SYB) can make traversals over two or more structures quite awkward. The discussion of the encoding of a generic zip-with function in [59] is indicative of this. Our approach of exposing the spine view via pattern matching significantly ameliorates this problem. Listing 7.11 encodes a generic zip-with function whose sheer simplicity clearly demonstrates the utility of the explicit spine view in this context. A generic zip-with function takes in two structures and traverses each in parallel. For each pair of atoms it finds, it combines them using the provided zip function (fn). Its result is a list of all the zipped values that it discovered along the way.

Listing 7.11: A generic zip-with function

```
1  def gzipwithq(fn, a, b) :: (∀ x, y. (x, y) -> r,a,b) -> list(r)  =
2       case [a,b] of
3            { [c1(z1), c2(z2)] -> append(gzipwithq(fn,c1,c2),gzipwithq(fn,z1,z2))
4            ; [p,q]            -> [fn(p,q)]
5            } otherwise        -> error "partial definition error in gzipwithq"
```

One difference between this encoding and the one in [59] is that constructors are included as atoms in the DGEN version. This means that the resulting list will include zip-

values for each constructor discovered along the way. This behaviour is both anticipated and to be desired since partially applied constructors are valid values in DGEN. SYB works in Haskell, where partially applied constructors are functions. Listing 7.12 shows an example of using `gzipwithq` and the result of that example.

Listing 7.12: Using `gzipwithq` (some code elided for space)

```
main = ... put ... gzipwithq(plus_or_none, [1],[2])))
       ... put ... gzipwithq(plus_or_none, [10,20,30],[1,2,3])))
...
// output
[None(),Some(3),None()]
[None(),Some(11),None(),Some(22),None(),Some(33),None()]
```

In this example we have used a polymorphic function with specific behaviour that works on two arguments (`plus_or_none` from Section 7.2.4) as the polymorphic input and each constructor in the input structures is represented by `None()` in the output list. This list could then be filtered to have just the valid integers if desired.

7.2.5 SYB Primitive Operations

As shown with generic zip-with, DGEN's spine view does not necessarily coincide with that in SYB. In this section we show that we can recover the SYB behaviour with encodings of the `gmapT` and `gmapQ` functions from [60]. In SYB, these functions (or the functions they rely on) are written either once for each datatype that we need to process generically, or are derived by the compiler. With DGEN they can be written once as library functions and will automatically work on all datatypes.

The function `gmapT` is a generic update function which only works on the immediate arguments of a constructed value (rather than recursively finding all children as generic update does). It also treats the constructor differently by *not* attempting to apply the polymorphic function with specific behaviour to it. Listing 7.13 shows the DGEN encoding of `gmapT`. It walks the spine of the value it has been given, applying the updating function to each argument as it encounters them and pasting the value back together once this is done.

The only atom encountered on the left of an application pattern match during a walk of the spine is the constructor. We use this fact to identify the constructor (in the second `case` alternative – line 4) and return it unchanged.

The function `gmapQ`, shown in Listing 7.14, is a generic query which also works only on

Listing 7.13: DGEN encoding of gmapT

```
1  def gmapT(fn,a) :: (∀ x . (x) -> x, a) -> a
2                 = case [a] of
3                       { [c(z)]    -> @gmapT(fn,c)(fn(z))
4                       ; [z]       -> z
5                       } otherwise -> error "partial definition error in gmap"
```

the immediate children of a value. Instead of updating values and pasting them back into the original structure (as gmapT does), gmapQ accumulates the transformed values in a list. This allows it to work with a type transforming function rather than a type-preserving one as gmapT and generic update require. Again we use the fact that a spine traversal can identify the constructor to avoid including its transformation in the list of returned values (line 6).

Listing 7.14: DGEN encoding of gmapQ

```
1  def gmapQ(fn,a) :: (∀ x . (x) -> r, a) -> list(r)
2                  = reverse(gmq(fn,a))
3  def gmq(fn,a) :: (∀ x. (x) -> r, a) -> list(r)
4                = case [a] of
5                      { [c(z)]    -> Cons(fn(z), gmq(fn,c))
6                      ; [z]       -> Nil()
7                      } otherwise -> error "partial definition error in gmq"
```

7.2.6 Expanding the Set of Built-In Operations

In Section 6.8 we saw how to create built-in equality and conversion to string (show) for all values by adding primitives for lonely constructors. We can repeat this process as many times as we want. For any operation we wish to do generically, we can define how to do it for each atomic value (including constructors with no arguments) and then build-up a function that can work on any value from these. A compiler writer just needs to take care that the type inference rule for built-in operations on lonely constructor are given polymorphic types since they will be used as the ultimate default function in (probably a string of) function extensions.

Generic Encode is a single function which can encode, as a bitstring, any value of any type (Listing 7.15).

Although it looks much more complicated than generic show, it works in the same way. A built-in operation on lonely constructors is used as the base to build up a bitstring function that can work for any atoms (bibitstring). Note that DGEN's parser requires

Listing 7.15: Generic Encode (bitstring function)

```
1  def bitstring_int(i) :: (int) -> list(bit())
2     = letrec bsi_help(bt, it) = case [bt] of
3                 { [0] -> error ”invalid input to bsi_help”
4                 ; [1] -> case [it] of
5                               { [0] -> [Zero()]
6                               ; [1] -> [One()]
7                               } otherwise -> error ”wrong remainder”
8                 } otherwise -> letrec g() = it / bt
9                                and    p() = it - g * bt
10                               in     case [g] of
11                                            { [0] -> Cons(Zero(), bsi_help(bt / 2, p))
12                                            ; [1] -> Cons(One(), bsi_help(bt / 2, p))
13                                            } otherwise -> error ”pfe in bsi_help”
14      in     bsi_help(1048576, i)
15
16 def bitstring_char(c) :: (char) -> list(bit())
17                      = bitstring_int(ord_char(c))
18
19 def bitstring_str(s) :: (string) -> list(bit())
20       = letrec bss_help(togo, ss) = if (togo i== 0)
21                                    then bitstring_char(char_at(ss,0))
22                                    else append(bss_help(togo-1,ss), bitstring_char(char_at(
                                           ss,togo)))
23         in     bss_help(str_len(s),s)
24
25 def bitstring_bool(b) = if (b) then [One()] else [Zero()]
26
27 def bitstring_constr(c) :: (a) -> list(bit())
28                         = bitstring_int(ord_constr(c))
29
30 def gbitstring(a) :: (a) -> list(bit())
31                  = case [a] of
32                         { [c(p)]    -> append(gbitstring(c), gbitstring(p))
33                         ; [z]       -> bibitstring(z)
34                         } otherwise -> error ”partial definition error in gbitstring”
35
36 def bibitstring() :: (a) -> list(bit())
37                  = let bsi(x) = bitstring_int(x)
38                    and bsc(x) = bitstring_char(x)
39                    and bsb(x) = bitstring_bool(x)
40                    and bss(x) = bitstring_str(x)
41                    and bsd(x) = bitstring_constr(x)
42                    in  bsi ▷ bsc ▷ bsb ▷ bss ▷ bsd
```

functions, not built-in operations, to be the arguments to ▷, so we need locally bound versions of each built-in operation. This would not be necessary with a more capable parser. A `case` expression is used to discriminate between compounds and atoms, applying the previously built-up operation for atoms and using recursion to dispatch one level of the structured data (`gbitstring`). The extra complexity comes from converting each type into a relatively efficient bitstring version. However, that algorithm is entirely standard.

7.2.7 Strategic Rewriting

As a final example of just how much we can achieve with these features, we give an encoding of strategic rewriting in DGEN. A strategic rewriting language (or library) typically provides a few primitive "strategy combinators" and allows the developer to build up more complex strategies from these. The canonical example is the Stratego programming language [87].

Strategic Rewriting Terminology

- A *rewrite rule* is a function which transforms a value into another form. It is defined as a left hand side which describes the term to rewrite and a right hand side which describes the transformed value. Function definition by pattern matching case expressions is sufficient to encode rewriting rules.
- A *primitive strategy* lifts one or more rewrite rules from working on one value to working on any value. Thus any encoding of primitive strategies must take account of the fact that a rewrite can fail (i.e. be applied to a value which can't match the right hand side of any underlying rewrite rule(s)).
- A *higher order strategy* is a function which takes a strategy as a parameter. Its result is a new strategy which uses the same lifted rewrite rule(s) but traverses values in a different way.

Our first task is to encode rewriting failure. We do this with an option/maybe type available in the DGEN libraries, `adt some(a)= Some(a)| None()`. A strategy is then a function from `a` to `some(a)`.

Listing 7.16 shows five primitive strategy combinators, which are taken from Stratego, encoded as functions in DGEN.

The `left_plus` function tries the first strategy and if that fails (returning `None()`) it applies the second strategy. The `left_star` function is similar but only runs the second strategy if the first succeeds. The `one` function will run the given strategy on the first appropriate child of the given value. A "child" in this context is an immediate argument of the constructed value (`one` fails trivially for atomic values like integers). The value is pulled apart with the first `case` expression and `one` is recursively applied. This ensures that we try from the first argument to the last, in a way that matches the semantics of the corresponding Stratego combinator. If this (`onesc`) fails, then we try the current argument

Listing 7.16: `<+`, `<*`, `one`, `id` and `fail` encoded in DGEN

```
1  //basic
2  def left_plus(s,t,d) = let sd() = s(d)
3                         in case [sd] of
4                                { [None()]  -> t(d)
5                                ; [Some(r)] -> Some(r)
6                                } otherwise -> error "partial defn error in left_plus"
7
8  def left_star(s,t,d) = let sd() = s(d)
9                         in case [sd] of
10                               { [None()]  -> None()
11                               ; [Some(r)] -> t(r)
12                               } otherwise -> error "partial defn error in left_star"
13
14 def one(s,d) :: (∀ a. (a) -> some(a), a) -> some(a)
15     = case [d] of
16           { [c(a)]  ->
17               let onesc() = one(s,c)
18               in case [onesc] of
19                       { [None()] -> let sa() = s(a)
20                                     in case [sa] of
21                                            { [None()] -> None()
22                                            ; [Some(r)] -> Some(@c()(r))
23                                            } otherwise -> error "partial defn error in one
                                                   [I]"
24                       ; [Some(r)] -> Some(@r()(a))
25                       } otherwise -> error "partial defn error in one [II]"
26           ; [o]     -> None()
27           } otherwise -> error "partial definition error in one"
28
29 def sid(d) = Some(d)  // needs s prefix to avoid name clash with normal id
30 def fail(d) = if (true) then None() else Some(d)
31   // the if is needed to force the type to be a -> some(a)
```

(a). In the event of success, either with `onesc` or `sa`, we reconstruct the original data but with the transformed argument. The `one` function requires that the strategy be a rank-2 argument since it is applied to all immediate children of the given value and we have no knowledge about what their types may be. We certainly have no guarantee that they are the same (under which circumstances we would not need the higher ranked argument). The `sid` and `fail` functions trivially succeed and fail respectively.

Listing 7.17: `attempt`, `oncetd`, `srepeat` and `outermost` encoded in DGEN

```
1  def attempt(s) = left_plus(s, sid)
2  def oncetd(s) :: (∀ a . (a) -> some(a), b) -> some(b)
3                 = left_plus(s, one(fun(g) = oncetd(s,g)))
4  def srepeat(s) :: (∀ a . (a) -> some(a), b) -> some(b)
5                 = attempt(left_star(s,fun(g) = srepeat(s,g)))
6  def outermost(s) :: (∀ a . (a) -> some(a), b) -> some(b)
7                   = srepeat(oncetd(s))
```

From these we can build higher order strategies, including strategies that traverse the whole value (not just the immediate children). Listing 7.17 shows encodings of four such strategies. These are direct translations of the definitions in Stratego with the exception of the enforced thunks in `oncetd` and `repeat`. These are required to stop the call by value semantics of DGEN from executing an infinite loop when expanding either of these. The `attempt` function (which is called `try` in Stratego, we use the Kiama [80] name since this code is ported from Kiama) creates a strategy that tries the strategy `s` and which will leave the input unchanged if that fails. The `oncetd` function will create a strategy that will try s on the top-most value and then, if that fails, try again on one of the children. It repeats this process until it finds some value to which it can be successfully applied. The `srepeat` function (which is called `repeat` in Kiama and Stratego, a name which clashes with a list function in DGEN) will repeatedly apply `s` until it fails. In other words, this strategy will be re-applied if it succeeds, until there is failure. The `outermost` function uses the above strategies to construct one that will will repeatedly apply `s` from the top of a value down. This means that if `s` succeeds *anywhere* in the value, it will be applied repeatedly, until it fails.

We demonstrate this strategic-rewriting module with an example, again ported from Kiama, which evaluates lambda calculus expressions with explicit substitutions [78] (Listings 7.18 and 7.19 and 7.20).

Listing 7.18: Datatype definition for a lambda calculus with explicit substitutions

```
1  adt exp() = Num(int)
2            | Var(string)
3            | Lam(string, exp())
4            | App(exp(), exp())
5            | Sub(exp(), string, exp())
```

Evaluation proceeds as constant iteration of the next possible *small-step*, applying the small-step rules to the outermost available reducible expression on each iteration. The `one_step` function encodes the small-step rules and `eval` encodes the iteration. The iteration uses the strategy combinators to first apply the small-step rules to the top-most reducible expression - creating a new expression - then repeating this process until there are no more reducible expressions.

The `outermost` function uses `one`, so it too has a higher-ranked argument. This requires us to extend the (monomorphic) small-step rule with one of the polymorphic strate-

Listing 7.19: One step of evaluating a lambda expression with explicit substitutions

```
def one_step(e) :: (exp()) -> some(exp())
     = case [e] of
          { [App(Lam(x,e1),e2)]    -> Some(Sub(e1,x,e2))
          ; [Sub(Var(x),y,n)]      -> if (x s== y)
                                        then Some(n)
                                        else Some(Var(x))
          ; [Sub(Lam(x, m),y,n)]   -> Some(Lam(x,Sub(m,y,n)))
          ; [Sub(App(m1, m2),y,n)] -> Some(App(Sub(m1,y,n),Sub(m2,y,n)))
          ; [Sub(m,x,n)]           -> if (elem(string_equality,x,fv(m)))
                                        then None()
                                        else Some(m)
          } otherwise              -> None()

def fv(t) = case [t] of
                 { [Num(x)]       -> []
                 ; [Var(x)]       -> [x]
                 ; [Lam(x,e)]     -> remove(string_equality, x, fv(e))
                 ; [App(m,n)]     -> append(fv(m),fv(n))
                 ; [Sub(m,x,n)]   -> append(remove(string_equality, x, fv(m)),fv(n))
                 } otherwise      -> error "partial definition error in fv"
```

Listing 7.20: Evaluating a lambda expression to normal form

```
def g_one_step() = one_step ▷ fail
def eval(d) = outermost(g_one_step,d)
```

gies (in this case `fail`) to get a polymorphic strategy we can try on the whole expression. This is an example of our type system flexing its muscles. Since `case` expressions (and functions built entirely from them) are defined on only one datatype, we must tell the compiler what function to run on other types. In a typical strategic programming system, an attempt to process an unknown type (or constructor) causes a fail value to be emitted. Our type tells us there are actually two choices, failure *or success*. Our type system enforced that we make this decision, and function extension (▷) allowed us to choose either option depending on our requirements.

7.3 Limitations

7.3.1 Can't Give Best Possible Type to geq

We have had to give generic equality the type `(a,b)-> Bool` when we would prefer to give it `(a,a)-> Bool` since it can never have the value `true` for values of different types. We can't give it the preferred type because there is no universal relationship between a type and the type of values stored in constructed values of that type. For example, knowing that `x` is of type `tree(a)` (where we have defined `adt tree(a)= Branch(a,tree(a),tree`

(a)) | Leaf(a)) tells us that kar(x)'s type could either be (tree(a))-> tree(a) or (a)-> tree(a). We can't know which. We can't even make broader statements about the type of kar(x). For example, given the datatype declaration

```
1  adt either(a,b) = Left(a) | Right(b)
```

and x of type either(a,b) plus y of type either(a,b), we *cannot* say that the type of kdr(x) and kdr(y) are the same. One may be a while the other is b, or vice versa. Thus a generic function like generic equality can make *no* assumptions about the potential relationships between the types of its two arguments. This forces the recursive calls to generic equality to be of type (a,b)-> Bool which forces the overall type to be (a,b)-> Bool.

This same problem appears in related systems such as SYB [59] and in dependent type theory where equality with the type (a,b)-> Bool is called *John Major's equality*[70]. The nomenclature is a joke referencing former British Prime Minister John Major's idea of social equality where the working class are allowed to believe they might become equal to the upper class in the absence of any mechanism by which that could happen.

7.3.2 Can't Encode Generic Map

Many authors have studied language features to provide a generic variant of mapping over each element of a list. Such a function would map over each element in *any* structure, applying a function at every node. The difference between this generic map and generic update is that generic map produces outputs which are structurally identical to the input data structures but which differ from them in the types of values stored at the nodes. Generic update produces output which are the same structure as the inputs and have the same types at all values. For example, one might like to change every character to its integer representation, which requires a change of type at each node. For this to be type safe, generic map needs to have a signature like

```
1  gmap(f,v) :: (a -> b, s a) -> s b
2            = ...
```

Notice that the type constructor s is a variable in the type definition. For example, s might be list for one application of gmap and set for another. Many datatype generic tools are capable of doing this, but DGEN is not. We can't do this because the universal representation of data that we are using (the spine view) cannot support it [38]. The problem arises because the generic function f must apply at all nodes of the structure. For this to be safe, the polymorphic part (recall this is the right hand argument to the function extension operation) must be identity. This forces the type type of f to be $\alpha \rightarrow \alpha$, making it impossible

to define `gmap`. One possible approach to correcting this is to enable another view on data (one which can support type variables) and to create a new type of `case` expression to discriminate upon it. This is the approach taken in bondi (section 14.4 [42]). Another possible approach is the "lifted spine view" [37] which instead *supplements* the spine view with the information required to encode more datatype generic examples.

7.3.3 Can't Encode Generic Read

A *generic read* function is one that can consume serialised data and turn it into a value for *any* datatype. It is the reverse of generic show. As it currently stands, the spine view of data we are using is not expressive enough to allow us to write a generic read operation. The same techniques that show promise for solving the generic map problem look likely to solve this problem as well. In particular, the "type spine view" of [37] has already been shown to allow encoding of generic producers like generic read.

7.4 Summary

In this chapter we have seen that DGEN is able to compile a great variety of generic programs, including strategic-rewriting and many datatype generic programs as well as our original eight snippets. We have also shown that the explicit spine view gives another front upon which to tackle the expression problem. We also noted that the specific spine view we are using is unable to compile a few commonly desired datatype generic functions. We noted earlier that, although the explicit spine view is used in a few other systems, our is the first compiled account of it. To our knowledge this chapter is the most comprehensive survey of what can be encoded with the explicit spine view, further contributing to the literature on this technique.

Chapter 8

Evaluating Compilation

In this chapter we demonstrate how successfully DGEN has achieved our goal of being a generic function *compiler*. We do this by evaluating three primary characteristics of the DGEN compiler, these being: the speed of compiled programs, the memory use of compiled programs, and the opportunities for optimisations. The first two are done by comparing the generic function overhead in DGEN to the same overhead in other generic function tools. We will show that DGEN performs significantly better than SYB, which is the most closely related generic functional library available for functional language compilers. To demonstrate optimisation opportunities we will implement one simple optimisation and discuss how to implement an existing optimisation from another compiler. In this chapter we also quantify the memory overhead that our encoding of `kar`/`kdr` introduces to generic programs.

8.1 A Note About our Benchmarks

The benchmarking of programming language compilers is an art which requires careful experimental design. To that end we have designed our benchmarks in an incremental fashion, to avoid problems that arise from direct comparisons of technically incomparable compilers. For example, GHC generates much faster non-generic code than DGEN does, so any naive approach to making direct performance comparisons between their generic facilities is doomed to failure. So we must adopt a method for making these assessments which allows us to faithfully compare results measured against different baseline scales. Specifically, we measure the percentage slowdown experienced when moving from a non-generic piece of code to an equivalent generic program, *as compiled by the same compiler*, and then compare these relative results from one compiler/toolset to the next. Secondly,

we benchmark on more than one type of generic program since different generic programs have different performance characteristics.

Thirdly, a certain input value could have a short-cut to the result in some generic programs, so we test each benchmark on multiple inputs. We choose to use differently sized inputs because an interpolation line between them will give a clear idea of performance for many different inputs. Finally, to avoid inadvertent selection of favourable inputs, all input values are randomly generated. While this last decision does introduce some residual noise into our experimental results, this inconvenience is more than outweighed by the reassurance provided by this approach. As we shall see, the signal-to-noise ratio is more than good enough to allow us to clearly identify the trends revealed by our experiments.

8.2 Compiled Speed

Our primary concern regarding speed of compiled programs is to determine the overhead of using generic code in comparison to monomorphic or (standard) polymorphic code. It has been shown [68, 93] that generic functions can have a significant overhead. One of the claims of this thesis is that we have significantly reduced this overhead in our implementation, primarily by implementing the most costly parts of the process in the compiler and run-time, where there is maximum information about the program and more opportunities to customise behaviour.

8.2.1 Maintaining Compilation to `switch` Statements.

Discriminations with `case` expressions are everywhere in functional code. In large part they drive the computation forward, indeed it can be said that they are the only source of evaluative impetus in lazily evaluated languages. Efficient compilation of these expressions is absolutely vital to quickly executing functional programs.

Such expressions are compiled into very fast discrimination, or branching, constructs in the target language. For example, if the target language is C, they are compiled into `switch` statements. If the target language is some machine code, they will probably be compiled to jumps. Compilation to such efficient code is only possible because of a few characteristics of the `case` expressions in the internal languages of the compiler (as opposed to those in the source language):

- Constructors are small integers. Using such a small value for a constructor is only

possible when constructors are unique per-datatype. If they needed to be unique per-program, we would need a more complex encoding.

- The `case` expressions only discriminate on constructors. Thus patterns that discriminate on something else (i.e. the *structure* of the value as in application pattern matches) can't be part of these `case` expressions.

By carefully choosing and constructing our solutions for polymorphic functions with specific behaviour and for structure agnosticism we have maintained these characteristics in DGEN's internal languages (CORE and SUPERCOMB). We will see that there *is* a small cost to be paid when using generic functions, but had we not maintained the above characteristics in our compiler, there would have been a *great* cost to be paid in *all* code. As things stand, our primary weapon against slow code, compilation to `switch`, has been maintained. Of course, we have not been able to do this without making a few trade-offs, most notably we had to exclude the option of extension typing (see Section 3.3.1). In its place we created the extension operator, making it general enough to replace that mechanism. We also used careful compilation and novel typing rules to allow application pattern matches to be converted into non-case expressions (i.e. `ispair`, `kar` and `kdr`).

8.2.2 Generic Libraries Cause Slowdown

We follow the methodology of Magalhães [68], which is sufficiently similar to Rodriguez [93] that his results are also enlightening. We define a generic function and a monomorphic version of that function for a specific data type. The two programs perform the same computation, and traverse values in the same way, but the monomorphic version uses the language's fastest mechanism (pattern matching on constructors) while the generic version uses the facilities provided by generic programming tools. In our case the facilities comprise application pattern matches and function extension.

The most appropriate comparison for this work is with SYB, which runs between 600% and 8600% the speed of the monomorphic version [68, 93]. As we will soon see, DGEN achieves between 290% and 310% of monomorphic speed.

8.2.3 DGEN has Relatively Little Slowdown due to Generics

Each of the following benchmarks takes a data value of a certain size which is generated with an algorithm that uses some randomness to ensure that each generated value is different to the last. The randomness also affects the shape of the generated data (i.e. some randomly generated values will be shallow but broad, others will be deep and narrow).

This removes the possibility that we have inadvertently chosen a favourable shape of data for the benchmark. It runs both the generic and non-generic code over this value 10 times and sums the run-times. To be clear, for each size of input the generic and non generic code is run on the same value. The generator then generates the next size value and the process is repeated. We choose to linearly increase these sizes because an accompanying linear increase in run-times is a useful sanity check. The graph of generic run time vs input size is superimposed on the graph of non-generic run time vs input size on a single chart (show on the left of each figure). For each input size the generic run-time divided by the monomorphic run time is plotted on another chart (shown on the right of each figure). This chart shows some noise due to the random input data so we give a line of best fit on that chart as well. We will include the details of the best fit calculations in each section. This line of best fit is the slowdown due to generic code in DGEN for the example and inputs in question.

Program execution time was recorded using the unix `time` command and the benchmarks were run on a MacBook Pro model 5,5 (Intel Core Duo 2.26Ghz, 4GB Memory). We benchmarked both the salary update (a generic traversal) and the name analysis (a generic query) snippets.

Salary Update

Figure 8.1 shows our benchmark results for the salary update snippet from Figure 2.3. We see a slowdown a little less than 3 times (precisely, 2.92 times). The best fit was calculated for a straight line in two dimensional space ($y = m * x + b$) and, the best fit calculations give a gradient of approximately 0[1] indicating we are recording a consistent slowdown.

Name Analysis

Figure 8.2 shows our benchmark results for the name analysis snippet from Figure 2.4. We see a slowdown a little more than 3 times (more precisely, 3.09 times). The best fit was calculated for a straight line in two dimensional space ($y = m * x + b$) and, again, the best fit calculations give a gradient of approximately 0[2].

8.2.4 Comparisons to Other Tools

In this section we compare the results we have collected for DGEN with similar results for other tools capable of processing generic functions. The ideal comparison would be

[1] actually, 0.000131332
[2] actually 2.32231e-05

Figure 8.1: Slowdown due to generic code for salary update snippet

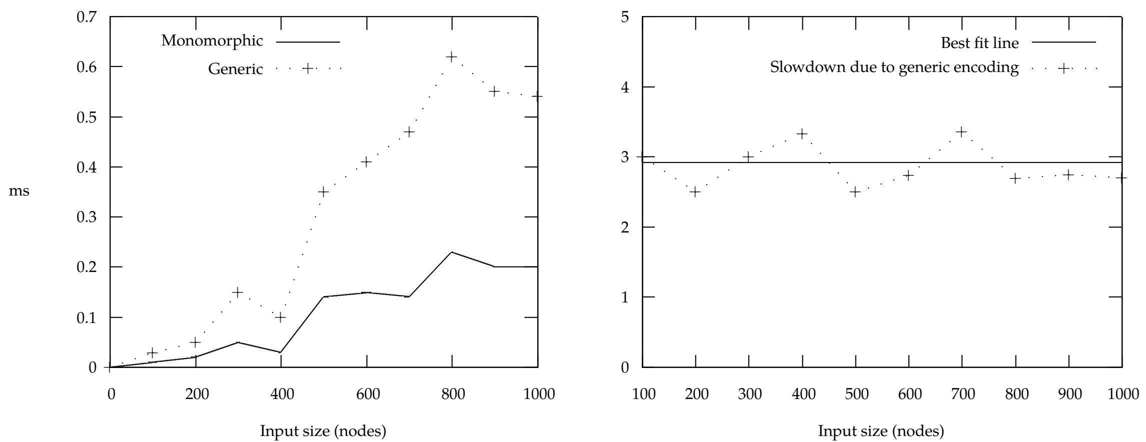
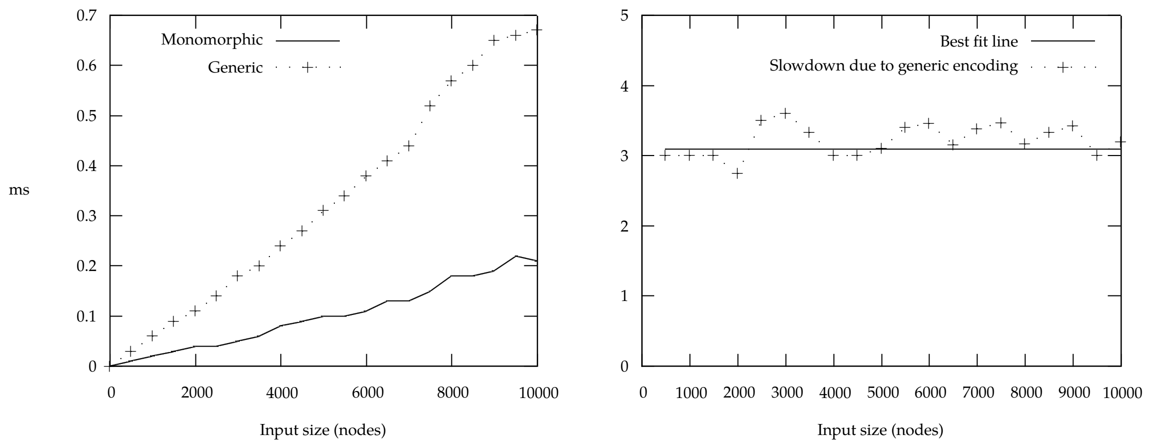


Figure 8.2: Slowdown due to generic code for name analysis snippet



between different tools running the same programs on the same machines, but there are two reasons we don't do this:

1. As noted by Hinze [38], not all programs are expressible in all tools. Even when they are, the encodings can differ substantially.

2. We can make use of existing published benchmarks which give a valid point of view, making it unnecessary for us to re-run all the benchmarks ourselves.

Thus we rely on other published benchmarks for this comparison. As we mentioned earlier, there are two published benchmarks of generic functions which use the same methodology we used. The first, by Rodriguez [93] is part of a comprehensive study of datatype generic libraries for Haskell. The second is by Magalhaẽs, Holdermans, Jeuring and Löh [68] and was done as part of research into optimising generic programs in

Haskell. Ordinarily, taking results from two different benchmarks and comparing them to a third would give meaningless results because the machines running the benchmarks are different. However, these benchmarks measure *slowdown*, which is independent of the machine's performance. However, the following caveats apply to using these three result sets together:

- Each uses a different compiler. Magalhaẽs tests with multiple compiler flags and different versions of GHC. Where we have the choice, we use the results from the latest measured version of GHC with the fewest optimisations enabled. These are the most appropriate comparisons because DGEN performs only one optimisation (Section 8.4).

- Each uses different programs. This is unavoidable because we can't encode the same function in all tools, but in comparing different benchmarks by different authors, we exacerbate this problem.

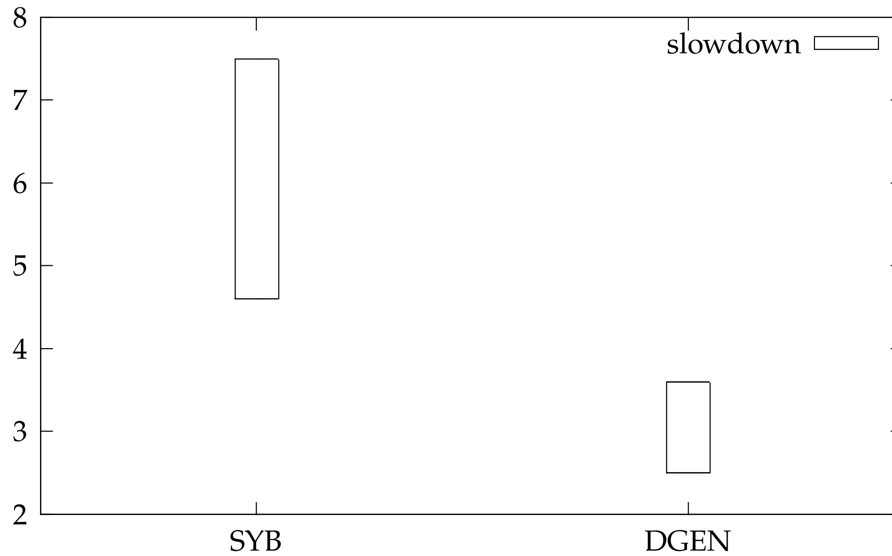
- Different tools are tested in different benchmarks. Some of the tools are tested in both benchmarks, some in only one.

- Only Haskell libraries are benchmarked. We have no results for bondi, RhoStratego or Stratego for example.

The benchmark literature ([93, 68]) shows that individual tools can have widely varying performance on different generic tasks. This means, for example, we can't directly compare the slowdown from a generic traversal in EMGM to the slowdown for generic query in Instant Generics. However, every result is *indicative* of the performance of a particular tool. Thus we have split the recorded slowdowns due to generics that exist in the literature into two groups, the *immediately comparable results* and the *less comparable results*. The immediately comparable results we have found are for generic traversal in the style of our salary update snippet. When we present the results we will show both these immediately comparable results *and* those that are less comparable. We do this because we have a paucity of relevant data and can't afford to discard any indicative data, but we want to highlight the most relevant data.

For this comparison, we chart the range of slowdown results given in the literature for each generic programming tool. The range is made up of two parts: a hollow section showing the immediately comparable results to our own, and a hatched section showing the range of less comparable results. For example, in Figure 8.4 you will see that we record two ranges for MultiRec; the open box is the range of slowdown measurements in imme-

Figure 8.3: Comparison of Slowdown due to Generics for DGEN and SYB



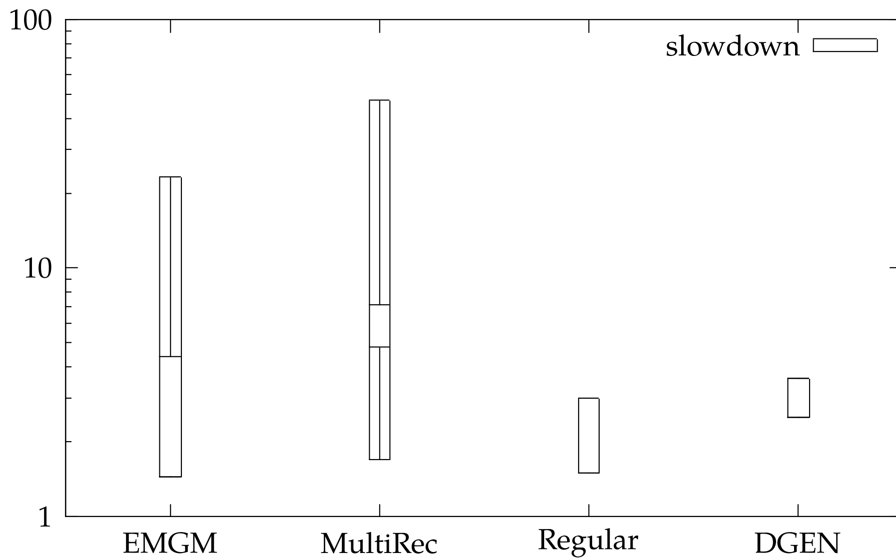
diately comparable benchmarks, while the hatched areas above and below show that a greater range is recorded if we take all benchmarks into account. The result for Regular has no hatched results because the benchmarks recorded in the literature are all immediately comparable to ours. EMGM has no hatched area below the hollow area since the lowest recorded slowdown was for a immediately comparable benchmark.

We break the comparison up into two parts. First we compare DGEN to SYB, which is the only benchmarked tool that works similarly to ours. Recall that our function extension operation is somewhat like SYB's `extT` and `extQ` and that we use an explicit spine view similar to SYB's implicit spine view. Secondly we compare DGEN to tools that make heavy use of Haskell's type classes, sometimes explicitly to remove the overhead of generic programming. There are a number of tools for which there *is* benchmark data in the literature, but where none of that data is immediately comparable to ours. We have chosen not include these results in the analysis that follows because less comparable results are only useful when immediately comparable results are also available to *ground* the analysis.

8.2.5 Comparison to SYB

Figure 8.3 compares the results we obtained for DGEN with those in the literature for SYB. We have excluded one outlier from the SYB data (a recorded slowdown of 8600% recorded in [93]) because it dominates the chart (even with a logarithmic scale), making any meaningful comparison impossible. There are both immediately comparable and less compa-

Figure 8.4: Comparison of Slowdown due to Generics for DGEN, EMGM, MultiRec and Regular



rable results for SYB but the full range of recorded slowdowns are within the immediately comparable results. You can see that DGEN generated code has lower overhead for generics than SYB code for all available benchmarks (i.e DGEN's worst slowdown is better than SYB's best slowdown). This result is expected since DGEN is, in some ways, a translation of SYB from a library into a compiler. We have been able to use efficient techniques in the run-time where SYB needs to use library routines. For example, our type check `typeOf` is a relatively efficient run-time operation where SYB needs to attempt a cast and then branch based on the result of that cast to achieve the same effect.

8.2.6 Comparison to Type-Class based Haskell Libraries

Generic programming libraries for Haskell use advanced type-class techniques and exploit GHC's mature compilation algorithms to achieve very efficient generic executables[3]. Using type-classes, the type system can ensure that the right function for the node in question is readily available without requiring run-time type checks. This can significantly reduce the run-time overhead of generic code. We wish to show that DGEN is able to achieve comparable efficiency without using any advanced features and with no generic-specific optimisations.

Figure 8.4 shows the range of slowdowns recorded in the literature for three Haskell

[3]SYB does use type-classes, but in a very different manner and so is not included in this group.

generics libraries and the results we recorded for DGEN:

EMGM EMGM [37, 10] is a large and mature library for generic programming in Haskell which uses the universal representation of data approach where datatype representations are defined in typeclasses.

MultiRec and Regular MultiRec [93] and Regular [68] are very similar Haskell libraries for generics. They both use type families to represent the structure of datatypes.

Conspicuously missing from this list is Instant Generics [13]. We have found no directly comparable benchmarks for this library and suspect it may be faster than any of those shown in Figure 8.4. The benchmarks that have been published [13] show that Instant Generics can get very close to the minimum possible slowdown (a multiplier of 1) However, given that Regular has slowdowns very close to 1, even better results for Instant Generics will not invalidate our analysis.

Where these tools can make the best use of type-classes we expect to see slowdown approaching one (i.e. no slowdown) but we also expect that the extra encoding will still have some cost for some benchmarks. Since DGEN has a low overhead for all code we expect these tools to sometimes perform better than DGEN and to sometimes perform worse. Figure 8.4 shows that, in general terms, Regular generates faster code than DGEN, MultiRec generates slower code and EMGM generates code of comparable speed to DGEN. These results correspond with our expectations and validate our claim that DGEN generates efficient generic code without type-classes or generic-specific optimisations. Recall (page 99) that DGEN does not even optimise calls to `kar` and `kdr` in the body of a `ispair`. This and many other optimisation opportunities that could speed up generic code are still to be exploited in DGEN. All the results shown for Haskell libraries have been achieved with the extremely sophisticated and mature compiler GHC and in most cases they use its specific extensions to Haskell98. Thus having comparable results from DGEN validates that the techniques we have used generate efficient code.

8.2.7 Summary of Speed Benchmarks

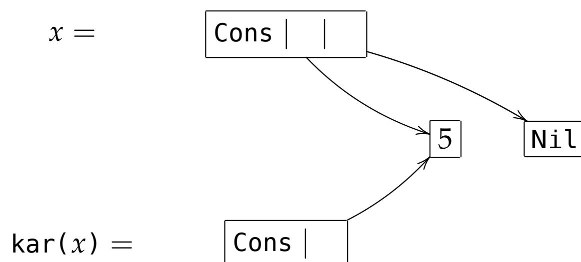
We have shown that using generic functions has some overhead compared to monomorphic encodings of the same function. We have shown that the same programs can have significantly greater overheads when using comparable generics in libraries. Magalhães et. al. have shown that the overhead of the library approach *can* be reduced for some programs with some libraries on some compilers. However DGEN has relatively little overhead

for any of its generic functions and the approach given in this thesis can be added to *any* functional language compiler.

8.3 Memory Use due to Generics

The other performance problem that generics introduce is memory overhead. We know of no published accounts of memory benchmarks for generic functions, but the methodology we used for speed benchmarks is applicable. Memory is recorded by instrumenting our program to report the maximum memory use during program execution. We are able to do this because we generate the C code into which the program gets compiled. Although it is beyond the scope of this thesis to perform a complete benchmark of memory use for generic programming tools, we have replicated two of our benchmarks with GHC and SYB to provide some context in which we can understand our own results.

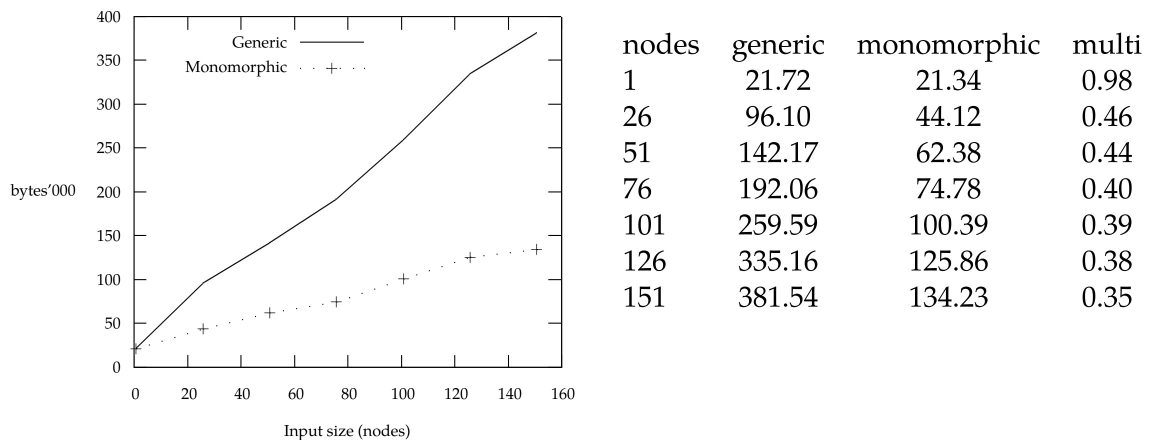
The extra memory use for generic code comes from the `kar` operation. It replicates the data that was passed into it to create the same data but applied to one less argument. Were we to replicate the full input datum (a so-called deep copy) we would get memory use polymorphic in the size of the input data. However, we only replicate the particular constructor we are working on, filling its arguments with the same datums as were in the input parameter. For example, given `x = Cons(5, Nil)`, then `kar(x)` at run-time is, shown diagrammatically,



Only the tagging constructor is replicated. In fact there is a little more overhead for each tagged datum, particularly the arity and the type string are stored.

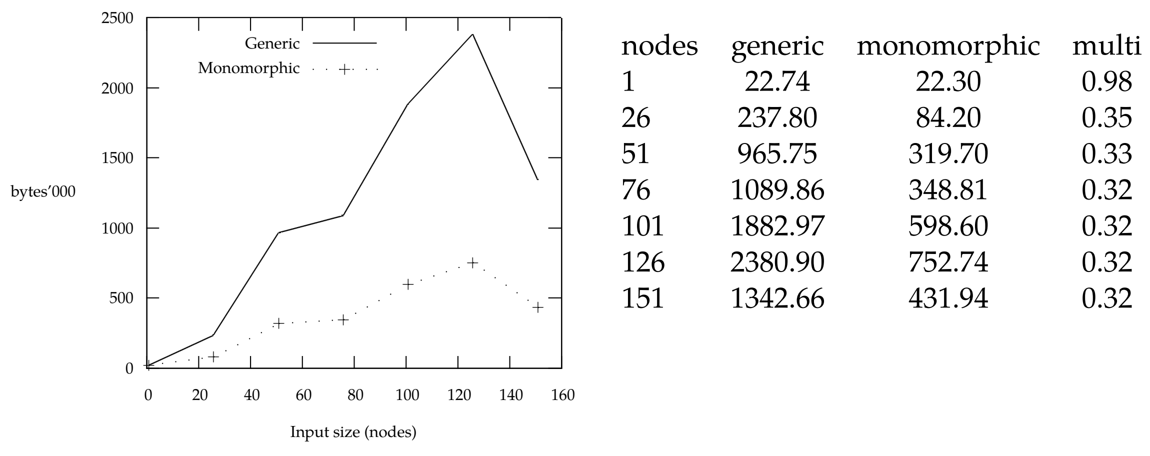
Figures 8.5, 8.6 and 8.7 give the memory overhead results for the name analysis snippet, salary update snippet and generic equality respectively. For each snippet we give a chart of the memory use for the generic and monomorphic variants and show the full results in a table. Figure 8.8 gives a summary of the memory overhead for each example we ran. The memory overhead for generic equality is much smaller because this function

Figure 8.5: Memory use comparison for name analysis snippet



nodes	generic	monomorphic	multi
1	21.72	21.34	0.98
26	96.10	44.12	0.46
51	142.17	62.38	0.44
76	192.06	74.78	0.40
101	259.59	100.39	0.39
126	335.16	125.86	0.38
151	381.54	134.23	0.35

Figure 8.6: Memory use comparison for salary update snippet



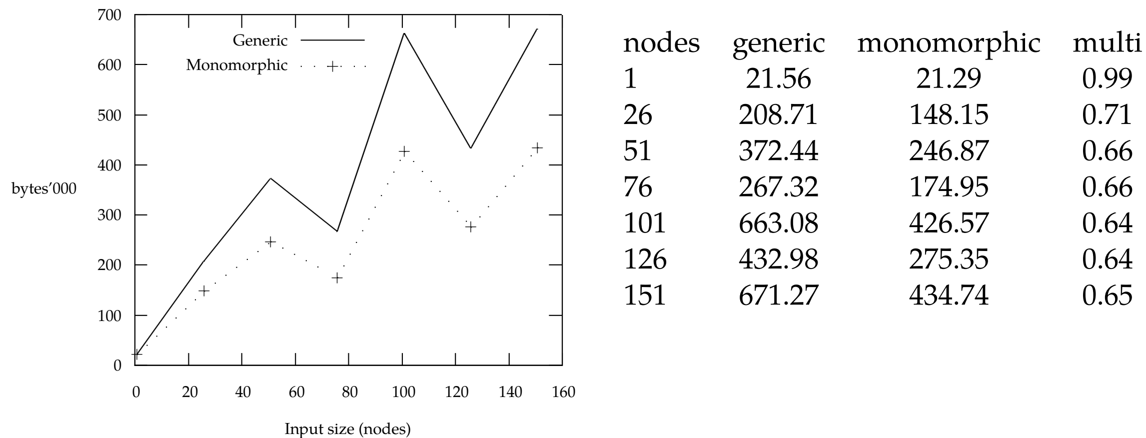
nodes	generic	monomorphic	multi
1	22.74	22.30	0.98
26	237.80	84.20	0.35
51	965.75	319.70	0.33
76	1089.86	348.81	0.32
101	1882.97	598.60	0.32
126	2380.90	752.74	0.32
151	1342.66	431.94	0.32

is not pasting the data back together, an operation that also has memory overhead.

8.3.1 Comparison to SYB

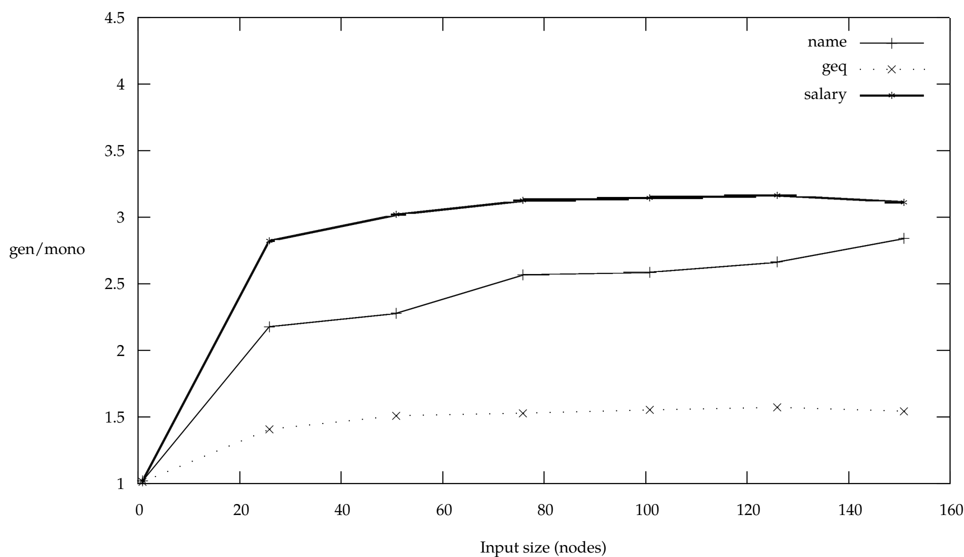
The name analysis and salary update snippets were ported to SYB/Haskell, with corresponding monomorphic versions, and the benchmarks were run in GHC version 6.12.1. SYB and DGEN are so similar that we are able to use almost identical encodings for those two snippets, making the benchmarks we present here completely equivalent to the DGEN memory benchmarks above. The memory use of the program was recorded using GHC's profiling capabilities. The programs were compiled with profiling enabled and the `-p` flag was given to the runtime system requesting basic profiling output. The memory use we record for each program is the total memory allocation (not including profiling overhead)

Figure 8.7: Memory use comparison for generic equality



nodes	generic	monomorphic	multi
1	21.56	21.29	0.99
26	208.71	148.15	0.71
51	372.44	246.87	0.66
76	267.32	174.95	0.66
101	663.08	426.57	0.64
126	432.98	275.35	0.64
151	671.27	434.74	0.65

Figure 8.8: Memory overhead summary



reported by the runtime. Figure 8.9 shows the memory use by the GHC-generated executables for the name analysis snippet. The chart shows the monomorphic memory use divided by the generic memory use.

You will notice that the chart has similar shape to those we recorded for DGEN but converge on a much higher multiplier. Our benchmarks show the generic code memory multiplier converging to 140. For salary update, the results of which are shown in Figure 8.10, we again record the same shape for memory overhead due to generics and a much larger overall overhead than we recorded for DGEN. For the salary update snippet the benchmarks show memory use for generics converging to 24 times that of monomorphic code.

Figure 8.9: Memory use comparison for the name analysis snippet with GHC and SYB

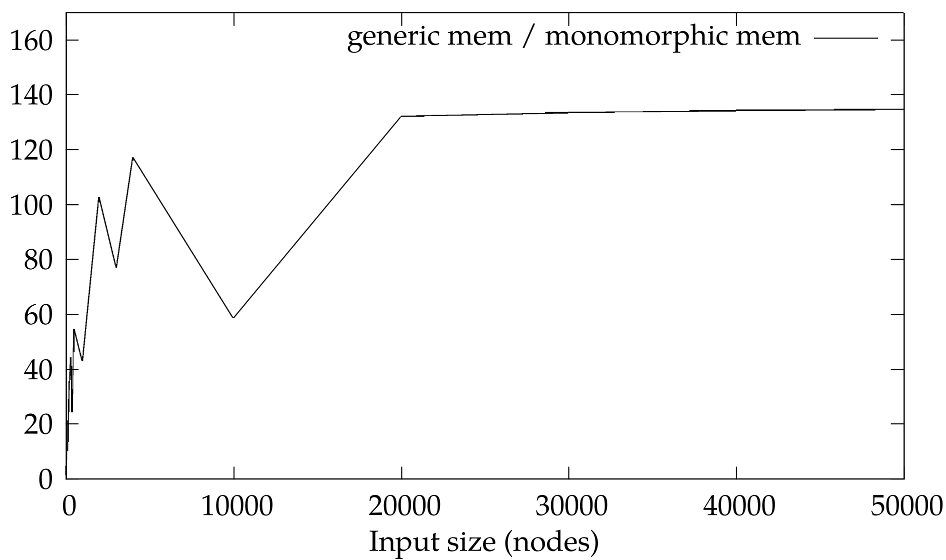
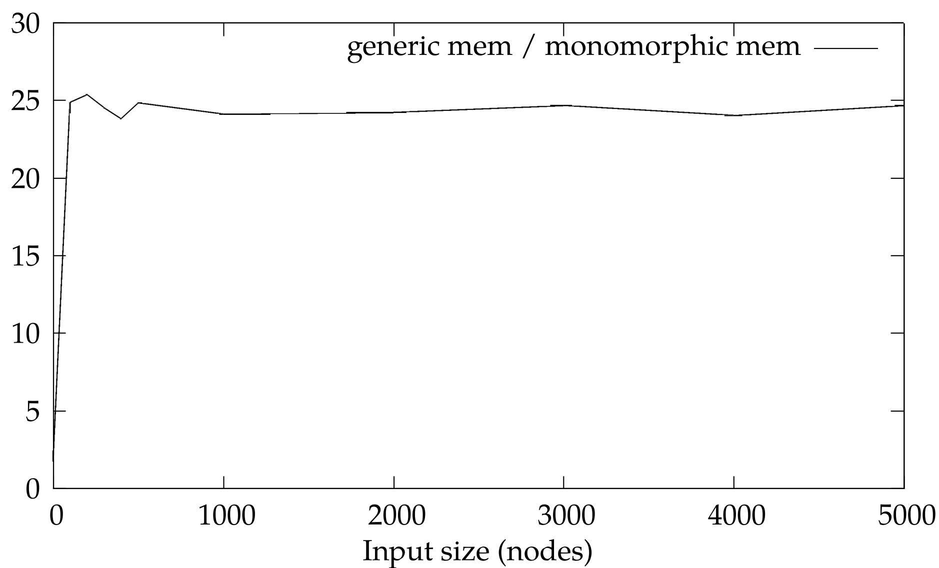


Figure 8.10: Memory use comparison for the salary update snippet with GHC and SYB



For the benchmarks we have been able to replicate in SYB, the overhead for generics in DGEN is 2.7 – 3.1 times while the overhead in GHC/SYB is 24 – 125 times. These results are for just one generic toolset on one compiler, however, they do suggest that DGEN is relatively efficient in its use of memory for generic code.

8.4 Optimisation Opportunities

One of the primary reasons to prefer compilation over interpretation is that it provides the language implementer with a more immediate opportunity to optimise the speed and space performance of the executed code. In this thesis we don't intend to discuss the implementation of a great number of optimisations, instead we content ourselves with demonstrating that our compiler is capable of supporting a full range of standard optimisations in a way which does not interfere with, and indeed directly enhances, its compilation of generic code. To do this, we present one simple optimisation that is made possible by the fact that we use a super-combinator representation and we demonstrate the significant speed gains it provides. In this section we also highlight an existing functional compiler optimisation that one might reasonably expect to be compromised by the use of the explicit spine view. In that case, we show that this optimisataion continues to operate identically on monomorphic code emitted by DGEN and indeed is immediately and usefully applicable to the forms into which it compiles generic functions.

8.4.1 An Optimisation: Trivial Super-Combinator Elimination

A *trivial super-combinator* is one that simply calls another super-combinator as its body, passing on – unchanged – any arguments that were passed to it. Clearly we would like to remove trivial super-combinators since super-combinators are, broadly speaking, translated into function calls. For example, Listing 8.1 will be lifted, using the algorithm we gave in Section 4.4, to the SUPERCOMB program in Listing 8.2. This optimisation will result instead in the SUPERCOMB program in Listing 8.3

Listing 8.1: CORE expression to be lifted

```
1 main = \s. \x. \y. case s of {True() -> x; False() -> y}
```

Listing 8.2: SUPERCOMB into which Listing 8.1 is lifted without any optimisations

```
1 SC0 x = x
2 SC1 y = y
3 SC2 s x y = switch s of {True -> SC0 x; False -> SC1 y}
4 SC3 s x y = SC2 s x y
5 SC4 s x y = SC3 s x y
6 SC5 s x y = SC4 s x y
7 in
8 SC5
```

Listing 8.3: SuperComb into which Listing 8.1 is lifted *with* trivial super-combinator elimination

```
1  SC0 x = x
2  SC1 y = y
3  SC2 s x y = switch s of {True -> SC0 x; False -> SC1 y}
4  in
5  SC2
```

The super-combinators SC5, SC4 and SC3 are trivial super-combinators and removing them could[4] speed up the program. Care must be taken however to ensure that we don't eliminate super-combinators that are used in an argument position. Since DGEN has call-by-value semantics, there is a semantic difference between (for example)

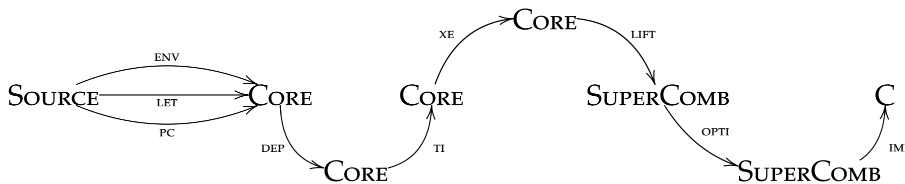
```
1  SC1 = SC3
2  SC3 s = SC4 s (SC5 (SC2 s))
3  SC2 s g = SC1 s g
4  ...
```

and

```
1  SC3 s = SC4 s (SC5 (SC3 s))
```

The obvious difference between the two is the removal of the trivial super-combinators SC2 and SC3. However, by doing this we have changed a (potentially) terminating program into a (certainly) non-terminating one. The call-by value semantics mean that (in the second one) we can't evaluate SC3 until we have evaluated all the arguments to SC4, which involves evaluating SC3, causing an infinite loop. So we have a side-condition on the optimisation; you can only eliminate super-combinators that never appear as an actual parameter.

To achieve this optimisation, we add another phase to the compiler. It runs after lambda lifting and before the conversion to imperative code.



This translation, which we denote $\overline{kd} \mid kp \Rightarrow_{opti} kp$, takes as inputs a set of super-combinator definitions that may benefit from the optimisation, the program in which they live, and will return an optimised super-combinator program. Thus we need to prime

[4]We say *could* because it depends on what optimisations are done in other phases of the compiler.

Figure 8.11: The trivial super-combinator elimination algorithm

(Empty)

$$\emptyset \mid kp \Rightarrow_{opti} kp$$

(TrivialSuperComb)

$$\frac{\overline{kd} \mid [kn/kn']kp_{-kn} \Rightarrow_{opti} kp' \qquad kn \notin ACT(kp)}{(kn\ \overline{x} = kn'\ \overline{x})\ \overline{kd} \mid kp \Rightarrow_{opti} kp'}$$

(Default)

$$\frac{\overline{kd} \mid kp \Rightarrow_{opti} kp'}{kd_1\ \overline{kd} \mid kp \Rightarrow_{opti} kp'}$$

the translation by passing it the un-optimised SuperComb program and all the super-combinators within it in a set. Note that kp_{-kn} is the super-combinator program kp with the definition of the super-combinator kn removed.

Figure 8.11 gives the algorithm for trivial super-combinator elimination. If the set of super-combinators is empty, we simply return the original program. Otherwise we look at the first super-combinator in the set to see if it matches our description of a trivial super-combinator. If it does, we replace all calls to it in the right hand sides of super-combinators in the program with the super-combinator it calls. The side condition states that we only do this if the super-combinator in question is not an actual parameter in the program ($ACT(kp)$ is the set of all actual parameters in the program). We then recursively run the same translation on the set with one less super-combinator in it.

While this is a very simple optimisation, it gives outstanding results. Figure 8.12 charts the run-time of the salary update and name analysis examples on variously sized inputs, with and without this optimisation. It shows that this optimisation consistently provides a shorter run-time. Table 8.1 shows more detail of the results showing that the average speedup[5] due to this optimisation is significant on *all* the programs we have tested. The best speedup is 50% and the worst is 0% (which occurred only on quite small inputs where the total run-time was within the expected margin of error for our timing system). Notice also that this optimisation is effective on both monomorphic and generic code. The benchmarks we have done indicate it may be more effective on generic code, but we need more analysis to be sure of this.

It is the internal structure of the compiler that has exposed the optimisation and made it so easy. If we had not taken the standard step of converting to super-combinator form during compilation, getting this speed gain would have been very much harder.

[5] Where *speedup* is $(u - o)/u$ if u is the unoptimised run-time and o is the optimised run-time.

Figure 8.12: Speedup due to trivial super-combinator elimination algorithm

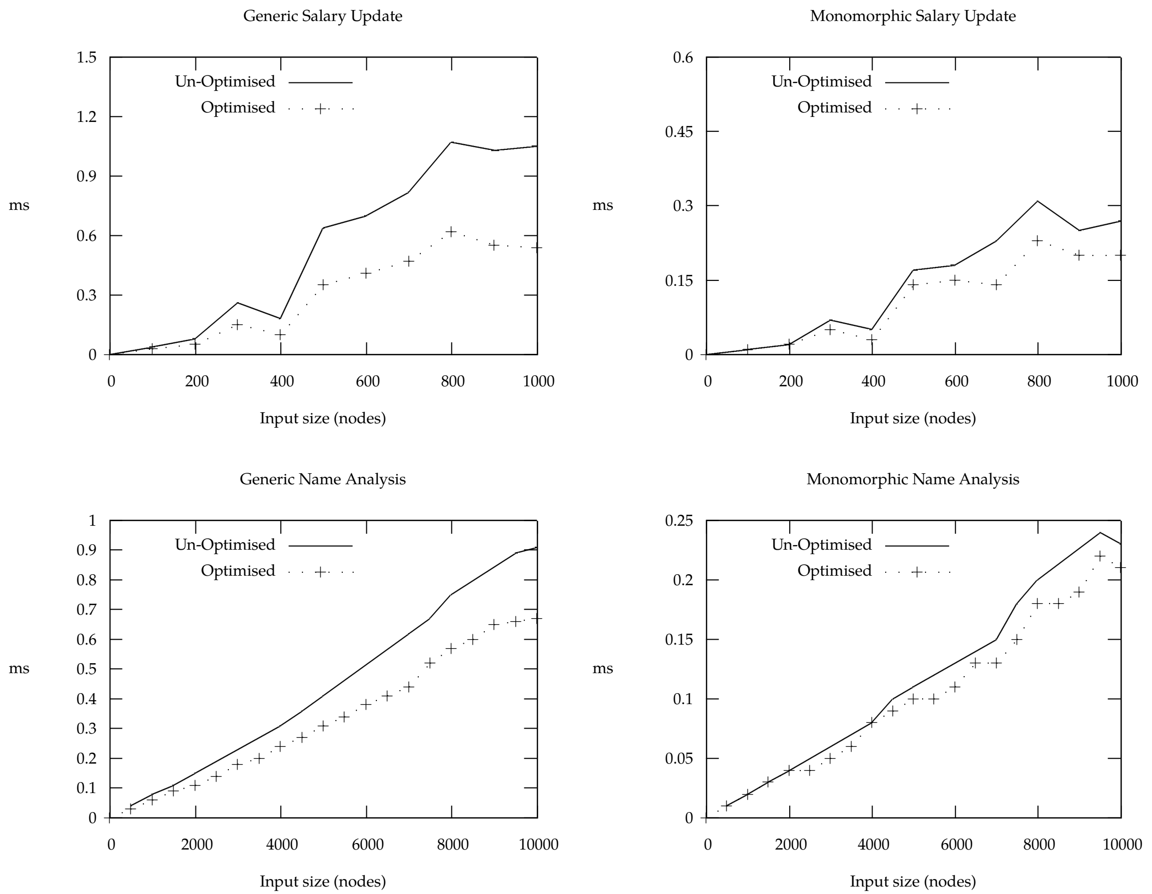


Table 8.1: Minimum, average and maximum speedups due to trivial super-combinator elimination in various DGEN programs

program	min speedup	avg speedup	max speedup
generic name analysis	0.18	0.24	0.29
monomorphic name analysis	0.00	0.07	0.17
generic salary update	0.25	0.43	0.50
monomorphic salary update	0.00	0.22	0.40

8.4.2 A Potentially Compromised Optimisation

Jones and Santos [74] showed how a language very similar to our Core[6] is able to support a set of useful optimisations. We now show that one of these optimisations, *case-of-known-constructor*, can be implemented in our compiler despite the potentially disruptive influence of application pattern matches.

Changing the compilation of `case` expressions could easily slow down the speed at which we can execute them. We have already shown in Section 8.2.1 that we have maintained *compilation to switch statements* and here we show that `case` based optimisations still work. More precisely, we show that our modifications do not disturb the *case-of-known-constructor* optimisation except on functions with generic patterns, and even then in a minimal way.

The case-of-known-constructor optimisation [74] applies anytime the scrutinee of a `case` expression is a datum known at compile time. In this circumstance we know which branch will be taken and can replace the whole `case` expression with the right-hand-side of that branch. Consider the following example from [74].

```
1 case x of
2   True  -> case False of {True -> e1; False -> e2} otherwise error
3   False -> case True  of {True -> e1; False -> e2} otherwise error
4   otherwise error
```

which is transformed by the optimisation into

```
1 case x of
2   True  -> e2
3   False -> e1
4   otherwise error
```

Exactly the same transformation works in Core because we have maintained all the conditions necessary for it, namely:

- `case` works on one scrutinee,
- `case` patterns do no computation, and
- `case` patterns are fully applied constructors.

Our decision to take application pattern matches out of `case` expressions has paid off again. By maintaining the usual semantics for `case` expressions in a core language, we have maintained the chance to apply known optimisations.

[6]Core does not have type abstraction and application, making it more difficult to check the *correctness* of an optimisation but this does not preclude optimisations.

However, there is a *potential* problem applying case-of-known-constructor to Core values. Our rules for compiling away application pattern matches break up `case` expressions, taking what might have been one `case` expression and turning it into two or more. This will reduce the effectiveness of any application of case-of-known-constructor since a particular `case` may have fewer alternatives that can be removed in one application of the optimisation. However, case-of-known-constructor is still applicable to each of the smaller `case` expressions and any compilation scheme which somehow *maintains* the larger `case` expression will necessarily not work over fully applied constructors, destroying *all* opportunities to apply case-of-known-constructor.

8.5 Summary

In this chapter we have shown that our compilation techniques for generic functions can achieve approximately 3 times slowdown on generic code, which is better than the most similar tool and competitive with the fastest tools. Furthermore we have shown that the same techniques require less than 350% memory overhead for generic code when the most comparable tool requires up to 14000% overhead. We have done this while maintaining the applicability of our approach to all programming languages. We have also shown that these same techniques make defining optimisations easy. These achievements demonstrate that adding these techniques to an existing compiler should not significantly affect the performance of the programs it compiles.

Chapter 9

Evaluating the Type System

In this chapter we will prove the soundness of the typing relation which underlies the type inference system in DGEN. We do this for a slightly simplified language (simpler than CORE) which has only single-branched `let`s, `letrec`s and `case`s and in which there are no primitive types. Polymorphic recursion and higher rank types are not relevant questions in a typing relation (they work without difficulty) and thus we have only one form of recursive `let` and no type annotations. However, we include all our other novel type system features: `ispair`, `kar`, `kdr` and function extension ($\triangleright$).

When we gave the type inference algorithm for CORE in Figure 4.5 we used a language where constructors have only one argument. We were able to do this because that simplification does not disguise any of the important features of the type inference algorithm. However, that simplification *would* hide important features of the type relation we present here. Thus for proving the soundness of the type system we use a language where constructors have multiple arguments, as they do in the CORE version we described in Figure 4.2. We begin with the definition of the language in question, shown in Figure 9.1. Figure 9.2 shows the small-step operational semantics of this language. Figure 9.3 shows the typing rules which we wish to prove sound for this language. Note that we denote the substitution of type variable α with the type ρ in the type τ as $[\alpha/\rho]\tau$ and the free type variables of type τ as $TV(\tau)$ (we can calculate the type variables of type schemes and type environments as well). $A_x, x\colon \tau$ denotes the type environment A with any binding for x updated to τ.

Figure 9.1: Modified CORE Language

Type Language

$$
\begin{aligned}
\sigma &::= \forall \alpha . \sigma \\
&\mid \quad \tau \\
\tau, \rho &::= \alpha && \text{(type variables)} \\
&\mid \quad T\ \tau_1 \ldots \tau_n && \text{(constructed types)} \\
&\mid \quad \tau \rightarrow \rho
\end{aligned}
$$

Term Language

$$
\begin{aligned}
e, f, g &::= v && \text{(expressions)} \\
&\mid \quad e\ f \\
&\mid \quad \texttt{let}\ x = e\ \texttt{in}\ f \\
&\mid \quad \texttt{letrec}\ x = e\ \texttt{in}\ f \\
&\mid \quad \texttt{case}\ e\ \texttt{of}\ (K(x_1, \ldots, x_n)) \rightarrow f \\
&\mid \quad \texttt{ispair}\ e\ \texttt{bind}\ (x, y)\ \texttt{in}\ f\ \texttt{else}\ g \\
&\mid \quad e \triangleright f \\
v &::= \lambda x . e && \text{(values)} \\
&\mid \quad K(v_1, \ldots, v_n) && \text{(constructor applied to some arguments)} \\
&\mid \quad v \triangleright v' \\
&\mid \quad sv \\
sv &::= x && \text{(semi-values)} \\
&\mid \quad sv\ v
\end{aligned}
$$

9.1 The Evaluation Relation

As well as the simplifications outlined above, there are some differences between this language and Core which are necessary to allow a proof via the small-step semantics of the language.

Values and Expressions

Expressions are split into those that can potentially be evaluated another step (e), and *values* which can't (v). Expressions that can't be evaluated any further are split into those that could be the result of a program (values, v) and those that are present in a partially evaluated program only (semi-values, sv). It is preferable to have only one class of values since semi-values are not what we would traditionally think of as values in a functional language. If we were developing this language for its evaluation machinery, instead of using it for a type soundness proof, we would like to prove some properties regarding where and when semi-values can occur. For example, it would be prudent to prove that *if e closed,* $A \vdash e \colon \tau$ *and* $e \longrightarrow e'$ *then* e' *is closed* and *if e is closed,* $A \vdash e \colon \tau$ *and* $e \longrightarrow v$ *then v is not a semi-value*. However, we are using this language as a vehicle to show type system soundness and in that respect we can admit semi-values as values.

Can't Restrict Expressions to Variables

Since the evaluation of the language must be described as translations from one expression to another *within* the language, we must allow any term to be the scrutinee of a `case` expression and the condition in an `ispair` expression, rather than restricting these to variables as we did in Core. This leads to the next modification.

Binding `IsPair`

Type inference in Core is done on an unevaluated expression. Due to pattern compilation, all `ispair` conditions are variables and all `kar` and `kdr` operations within the scope of that `ispair` take this variable as an argument. This invariant is vital to the type system. For the type soundness proof we need this invariant to survive evaluation so that the partially evaluated program is also well typed. To achieve this we use a *binding* version of `ispair` so that this variable can't become something else due to substitution during evaluation. One possible solution is to use a slight variant of Core's `ispair`, `ispair` $x' = x$ `in` $f\ g$, such that

$$\texttt{ispair}\ x\ f\ g \equiv \texttt{ispair}\ x' = x\ \texttt{in}\ [x'/x]f\ [x'/x]g$$

However, on this path is an even simpler solution which automatically binds `kar(x)` to a variable and `kdr(x)` to another variable. I.e. the splitting of data is done as a part of the `ispair` expression.

$$\texttt{ispair}\ e\ \texttt{bind}\ (x, y)\ \texttt{in}\ f\ \texttt{else}\ g$$

In this version of `ispair`, the variable x within f is bound to $\texttt{kar}(e)$ and the variable y within f is bound to $\texttt{kdr}(e)$. This removes the need for `kar` and `kdr` expressions in the language at all, significantly simplifying it. We can recover the semantics of Core with the following equivalence between Core's `ispair` expressions and these binding `ispair` expressions.

$$\texttt{let}\ x = e\ \texttt{in}\ (\texttt{ispair}\ x\ \texttt{bind}\ (x', y')\ \texttt{in}\ f\ \texttt{else}\ g) \equiv \texttt{ispair}\ x\ [x'/\texttt{kar}(x), y'/\texttt{kdr}(x)]f\ g$$

We have shown that binding `ispair` is equivalent to Core's `ispair`, but we also need to justify why we use the two different styles. Binding `ispair` is much simpler, particularly in the type soundness proof that follows, thus we use it for proving properties of the type system. Core's `ispair` accurately reflects what the dgen compiler *actually does* and so we use it for all other parts of the thesis. Furthermore, the use of separate `kar` and `kdr` primitives makes clearer the fact that the spine view is just a *view* of data, not an *encoding* of data. It remains as future work to determine if the binding `ispair` is a better alternative for actual implementations. Its simplicity certainly recommends it, but delaying the operation of pulling apart data can also have advantages, depending on the language's evaluation order, implemented optimisations and usage patterns.

9.2 The Typing Relation

The typing relation, shown in Figure 9.3 makes use of the notion of *generalisation*. A type scheme σ is a generalisation of a type τ, denoted $\sigma \succ \tau$, if there is some substitution for the bound variables of σ which gives τ. We will require the following property of generalisation during the proof.

Lemma 9.2.1. *If* $\forall\alpha_1 \cdots \alpha_n.\tau \succ \rho$ *and* $FV(\tau) = \emptyset$ *then* $\tau = \rho$*.*

Furthermore, we can generalise a type τ to a type scheme with the *Gen* operation

$$Gen(\tau, A) = \forall\alpha_1 \ldots \alpha_n.\tau \text{ where } \{\alpha_1 \ldots \alpha_n\} = TV(\tau) \setminus TV(A)$$

Figure 9.2: Operational Semantics ($e \longrightarrow f$)

(E-App1)
$$\frac{e \longrightarrow e'}{e\ f \longrightarrow e'\ f}$$

(E-App2)
$$\frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'}$$

(E-Lam)
$$(\lambda x.e)\ v \longrightarrow [x/v]e$$

(E-Let1)
$$\frac{e \longrightarrow e'}{\texttt{let}\ x = e\ \texttt{in}\ f \longrightarrow \texttt{let}\ x = e'\ \texttt{in}\ f}$$

(E-Let2)
$$\texttt{let}\ x = v\ \texttt{in}\ f \longrightarrow [x/v]f$$

(E-LetRec1)
$$\frac{e \longrightarrow e'}{\texttt{letrec}\ x = e\ \texttt{in}\ f \longrightarrow \texttt{letrec}\ x = e'\ \texttt{in}\ f}$$

(E-LetRec2)
$$\frac{f \longrightarrow f'}{\texttt{letrec}\ x = v\ \texttt{in}\ f \longrightarrow \texttt{letrec}\ x = v\ \texttt{in}\ f'}$$

(E-LetRec3)
$$\frac{x \in FV(v_f)}{\texttt{letrec}\ x = v\ \texttt{in}\ v_f \longrightarrow \texttt{letrec}\ x = v\ \texttt{in}\ [x/v]v_f}$$

(E-LetRec4)
$$\frac{x \notin FV(v_f)}{\texttt{letrec}\ x = v\ \texttt{in}\ v_f \longrightarrow v_f}$$

(E-Case1)
$$\frac{e \longrightarrow e'}{(\texttt{case}\ e\ \texttt{of}\ (K(x_1,\ldots,x_n)) \to f) \longrightarrow \texttt{case}\ e'\ \texttt{of}\ (K(x_1,\ldots,x_n)) \to f}$$

(E-Extn1)
$$\frac{e \longrightarrow e'}{e \triangleright f \longrightarrow e' \triangleright f}$$

(E-Case2)
$$(\texttt{case}\ (K(v_1,\ldots,v_n))\ \texttt{of}\ (K(x_1,\ldots,x_n)) \to e) \longrightarrow [x_1/v_1]\cdots[x_n/v_n]e$$

(E-Extn2)
$$\frac{e \longrightarrow e'}{v \triangleright e \longrightarrow v \triangleright e'}$$

(E-Extn3)
$$\frac{A \vdash v\colon \tau \to \rho \qquad A \vdash v''\colon \tau \qquad TV(\tau) = \emptyset}{(v \triangleright v')\ v'' \longrightarrow v\ v''}$$

(E-Extn4)
$$\frac{A \vdash v\colon \tau \to \rho \qquad A \vdash v''\colon \tau' \qquad \tau \neq \tau'}{(v \triangleright v')\ v'' \longrightarrow v'\ v''}$$

(E-Con1)
$$\frac{e_j \longrightarrow e'_j}{K(v_1,\ldots,v_{j-1},e_j,e_{j+1},\ldots,e_m) \longrightarrow K(v_1,\ldots,v_{j-1},e'_j,e_{j+1},\ldots,e_m)}$$

(E-Con2)
$$K(v_1,\ldots,v_m)\ e \longrightarrow K(v_1,\ldots,v_m,e)$$

(E-IsPair1)
$$\frac{e \longrightarrow e'}{\texttt{ispair}\ e\ \texttt{bind}\ (x,y)\ \texttt{in}\ f\ \texttt{else}\ g \longrightarrow \texttt{ispair}\ e'\ \texttt{bind}\ (x,y)\ \texttt{in}\ f\ \texttt{else}\ g}$$

(E-IsPair2)
$$\texttt{ispair}\ (K(v_1,\ldots,v_m))\ \texttt{bind}\ (x,y)\ \texttt{in}\ e\ \texttt{else}\ f \longrightarrow [x/K(v_1,\ldots,v_{m-1}),y/v_m]e$$

(E-IsPair3)
$$\frac{v \neq K(v_1,\ldots,v_m)}{\texttt{ispair}\ v\ \texttt{bind}\ (x,y)\ \texttt{in}\ e\ \texttt{else}\ f \longrightarrow f}$$

Figure 9.3: Type Relation ($A \vdash e\colon \tau$)

(T-Var)
$$\frac{x\colon \sigma \in A \qquad \sigma \succ \tau}{A \vdash x\colon \tau}$$

(T-App)
$$\frac{A \vdash e\colon \rho \to \tau \qquad A \vdash f\colon \rho}{A \vdash e\, f\colon \tau}$$

(T-Let)
$$\frac{A \vdash e\colon \tau' \qquad A_x, x\colon Gen(\tau', A) \vdash f\colon \tau}{A \vdash (\texttt{let}\; x = e\; \texttt{in}\; f)\colon \tau}$$

(T-LetRec)
$$\frac{A_x, x\colon Gen(\tau', A) \vdash e\colon \tau' \qquad A_x, x\colon Gen(\tau', A) \vdash f\colon \tau}{A \vdash (\texttt{letrec}\; x = e\; \texttt{in}\; f)\colon \tau}$$

(T-Lam)
$$\frac{A_x, x\colon \rho \vdash e\colon \tau}{A \vdash (\lambda x.e)\colon \rho \to \tau}$$

(T-IsPair)
$$\frac{A \vdash e\colon \tau_e \qquad A_{x,y}, x\colon \alpha \to \tau_e, y\colon \alpha \vdash f\colon \tau \qquad A \vdash g\colon \tau \qquad \alpha \notin TV(A, \tau, \tau_e)}{A \vdash (\texttt{ispair}\; e\; \texttt{bind}\; (x, y)\; \texttt{in}\; f\; \texttt{else}\; g)\colon \tau}$$

(T-Ext)
$$\frac{A \vdash g\colon \tau_g \to \tau'_g \qquad A \vdash f\colon \tau_f \to \tau'_f \qquad Gen(\tau_f \to \tau'_f) \succ \tau_g \to \tau'_g \qquad TV(\tau_g, \tau'_g) = \emptyset}{A \vdash g \triangleright f\colon \tau_f \to \tau'_f}$$

for each K : $((\forall \alpha_1.\exists \beta_1.\tau'_1), \ldots, (\forall \alpha_n.\exists \beta_n.\tau'_n)) \to \tau_K$ where τ_K is unique to this K

(T-Con)
$$\frac{\forall i.(A \vdash e_i\colon [\beta_i/\rho_i]\tau'_i) \qquad \forall i(\alpha_i \notin TV(A))}{A \vdash (K(e_1, \ldots, e_n))\colon \tau_K}$$

(T-Con2)
$$\frac{\forall i \in 1 \ldots m.(A \vdash e_i\colon [\beta_i/\rho_i]\tau'_i) \qquad m < n \qquad \forall i(\alpha_i \notin TV(A))}{A \vdash (K(e_1, \ldots, e_m))\colon [\beta_{m+1}/\rho_{m+1}]\tau'_{m+1} \to (\cdots \to ([\beta_n/\rho_n]\tau'_n \to \tau_K) \cdots)}$$

(T-Case)
$$\frac{A_{x_1 \cdots x_n}, x_1\colon [\alpha_1/\rho_1]\tau'_1, \ldots, x_n\colon [\alpha_n/\rho_n]\tau'_n \vdash e\colon \tau_e \qquad \forall i.(\beta_i \notin TV(A, \tau_e, \rho_i)) \qquad A \vdash f\colon \tau_K}{A \vdash (\texttt{case}\; f\; \texttt{of}(K(x_1, \ldots, x_n)) \to e)\colon \tau_e}$$

9.3 Soundness Proof

We take the approach of Wright and Felleisen [92] and develop a proof based on the language's small-step operational semantics. Our proof hinges on two main results:

Progress If $A \vdash e\colon \tau$ then $e \longrightarrow e'$ or e is a value.

Preservation If $A \vdash e\colon \tau$ and $e \longrightarrow e'$ then $A \vdash e'\colon \tau$

If both progress and preservation are proven then the type system is *sound*.

Theorem 9.3.1 (Progress). *If $A \vdash e\colon \tau$ then $e \longrightarrow e'$ or e is a value.*

Proof. The proof proceeds by induction on the length of the type deduction for $A \vdash e\colon \tau$, with one case for each possible final deduction rule.

Case T-Var Immediate since x is a value.

Case T-App If the final deduction rule is T-App and $A \vdash e\ f : \tau$ then we have

$$A \vdash e : \rho \to \tau \tag{9.1}$$

$$A \vdash f : \rho \tag{9.2}$$

By the inductive hypothesis we know either $e \longrightarrow e'$ or e is a value. If e is not a value (i.e. $e \longrightarrow e'$) , by E-App we have $e\ f \longrightarrow e'\ f$. If e is a value (say v_e), then we need to consider the possible forms for f. From (9.2) and the induction hypothesis we have either f is a value or $f \longrightarrow f'$. If f is not a value (i.e. $f \longrightarrow f'$) then by E-App2 we have $v_e\ f \longrightarrow v_e\ f'$. If both e and f are values, say f is v_f, then we need to consider all the possible forms of v_e:

v_e **is** $\lambda x.g$ Then by E-Lam we have $(\lambda x.g)\ v_f \longrightarrow [x/v_f]g$.

v_e **is** $K(v_1, \ldots, v_n)$ Then by E-Con2 we have $K(v_1, \ldots, v_n)\ v_f \longrightarrow K(v_1, \ldots, v_n, v_f)$

v_e **is** $(v \triangleright v')$ Either the type of v_f is the type of the domain of v, in which case by E-Extn3 we have $(v \triangleright v') \longrightarrow v\ v_f$, or it is not, in which case by E-Ext4 we have $(v \triangleright v') \longrightarrow v'\ v_f$.

v_e **is a semi-value** Immediate since $v_e\ v_f$ is a value, by virtue of being a semi-value.

Case T-Let If the final deduction rule is T-Let and $A \vdash (\texttt{let}\ x = e\ \texttt{in}\ f) : \tau$ then we have $A \vdash e : \tau'$. By this and the inductive hypothesis we have that $e \longrightarrow e'$ or e is a value. If e is a value (say v), then by E-Let2 we have $\texttt{let}\ x = v\ \texttt{in}\ f \longrightarrow [x/v]f$. If e is not a value (i.e. $e \longrightarrow e'$), then by E-Let1 we have $\texttt{let}\ x = e\ \texttt{in}\ f \longrightarrow \texttt{let}\ x = e'\ \texttt{in}\ f$.

Case T-LetRec If the final deduction rule is T-LetRec and $A \vdash (\texttt{letrec}\ x = e\ \texttt{in}\ f) : \tau$ then we have $A_x, x : Gen(\tau', A) \vdash e : \tau'$ and $A_x, x : Gen(\tau', A) \vdash f : \tau'$. By these and the inductive hypothesis we have $e \longrightarrow e'$ or e is a value; plus $f \longrightarrow f'$ or f is a value. If e is a value (say v_e) we need to prove the proposition for:

$f \longrightarrow f'$ Then by E-LetRec2 we have $\texttt{letrec}\ x = e\ \texttt{in}\ f \longrightarrow \texttt{letrec}\ x = e\ \texttt{in}\ f'$.

f **is a value and** $x \in FV(f)$ By E-LetRec3 we have $\texttt{letrec}\ x = v_e\ \texttt{in}\ f \longrightarrow \texttt{letrec}\ x = v_e\ \texttt{in}\ [x/v]f$

f **is a value and** $x \notin FV(f)$ By E-LetRec4 we have $\texttt{letrec}\ x = v_e\ \texttt{in}\ f \longrightarrow f$.

If e is not a value (i.e. $e \longrightarrow e'$) then by E-LetRec1 we have $\texttt{letrec}\ x = e\ \texttt{in}\ f \longrightarrow \texttt{letrec}\ x = e'\ \texttt{in}\ f$

Case T-Lam Immediate since $\lambda x.e$ is a value.

Case T-IsPair If the final deduction rule is T-IsPair and $A \vdash (\texttt{ispair}\, e\, \texttt{bind}\, (x, y)\, \texttt{in}\, f\, \texttt{else}\, g) \colon \tau$ we have

$$A \vdash e \colon \tau_e \tag{9.3}$$

This with the induction hypothesis gives either e is a value or $e \longrightarrow e'$. Furthermore, if e is a value, it is either a constructed value ($K(v_1, \ldots, v_n)$) or it is not. These possibilities give rise to the following cases:

e is not a value By $e \longrightarrow e'$ and E-IsPair1 we have $\texttt{ispair}\, e\, \texttt{bind}\, (x, y)\, \texttt{in}\, f\, \texttt{else}\, g \longrightarrow \texttt{ispair}\, e'\, \texttt{bind}\, (x, y)\, \texttt{in}\, f\, \texttt{else}\, g$.

e is a constructed value Say e is the constructed value $K(e_1, \ldots, e_n)$. By E-IsPair2 we have $\texttt{ispair}\, K(v_1, \ldots, v_n)\, \texttt{bind}\, (x, y)\, \texttt{in}\, f\, \texttt{else}\, g \longrightarrow [x/K(v_1, \ldots, v_{n-1}), y/v_n]f$.

e is a value, but not a constructed value Say e is the non-constructed value v_e. By E-IsPair3 we have $\texttt{ispair}\, v_e\, \texttt{bind}\, (x, y)\, \texttt{in}\, f\, \texttt{else}\, g \longrightarrow g$.

Case T-Ext If the final deduction rule is T-Ext and $A \vdash e \triangleright f \colon \tau_f \to \tau_f'$ then we have

$$A \vdash e \colon \tau_e \to \tau_e' \tag{9.4}$$

$$A \vdash f \colon \tau_f \to \tau_f' \tag{9.5}$$

From (9.4) and the induction hypothesis we have that either e is a value or $e \longrightarrow e'$. If e is not a value (i.e. $e \longrightarrow e'$), then by E-Extn1 we have $e \triangleright f \longrightarrow e' \triangleright f$. For e is a value, say v_e, the inductive hypothesis and (9.5) give us that either f is a value or $f \longrightarrow f'$. If f is not a value (i.e. $f \longrightarrow f'$), then by E-Extn2 we have $v_e \triangleright f \longrightarrow v_e \triangleright f'$. If f is a value, say v_f, then $v_e \triangleright v_f$ is a value.

Case T-Con If $A \vdash K(e_1, \ldots, e_n) \colon \tau$, and all e_i are values, then $K(e_1, \ldots, e_n)$ is a value. If $A \vdash K(e_1, \ldots, e_n) \colon \tau$, and one of e_i is not evaluated (say e_j), then by T-Con, $A \vdash e_j \colon [\beta_j/\rho_j]\tau_j'$. Hence by the induction hypothesis $e_j \longrightarrow e_j$ and $K(e_1, \ldots, e_j, \ldots, e_n) \longrightarrow K(e_1, \ldots, e_j', \ldots, e_n)$

Case T-Con2 Similar to T-Con case above.

Case T-Case If the final deduction rule is T-Case and $A \vdash (\texttt{case}\, f\, \texttt{of}\, (K(x_1, \ldots, e_n)) \to$

$e): \tau$ then we have

$$A \vdash f\colon \tau_K \tag{9.6}$$

$$K\colon ((\forall \alpha_1.\exists \beta_1.\tau_1'), \ldots, (\forall \alpha_n.\exists \beta_n.\tau_n')) \to \tau_K \tag{9.7}$$

From (9.6) and the induction hypothesis we have that either f is a value or $f \longrightarrow f'$. If f is not a value (i.e. $f \longrightarrow f'$) then by E-Case1 we have $\texttt{case}\ f\ \texttt{of}\ (K(x_1, \ldots, x_n)) \to e \longrightarrow \texttt{case}\ f'\ \texttt{of}\ (K(x_1, \ldots, x_n)) \to e$. If f is a value then by (9.7) and Lemma 9.3.5 we have that f is $K(v_1', \ldots, v_n')$ for some $v_1', \ldots, v_n'$. By this and E-Case2 we have $(\texttt{case}\ K(v_1', \ldots, v_n')\ \texttt{of}\ (K(x_1, \ldots, x_n)) \to e) \longrightarrow [x_1/v_1'] \cdots [x_n/v_n']e$

□

Theorem 9.3.2 (Preservation)**.** *If $A \vdash e\colon \tau$ and $e \longrightarrow e'$ then $A \vdash e'\colon \tau$*

Proof. The proof proceeds by induction on the depth of the evaluation tree, with one case for each possible final reduction $e \longrightarrow e'$.

Case E-App1 If $e\ f\colon \tau$ and $e\ f \longrightarrow e'\ f$ then, by T-App we have

$$A \vdash e\colon \rho \to \tau \tag{9.8}$$

$$A \vdash f\colon \rho \tag{9.9}$$

By (9.8) and the inductive hypothesis we have $e'\colon \rho \to \tau$. Combining this with (9.9) and T-App gives $e'\ f\colon \tau$.

Case E-App2 This case is done in the same way as E-App1, we won't repeat the argument. This argument also works for E-Let1, E-LetRec1,E-LetRec2, E-Case1, E-Extn1, E-Extn2, E-IsPair1 and T-Con1.

Case E-Lam If $(\lambda x.e)\ v \longrightarrow [x/v]e$ and $A \vdash (\lambda x.e)\ v\colon \tau$, by T-App we have

$$A \vdash (\lambda x\ .e)\colon \rho \to \tau \tag{9.10}$$

$$A \vdash v\colon \rho \tag{9.11}$$

From (9.10) and T-Lam we have

$$A_x, x\colon \rho \vdash e\colon \tau \tag{9.12}$$

By (9.11), (9.12) and Lemma 9.3.1 we have $A \vdash [x/v]e\colon \tau$.

Case E-Let2 If $\texttt{let}\ x = v\ \texttt{in}\ f \longrightarrow [x/v]f$ and $A \vdash (\texttt{let}\ x = v\ \texttt{in}\ e) \colon \tau$, by T-Let we have

$$A \vdash v \colon \tau' \tag{9.13}$$

$$A_x, x \colon Gen(\tau', A) \vdash f \colon \tau \tag{9.14}$$

Since $Gen(\tau', A)$ is $\forall \alpha_1 \dots \alpha_n.\tau$ where $\{\alpha_1 \dots \alpha_n\} = TV(\tau) \setminus TV(A)$; by (9.13), (9.14) and Lemma 9.3.1 we have $[x/v]f \colon \tau$.

Case E-LetRec3 If $\texttt{letrec}\ x = v\ \texttt{in}\ f \longrightarrow \texttt{letrec}\ x = v\ \texttt{in}\ [x/v]f$ and $A \vdash \texttt{letrec}\ x = v\ \texttt{in}\ f \colon \tau$, by T-LetRec we have

$$A_x, x \colon Gen(\tau', A) \vdash v \colon \tau' \tag{9.15}$$

$$A_x, x \colon Gen(\tau', A) \vdash f \colon \tau \tag{9.16}$$

From (9.16) and Lemma 9.3.3 we have

$$A_x, x \colon Gen(\tau', A), y \colon Gen(\tau', A) \vdash [x/y]f \colon \tau \tag{9.17}$$

by Lemma 9.3.1 and (9.17) we have

$$A_x, x \colon Gen(\tau', A) \vdash [y/v][x/y]f \colon \tau \tag{9.18}$$

Since y is fresh it is not in the free variables of v and $[y/v][x/y] = [x/v]$, thus we have

$$A_x, x \colon Gen(\tau', A) \vdash [x/v]f \colon \tau \tag{9.19}$$

By (9.19), (9.15) and T-LetRec we have $A \vdash \texttt{letrec}\ x = v\ \texttt{in}\ [v/x]f \colon \tau$

Case E-LetRec4 If $\texttt{letrec}\ x = v\ \texttt{of}\ f \longrightarrow f$ and $A \vdash \texttt{letrec}\ x = v\ \texttt{of}\ f \colon \tau$ we have

$$A_x, x \colon Gen(\tau', A) \vdash f \colon \tau \tag{9.20}$$

$$x \notin FV(f) \tag{9.21}$$

By these and Lemma 9.3.2 we have $A \vdash f \colon \tau$.

Case E-Case2 From $A \vdash (\texttt{case}\ (K(v_1, \dots, v_n))\ \texttt{of}\ (K(x_1, \dots, x_n) \to e) \colon \tau_e$ and T-Case we

have

$$A_{x_1,\ldots,x_n}, x_1\colon [\alpha_1/\rho_1]\tau_1', \ldots, x_n\colon [\alpha_n/\rho_n]\tau_n' \vdash e\colon \tau_e \tag{9.22}$$

$$A \vdash K(v_1, \ldots, v_n)\colon \tau_K \tag{9.23}$$

$$\forall i(\beta \notin TV(A, \tau_e, \rho_i)) \tag{9.24}$$

By (9.23) and T-Con we have

$$\forall i.(A \vdash e_i\colon [\beta_i/\rho_i']\tau_i') \tag{9.25}$$

$$\forall i.(\alpha_i \notin TV(A)) \tag{9.26}$$

Since $\forall i(\beta_i \notin TV(A, \tau_e, \rho_i))$, we can generalise (9.22) to

$$A_{x_1,\ldots,x_n}, x_1\colon [\alpha_1/\rho_1][\beta_1/\rho_1']\tau_1', \ldots, x_n\colon [\alpha_n/\rho_n][\beta_n/\rho_n']\tau_n' \vdash e\colon \tau_e \tag{9.27}$$

It must be the case, since all α are variables bound by an inner universal quantified in a constructor type, that no α is free in any τ_K. By this, Lemma 9.3.6, (9.25) and (9.26) we have

$$\forall i.(A \vdash e_i\colon [\alpha_i/\rho_i][\beta_i/\rho_i']\tau_i') \tag{9.28}$$

By (9.27), (9.28) and a multi-substitution generalisation of Lemma 9.3.1 have that $A \vdash [x_1/v_1]\cdots[x_n/v_n]e\colon \tau_e$ as required.

Case E-IsPair2 If $\texttt{ispair}\ (K(v_1, \ldots, v_m))\ \texttt{bind}\ (x, y)\ \texttt{in}\ f\ \texttt{else}\ g \longrightarrow [x/K(v_1, \ldots, v_{m-1}), y/v_m]f$ and $A \vdash \texttt{ispair}\ (K(v_1, \ldots, v_m))\ \texttt{bind}\ (x, y)\ \texttt{in}\ f\ \texttt{else}\ g\colon \tau$, by T-IsPair we have

$$A \vdash K(v_1, \ldots, v_m)\colon \tau' \tag{9.29}$$

$$A_{x,y}, x\colon \alpha \to \tau', y\colon \alpha \vdash v'\colon \tau \tag{9.30}$$

where α is unique. From (9.29) and T-Con2 we have

$$\forall i \in 1\ldots m.(A \vdash v_i\colon [\beta_i/\rho_i]\tau_i') \tag{9.31}$$

By (9.31) and T-Con2 again (this time in the opposite direction) we have

$$A \vdash K(v1, \ldots, v_{m-1})\colon [\beta_m/\rho_m]\tau_m' \to \tau' \tag{9.32}$$

Note also that (9.31) includes the fact that $A \vdash v_m\colon [\beta_m/\rho_m]\tau_m'$. This with (9.32),

(9.30) and Lemma 9.3.4 gives $A \vdash [x/K(v_1, \ldots, v_{m-1}), y/v_m)]f \colon \tau$ as required.

Case E-IsPair3 If $\texttt{ispair}\ v\ \texttt{bind}\ (x, y)\ \texttt{in}\ f\ \texttt{else}\ g \longrightarrow g$ and $A \vdash \texttt{ispair}\ v\ \texttt{bind}\ (x, y)\ \texttt{in}\ f\ \texttt{else}\ g \colon \tau$, by T-IsPair we have $A \vdash g \colon \tau$ as required.

Case E-Extn3 If $A \vdash (v \triangleright v')\ v'' \colon \tau$, by T-App we have

$$A \vdash (v \triangleright v') \colon \rho \to \tau \tag{9.33}$$

$$A \vdash v'' \colon \rho \tag{9.34}$$

By (9.34) and E-Ext3 we have

$$TV(\rho) = \emptyset \tag{9.35}$$

By (9.33) and T-Ext we have

$$A \vdash v \colon \tau_\alpha \to \tau'_\alpha \tag{9.36}$$

$$Gen(\rho \to \tau) \succ \tau_\alpha \to \tau'_\alpha \tag{9.37}$$

$$TV(\tau_\alpha) = \emptyset \tag{9.38}$$

ρ has no type variables, hence τ has no type variables, hence $Gen(\rho \to \tau) = \rho \to \tau$ and

$$\rho \to \tau \succ \tau_\alpha \to \tau'_\alpha \tag{9.39}$$

By (9.39) we have

$$\rho \succ \tau_\alpha \tag{9.40}$$

$$\tau \succ \tau'_\alpha \tag{9.41}$$

By (9.41) and Lemma 9.2.1 we have $\tau = \tau'_\alpha$. By (9.40) and Lemma 9.2.1 we know $\tau' = \tau_\alpha$. These equalities, with (9.34), (9.36) and T-App, give $A \vdash v\ v'' \colon \tau$.

Case E-Extn4 If $(v \triangleright v')\ v'' \longrightarrow v'\ v''$ and $A \vdash (v \triangleright v')\ v'' \colon \tau$, by T-App we have

$$A \vdash (v \triangleright v') \colon \rho \to \tau \tag{9.42}$$

$$A \vdash v'' \colon \rho \tag{9.43}$$

From $A \vdash (v \triangleright v')\colon \rho \to \tau$ and T-Ext we have

$$A \vdash v'\colon \rho \to \tau \tag{9.44}$$

By (9.43), (9.44) and T-App we have $A \vdash v'\ v''\colon \tau$ as required.

Case T-Con2 By $A \vdash K(v_1, \ldots, v_m)\ e\colon \tau$ and T-App we have

$$A \vdash K(v_1, \ldots, v_m)\colon \rho \to \tau \tag{9.45}$$

$$A \vdash e\colon \rho \tag{9.46}$$

By (9.45) and T-Con2 we have $A \vdash K(v_1, \ldots, v_m)\colon [\beta_{m+1}/\rho'_{m+1}]\tau'_{m+1} \to ([\beta_{m+2}/\rho'_{m+2}]\tau'_{m+2} \to \cdots \to \tau_K)$ Hence $\tau = ([\beta_{m+2}/\rho'_{m+2}]\tau'_{m+2} \to \cdots \to \tau_K)$ and $\rho = [\beta_{m+1}/\rho'_{m+1}]\tau'_{m+1}$. These equalities with (9.6), (9.7) and T-Cons2 give $A \vdash K(v1, \ldots, v_m, e)\colon \tau$ as required.

□

Lemma 9.3.1 (Substitution). *If $A_x, x\colon \forall\alpha_1 \ldots \alpha_n.\tau \vdash e\colon \tau'$, $x \notin Dom(A)$, $A \vdash v\colon \tau$ and $\{\alpha_1 \ldots \alpha_n\} \cap TV(A) = \emptyset$ then $A \vdash [x/v]e\colon \tau'$*

Proof. The proof is by induction on the length of the type deduction for $A_x, x\colon \forall\alpha_1 \ldots \alpha_n.\tau \vdash e\colon \tau'$, with one case for each possible final deduction rule. □

Lemma 9.3.2 (Unused Variable). *If $A_x, x\colon \sigma \vdash e\colon \tau$ and $x \notin FV(e)$ then $A \vdash e\colon \tau$*

Proof. The proof is by induction on the length of the type deduction for $A_x, x\colon \sigma \vdash e\colon \tau$, with one case for each possible final deduction rule. □

Lemma 9.3.3 (Weakening). *If $A \vdash e\colon \tau$ and $x\colon \sigma \in A$ and $y \notin FV(e, A)$ then $A, y\colon \sigma \vdash [x/y]e\colon \tau$*

Proof. The proof is by induction on the length of the type deduction for $A \vdash e\colon \tau$, with one case for each possible final deduction rule. □

Lemma 9.3.4 (Existential Instantiation). *If $A_{x,y}, x\colon \alpha \to \tau', y\colon \alpha \vdash e\colon \tau$ and $\alpha \notin TV(A, \tau, \tau', \rho')$ and $A \vdash v'\colon \rho' \to \tau'$ and $A \vdash v''\colon \rho'$ then $A \vdash [x/v', y/v'']e\colon \tau$ for* any ρ'.

Proof. The proof is by induction on the length of the type deduction for $A_{x,y}, x\colon \alpha \to \tau', y\colon \alpha \vdash v\colon \tau$, with one case for each possible final deduction rule. □

Lemma 9.3.5 (Constructor Types). *If* $A \vdash v\colon \tau_K$ *and* $K\colon ((\forall\alpha_1.\exists\beta_1.\tau'_1) \to \cdots \to (\forall\alpha_n.\exists\beta_n.\tau'_n)) \to \tau_K$ *then* v *is* $K(v_1,\ldots,v_n)$ *for some* $v_1,\ldots,v_n$

Proof. The proof relies on the fact that τ_K is unique for each K. □

Lemma 9.3.6 (Generalisation). *If* $A \vdash e\colon \tau$, $\forall K.(\alpha \notin TV(\tau_K))$ *and* $\alpha \notin TV(A)$ *then* $A \vdash e\colon [\alpha/\rho]\tau$

Proof. The proof proceeds by induction on the length of the type deduction for $A \vdash e\colon \tau$, with one case for each possible final deduction rule.

Case T-Var By $A \vdash x\colon \tau$ and T-Var we have

$$x\colon \sigma \in A \tag{9.47}$$

$$\sigma \succ \tau \tag{9.48}$$

Since $\sigma \in A$ and $\alpha \notin A$, $\alpha \notin TV(\sigma)$. However α could be in the free type variables of τ. If $\alpha \in TV(\tau)$, since $\sigma \succ \tau$, it must be possible to substitute a bound type variable in σ for α to get τ. Hence, we can substitute that bound variable with ρ to get $[\alpha/\rho]\tau$. Hence

$$\sigma \succ [\alpha/\rho]\tau \tag{9.49}$$

By (9.47) and (9.49) and T-Var we get $A \vdash x\colon [\alpha/\rho]\tau$.

If $\alpha \notin TV(\tau)$ then $[\alpha/\rho]\tau = \tau$ and $\sigma \succ [\alpha/\rho]\tau$. By this and (9.47) we have $A \vdash x\colon [\alpha/\rho]\tau$.

Case T-App By $A \vdash e\, f\colon \tau$ and T-App we have

$$A \vdash e\colon \tau' \to \tau \tag{9.50}$$

$$A \vdash f\colon \tau' \tag{9.51}$$

By (9.50), and the induction hypothesis we have

$$A \vdash e\colon [\alpha/\rho]\tau' \to [\alpha/\rho]\tau \tag{9.52}$$

By (9.51), and the induction hypothesis we have

$$A \vdash f\colon [\alpha/\rho]\tau' \tag{9.53}$$

By (9.52), (9.53) and T-App we have $A \vdash e\, f\colon [\alpha/\rho]\tau$.

Case T-Let, T-LetRec and T-Case The proof is similar to the T-App case above.

Case T-Lam If $A \vdash (\lambda x.e)\colon \tau' \to \tau$ then by T-Lam we have $A_x, x\colon \tau' \vdash e\colon \tau$. If $\alpha \notin TV(\tau')$ then $\alpha \notin A_x, x\colon \tau'$ and by the inductive hypothesis $A_x, x\colon \tau' \vdash [\alpha/\rho]\tau$. Furthermore, since $\alpha \notin TV(\tau')$, $[\alpha/\rho]\tau' = \tau'$ and $A_x, x\colon [\alpha/\rho]\tau' \vdash [\alpha/\rho]\tau$. This gives $A \vdash (\lambda x.e)\colon [\alpha/\rho](\tau' \to \tau)$ as required. If $\alpha \in TV(\tau')$, then by Lemma 9.3.7 we have $A_x, x\colon [\alpha/\rho]\tau' \vdash [\alpha/\rho]\tau$. This gives $A \vdash (\lambda x.e)\colon [\alpha/\rho](\tau' \to \tau)$ as required.

Case T-IsPair If $A \vdash (\texttt{ispair}\ e\ \texttt{bind}\ (x, y)\ \texttt{in}\ f\ \texttt{else}\ g)\colon \tau$, T-IsPair gives

$$A \vdash e\colon \tau_e \tag{9.54}$$

$$A_x, x\colon \beta \to \tau_e, y\colon \beta \vdash f\colon \tau \tag{9.55}$$

$$A_x \vdash g\colon \tau \tag{9.56}$$

By the same argument we used in the T-Lam case, from $A_x, x\colon \beta \to \tau_e, y\colon \beta \vdash f\colon \tau$ and the fact that α cannot be in $TV(\beta)$ we have

$$A_x, x\colon \beta \to [\alpha/\rho]\tau_e, y\colon \beta \vdash f\colon [\alpha/\rho]\tau \tag{9.57}$$

By (9.56) and the induction hypothesis we have

$$A \vdash g\colon [\alpha/\rho]\tau \tag{9.58}$$

By (9.54), (9.57), (9.58) and T-IsPair we have $A \vdash (\texttt{ispair}\ e\ \texttt{bind} = (x, y)\ \texttt{in}\ f\ \texttt{else}\ g)\colon [\alpha/\rho]\tau$ as required.

Case T-Ext Our initial assumptions are

$$A \vdash g \triangleright f\colon \tau_f \to \tau_f' \tag{9.59}$$

$$\alpha \notin TV(A) \tag{9.60}$$

$$\forall K.(\alpha \notin TV(\tau_K)) \tag{9.61}$$

By (9.59) and T-Ext we have

$$A \vdash g \colon \tau_g \to \tau'_g \tag{9.62}$$

$$A \vdash f \colon \tau_f \to \tau'_f \tag{9.63}$$

$$Gen(\tau_f \to \tau'_f) \succ \tau_g \to \tau'_g \tag{9.64}$$

$$TV(\tau_g) = \emptyset \tag{9.65}$$

By (9.63) and the induction hypothesis we have

$$A \vdash f \colon [\alpha/\rho](\tau_f \to \tau'_f) \tag{9.66}$$

which gives

$$A \vdash f \colon ([\alpha/\rho]\tau_f \to [\alpha/\rho]\tau'_f) \tag{9.67}$$

By (9.62) and the induction hypothesis we have

$$A \vdash g \colon [\alpha/\rho](\tau_g \to \tau'_g) \tag{9.68}$$

which gives

$$A \vdash g \colon ([\alpha/\rho]\tau_g \to [\alpha/\rho]\tau'_g) \tag{9.69}$$

(9.65) immediately gives

$$TV([\alpha/\rho]\tau_g) = \emptyset \tag{9.70}$$

Lemma 9.3.8 tells us that if $\alpha \in TV(\tau_f, \tau'_f)$ there is also a derivation which gives different τ_f, τ'_f which *don't* have $\alpha \in TV(\tau_f, \tau'_f)$. Hence we continue under the assumption that $\alpha \notin TV(\tau_f, \tau'_f)$.

From $Gen(\tau_f \to \tau'_f) \succ \tau_g \to \tau'_g$, $TV(\tau_g, \tau'_g) = \emptyset$ and $\alpha \notin TV(\tau_f, \tau'_f)$, we have $Gen([\alpha/\rho]\tau_f \to [\alpha/\rho]\tau'_f) \succ [\alpha/\rho]\tau_g \to [\alpha/\rho]\tau'_g$. Finally, by this, (9.67), (9.69), (9.70) and T-Ext we have

$$A \vdash f \triangleright g \colon [\alpha/\rho]\tau_f \to [\alpha/\rho]\tau'_f \tag{9.71}$$

and hence

$$A \vdash f \triangleright g \colon [\alpha/\rho](\tau_f \to \tau'_f) \tag{9.72}$$

as required.

Case T-Con By the assumptions of this case we have that $A \vdash (K(e_1, \ldots, e_n)) \colon \tau_K$ and that $\alpha \notin TV(\tau_K)$. Since $\alpha \notin TV(\tau_K)$ then $[\alpha/\rho]\tau_K = \tau_K$ and $A \vdash (K(e_1, \ldots, e_n)) \colon [\alpha/\rho]\tau_K$

as required.

Case T-Con2 Similar to the T-Con case.

□

Lemma 9.3.7 (Double Substitution)**.** *If* $A_x, x\colon \tau' \vdash e\colon \tau$*,* $\alpha \notin TV(A)$ *and* $\alpha \in TV(\tau')$ *then* $A_x, x\colon [\alpha/\rho]\tau' \vdash e\colon [\alpha/\rho]\tau$*.*

Proof. Proof is by induction on the length of the type deduction of $A_x, x\colon \tau' \vdash e\colon \tau$. □

Lemma 9.3.8 (Equivalence of Free Type Variables)**.** *If* $A \vdash e\colon \tau$*,* $\alpha \notin TV(A)$ *and* $\beta \notin TV(A, \tau)$ *then* $A \vdash e\colon [\alpha/\beta]\tau$*.*

Proof. Proof uses Lemma 9.3.6 and Lemma 9.3.7. □

9.4 Summary

We have proven the soundness of the typing relation which underlies the type inference done by Core. We have done this by proving progress and preservation theorems for a representative subset of Core using the techniques of Wright and Felleisen [92]. Included with this proof is a small-step operational semantics for this subset of Core which further illuminates, via a binding `ispair`, the operation of `kar`, `kdr`, `ispair`; and which also clarifies the $\triangleright$ operation.

Chapter 10

Conclusion

10.1 A Type-Safe Function Extension Mechanism

Of the available techniques for creating polymorphic functions with specific behaviour, we discovered that none were entirely suitable for compilation in a baseline functional compiler. We created our own using features from Scrap Your Boilerplate (and its precursors) and the pattern calculus. The end result is a single function extension operator which is able to create both type preserving transformations and polymorphic accumulators. To accompany this new source language facility, we have defined new type inference rules which are able to capture, in one simple rule, *both* possible behaviours of function extension. Furthermore we showed that function extension can be converted to a very simple run-time operation which requires only type information to be added to the run-time, not pervasive type inspection. The end result is a mechanism for polymorphic functions with specific behaviour that is simple to incorporate in a baseline functional compiler and only checks run-time type information when absolutely necessary.

10.2 A Compilation Scheme and a Type Inference Algorithm for Application Pattern Matches

We have created the first compiled implementation of the explicit spine view approach to structure agnosticism. This particular universal view of data has been used implicitly or interpreted in other systems but ours is the first compiled instance of it. We contribute to the literature on the spine view by giving a new account of static type inference for it. Our solution is more explicit than those used for implicit spine views and is simpler than that used in bondi and the pattern calculus. Again we show how we can compile

this mechanism to very simple run-time operations which are easy to add to a baseline functional compiler. We also gave an account of dealing with so-called lonely constructors in the run-time. Again, this solution is simple to add to a baseline compiler and it is powerful enough to admit generic show and generic encode. In particular, it allowed DGEN to compile these two snippets without making partially applied constructors first-class citizens of the source language.

10.3 Demonstration of Techniques

This unique combination of techniques was investigated for its expressiveness and efficiency of compiled code. We demonstrated that we can implement our eight example snippets and a great deal more. We showed that the run-time performance of generic programs compiled with DGEN is excellent in comparison to the performance of non-generic programs compiled with DGEN, demonstrating that the techniques we describe cause relatively little slowdown or memory overhead. Finally, we have proven the soundness of the typing relation by which all these new features are realised in a statically typed functional programming language.

10.4 Future Work

We plan to use the compiler we have created for further investigations, particularly we want to study the following areas.

10.4.1 More Expressive Spine Views

Hinze et. al. [38, 37] have laid out two possible extensions to the basic spine view that we currently use in DGEN. Implementing and experimenting with these could expand the set of admissible functions enough to include all canonical datatype generic functions. For example, in Chapter 7 we noted that more expressive spine views could allow generic read and generic map. The simple spine view we are using is able to be compiled into very simple underlying mechanisms and discovering whether or not this is still possible with a more expressive spine view would be very interesting. If it is not possible, we would like to extract the minimal cost to accompany the improved expressiveness.

10.4.2 Object Oriented Application Pattern Matching

Scala [72] is a language with very expressive pattern matching where pattern compilation can be supplemented by library authors [26, 25]. We want to investigate whether the pattern compilation techniques we used for application pattern matches can be translated to that system. F# [23] has a very similar feature called *active patterns* [83]. They are used not just to perform pattern matching on Objects, but also on any abstract data type. We would also like to investigate building application pattern matches with active patterns in F#.

10.4.3 Integration Into Other Compilers

We have made a compelling case that our techniques (function extension and application pattern matches) could be used in most functional programming languages. To *demonstrate* this one needs to actually extend an existing compiler. OCaml [34] would be the most interesting target because Haskell already has many generic programming tools. However, this would be a significant project because OCaml does not have either polymorphic recursion or rank-2 types, although there is work that could inform these extensions [27]. The mechanisms we described for rank-2 arguments and polymorphic recursion rely on just a single mechanism not in standard Hindley-Milner, the set of fixed-for-unification type variables. Thus there is a good chance our techniques could be adapted to work in OCaml.

An implementation in a Haskell compiler would also be enlightening. It would then be possible to implement our solution and the generics libraries like Regular and Instant Generics in the same system, allowing us to make more accurate comparisons between our approach and those that use type classes to achieve minimal slowdown for generic code. We could also extract data illuminating to what extent the speed of these libraries is a result of GHC's sophisticated implementation and how much is inherent to using type classes.

10.4.4 Extending Support for the Pattern Calculus

Throughout this thesis we have referred to the pattern calculus as a source of results about the explicit spine view and for its use of extension typing (from which our function extension drew inspiration); and to bondi (an implementation of the pattern calculus) as interpreted implementation of techniques related to ours. However, there are still many ideas in the pattern calculus and in bondi that we have not implemented. Specifically we

would like to answer the following questions:

- How well can we enable extension typing with a single function extension operator?
- Can we encode the pattern calculus account of objects with our techniques?
- Can we support free variables in patterns as we did application pattern matches?
- Can we extract and use linearity constraints (required for free variables in patterns) from algebraic data type definitions?

10.4.5 Function Extension with Fewer Restrictions

We have enforced that the function extension operator only extend with non-arrow types which have no free type variables in them. This helped us avoid a number of tricky situations. However, it may be that there is room for useful behaviour in the things we are restricting. For example, why not allow type unification to happen at run-time? It would introduce some run-time cost, but how much? What new programs become expressible? We have not entirely convinced ourself that there is no reasonable behaviour for functions taking arrow types either. There is no extensional equality for these values, but we are only concerned with type safety, so perhaps type equivalence is sufficient.

10.4.6 Failure as Trivial Success

We have shown an encoding of term rewriting using an option type (`some`) to encode failure. Without this extra layer, the only possible generic function for a generic update (as the term-rewriting traversals are) is the trivial success function, identity. We would like to investigate whether the failure mechanism used in term rewriting can be emulated with function extension and identity (or vice versa).

10.4.7 Joining the Rewriting Game

Since DGEN is capable of term-rewriting we would like to include it in the "rewrite engines competition"[24]. This would give an even clearer measure of our success in terms of expressiveness and speed of compiled programs.

Bibliography

[1] Alimarine, A., and Plasmeijer, R. A generic programming extension for clean. In *Implementation of Functional Languages*, T. Arts and M. Mohnen, Eds., vol. 2312 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2002, pp. 15–30.

[2] Appel, A. W., and MacQueen, D. B. A standard ml compiler. In *Proc. of a conference on Functional programming languages and computer architecture* (London, UK, 1987), Springer-Verlag, pp. 301–324.

[3] Appel, A. W., and MacQueen, D. B. Standard ml of new jersey. In *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming* (1991), J. Maluszyński and M. Wirsing, Eds., no. 528, Springer Verlag, pp. 1–13.

[4] Arbiser, A., Miquel, A., and Rìos, A. A lambda-calculus with constructors. *Term Rewriting and Applications* (2006).

[5] Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E., and Reilles, A. Tom: Piggybacking rewriting on java. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications* (2007), Lecture Notes in Computer Science, Springer-Verlag.

[6] Barret, G., and Wadler, P. Derivation of a pattern-matching compiler. Tech. rep., Programming Research Group, Oxford, 1986.

[7] Borovansky, P., Kirchner, C., Kirchner, H., Moreau, P., and Ringeissen, C. An overview of elan. In *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications.* (Pone-A-Mausson, France, September 1998), C. Kirchner and H. Kirchner, Eds.

[8] Brady, E. Lightweight invariants with full dependent types. In *Trends in Functional Programming* (2008), M. Morazán, P. Koopman, and P. Achten, Eds., vol. 9, Intellect.

[9] Brian. Scrap your boilerplate in f#. http://stackoverflow.com/questions/3596718/scrap-your-boilerplate-in-f, 2010.

[10] Bruno C. d. S. Oliveira, R. H., and Loeh, A. Extensible and modular generics for the masses. In *Trends in Functional Programming*, H. Nilsson, Ed. 2007.

[11] Brus, T. H., van Eekelen, C. J. D., van Leer, M. O., and Plasmeijer, M. J. Clean: A language for functional graph rewriting. In *Proc. of a conference on Functional programming languages and computer architecture* (London, UK, 1987), Springer-Verlag, pp. 364–384.

[12] Burstall, R., MacQueen, D., and Sannella, D. Hope: An experimental applicative language. In *Proceedings of the 1980 ACM conference on LISP and functional programming* (1980), ACM New York, NY, USA, pp. 136–143.

[13] Chakravarty, M., Ditu, G., and Leshchinskiy, R. Instant generics: Fast and easy.

[14] Cheney, J., and Hinze, R. A lightweight implementation of generics and dynamics. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell* (New York, NY, USA, 2002), ACM, pp. 90–104.

[15] Cirstea, H., Kirchner, C., and Liquori, L. Matching Power. In *Proceedings of RTA'2001* (May 2001), Lecture Notes in Computer Science, Springer-Verlag.

[16] Cirstea, H., Kirchner, C., and Liquori, L. Rewriting calculus with(out) types. In *Proceedings of the fourth workshop on rewriting logic and applications* (Pisa (Italy), Sept. 2002), F. Gadducci and U. Montanari, Eds., Electronic Notes in Theoretical Computer Science.

[17] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. The maude 2.0 system. In *Rewriting Techniques and Applications (RTA 2003)* (June 2003), R. Nieuwenhuis, Ed., no. 2706 in Lecture Notes in Computer Science, Springer-Verlag, pp. 76–87.

[18] Clément, D., Despeyroux, T., Kahn, G., and Despeyroux, J. A simple applicative language: mini-ml. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming* (New York, NY, USA, 1986), ACM, pp. 13–27.

[19] Damas, L., and Milner, R. Principal type-schemes for functional programs. *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1982), 207–212.

[20] Dijkstra, A., Fokker, J., and Swierstra, S. The architecture of the Utrecht Haskell compiler. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell* (2009), ACM, pp. 93–104.

[21] Dolstra, E. Firstclass rules and generic traversals for program transformation languages. Master's thesis, Institute of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, August 2001.

[22] Dolstra, E., and Visser, E. First-class rules and generic traversal. Tech. rep., Institute of Information and Computing Sciences, Utrecht University, November 2001.

[23] Don Syme, Adam Granicz, A. C. *Expert F#*. Apress, 2007.

[24] Durán, F., Roldán, M., Balland, E., van den Brand, M., Eker, S., Kallenberg, K., Kars, L., Moreau, P., Schevchenko, R., and Visser, E. The second rewriting engines competition. In *Proceedings of the 7th International Workshop on Rewrting Logic and its Applications (WRLA 2008)* (2009), G. Rosu, Ed., vol. ENTCS 238(3), pp. 281–291.

[25] Emir, B. *Object-Oriented Pattern Matching*. PhD thesis, 2007.

[26] Emir, B., Odersky, M., and Williams, J. Matching objects with patterns. *Lecture Notes in Computer Science 4609* (2007), 273.

[27] Emms, M., and Leiß, H. Extending the type checker of standard ml by polymorphic recursion. *Theor. Comput. Sci. 212* (February 1999), 157–181.

[28] Faure, G. Encoding rewriting strategies in λ-calculi with patterns. Tech. Rep. 7025, INRIA, 2009.

[29] Given-Wilson, T. Interpreting the untyped pattern calculus in bondi. Master's thesis, University of Technology, Sydney, 2007.

[30] Hallet, J. J., and Kfoury, A. J. Programming examples needing polymorphic recursion.

[31] Hartel, P. H., and Langendoen, K. Benchmarking implementations of lazy functional languages. In *Functional Programming Languages and Computer Architecture* (1993), pp. 341–349.

[32] Henderson, F., Conway, T., Somogyi, Z., Jeffery, D., Schachte, P., Taylor, S., Speirs, C., Dowd, T., and andMark Brown, R. B. The mercury language reference manual, Feb 2010.

[33] Henglein, F. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst. 15*, 2 (1993), 253–289.

[34] Hickey, J. *Introduction to the Objective Caml Porgramming Language*. 2006.

[35] Hinze, R. A generic programming extension for haskell. In *Utrecht University* (1999), pp. 1999–28.

[36] Hinze, R., Jeuring, J., and Löh, A. Comparing approaches to generic programming in haskell. In *Datatype-Generic Programming* (2007), vol. LNCS 4719/2007.

[37] Hinze, R., and Löh, A. "scrap your boilerplate" revolutions. In *Mathematics of Program Construction*, T. Uustalu, Ed., vol. 4014 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 180–208.

[38] Hinze, R., Löh, A., and Oliveira, B. C. D. S. Scrap your boilerplate" reloaded. Springer, pp. 13–29.

[39] Hinze, R., Löh, A., and Oliveira, B. C. D. S. Scrap your boilerplate" reloaded. Tech. rep., 2006. retrived from `http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.100.6601&rep=rep1&type=pdf`.

[40] Hoffmann, C., and ODonnell, M. Implementation of an interpreter for abstract equations. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1984), ACM New York, NY, USA, pp. 111–121.

[41] Jay, B. Typing first-class patterns. In *Proceedings of The Third International Workshop on Higher-Order Rewriting* (2006).

[42] Jay, B. Pattern calculus computing with functions and structures part i and most of part ii (under construction). personal correspondance, January 2007.

[43] Jay, B. *Computing with Functions and Structures*. Spinger-Verlag, 2009.

[44] Jay, C. bondi. `http://bondi.it.uts.edu.au/`, Dec 2010.

[45] Jay, C. B. The pattern calculus. *ACM Transactions of Programming Languages and Systems (TOPLAS) 26*, 6 (November 2004), 911–937.

[46] Jay, C. B., and Kesner, D. First-class patterns. *Journal of Functional Programming 19*, 02 (2009), 191–225.

[47] Johnsson, T. Lambda lifting: Transforming programs to recursive equations.

[48] Jones, M. First-class polymorphism with type inference. *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1997), 483–496.

[49] Jones, S., Ed. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[50] Jones, S., Vytiniotis, D., Weirich, S., and Shields, M. Practical type inference for arbitrary-rank types. *Journal of Functional Programming 17*, 01 (2006), 1–82.

[51] Jones, S. L. P. Compiling haskell by program transformation: A report from the trenches. In *European Symposium on Programming* (1996), pp. 18–44.

[52] Jones, S. L. P., Hall, C. V., Hammond, K., Partain, W., and Wadler, P. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference* (93).

[53] Jones, S. P. Re: Polymorphic recursion. `http://www.mail-archive.com/haskell@haskell.org/msg00492.html`.

[54] Jones, S. P. *Implementing Functional Languages*. Prentice Hall, 1987.

[55] Jones, S. P., and Lester, D. *Implementing Functional Languages: A Tutorial*. Prentice Hall, 1992.

[56] Jones, S. P., and Santos, A. L. M. Compilation by transformation in the glasgow haskell compiler. In *Functional Programming, Workshops in Computing* (1994), K. Hammond, D. N. Turner, and P. M. Sansom, Eds., Springer Verlag, pp. 184–204.

[57] Kelsey, R., and Hudak, P. Realistic compilation by program transformation. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages* (Austin, Texas, 1989), pp. 281–292.

[58] Lämmel, R. Typed generic traversal with term rewriting strategies. *Journal of Logic and Algebraic Programming 54* (2003). Also available as arXiv technical report cs.PL/0205018.

[59] Lämmel, R., and Jones, S. P. Scrap more boilerplate: reflection, zips, and generalised casts. *SIGPLAN Not. 39*, 9 (2004), 244–255.

[60] Lämmel, R., and Peyton Jones, S. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices 38*, 3 (Mar. 2003), 26–37. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[61] Lämmel, R., and Peyton Jones, S. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)* (Sept. 2005), ACM Press, pp. 204–215.

[62] Lämmel, R., Thompson, S., and Kaiser, M. Programming errors in traversal programs over structured data. *ENTCS* (2008). Short version appeared in LDTA 2008 ENTCS Proceedings. Full version submitted to SCP Special Issue.

[63] Le Fessant, F., and Maranget, L. Optimizing pattern matching. *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming* (2001), 26–37.

[64] Lee, O., and Yi, K. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst. 20* (July 1998), 707–723.

[65] Leijen, D. Hmf: simple type inference for first-class polymorphism. *SIGPLAN Not. 43*, 9 (2008), 283–294.

[66] Leroy, X., Doligeq, D., Garrigue, J., Rémy, D., and Jérôuillon. The objective caml system, release 3.11, documentation and user's manual. Tech. rep., Intitut National de Recherche en Informatique et en Automatique, 2008.

[67] Lippmeier, B. *Type Inference and Optimisation for an Impure World*. PhD thesis, Australian National University, 2009.

[68] Magalhães, J. P., Holdermans, S., Jeuring, J., and Löh, A. Optimizing generics is easy! In *PEPM '10: Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation* (New York, NY, USA, 2010), ACM, pp. 33–42.

[69] Maranget, L. Compiling lazy pattern matching. In *Proceedings of the 1992 ACM conference on LISP and functional programming* (1992), ACM New York, NY, USA, pp. 21–31.

[70] McBride, C. Elimination with a motive. In *Types for Proofs and Programs*, P. Callaghan, Z. Luo, J. McKinna, R. Pollack, and R. Pollack, Eds., vol. 2277 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2002, pp. 727–727.

[71] Mitchell, N., and Runciman, C. Uniform boilerplate and list processing. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell* (September 2007), ACM, pp. 49–60.

[72] Odersky, M., Spoon, L., and Venners, B. *Programming in Scala*. Artima Inc, 2008.

[73] Oliveira, B. C., and Gibbons, J. Scala for generic programmers. In *WGP '08: Proceedings of the ACM SIGPLAN workshop on Generic programming* (New York, NY, USA, 2008), ACM, pp. 25–36.

[74] Peyton Jones, S. L., and Santos, A. L. M. A transformation-based optimiser for Haskell. *Science of Computer Programming 32*, 1–3 (1998), 3–47.

[75] Pope, B. polymorphic recursion (was: Re: Implict parameters and monomorphism). `http://www.mail-archive.com/haskell@haskell.org/msg08549.html`, May 2001.

[76] Reynolds, J. *Theories of Programming Languages*. Cambridge University Press, 1998.

[77] Rodriguez, A., Jeuring, J., Jansson, P., Gerdes, A., Kiselyov, O., and Oliveira, B. C. d. S. Comparing libraries for generic programming in haskell.

[78] Rose, K. H. *Explicit Substitution; Tutorial and Survey*, vol. 96. BRICS, 1996.

[79] Scott, K., and Ramsey, N. When do match-compilation heuristics matter? Tech. rep., Charlottesville, VA, USA, 2000.

[80] Sloane, A. M. Experiences with domain-specific language embedding in scala. In *2nd International Workshop on Domain-Specific Program Development* (2008), L. J and L. Reveillere, Eds.

[81] Somogyi, Z., Henderson, F., and Conway, T. The execution algorithm of mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming 29*, 1-3 (1996), 17 – 64. High-Performance Implementations of Logic Programming Systems.

[82] Sulzmann, M., Chakravarty, M. M. T., Jones, S. P., and Donnelly, K. System f with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation* (New York, NY, USA, 2007), ACM Press, pp. 53–66.

[83] Syme, D., Neverov, G., and Margetson, J. Extensible pattern matching via a lightweight language extension. *Proceedings of the ICFP'07 conference 42*, 9 (2007), 29–40.

[84] Unknown. The mercury project: Comparing mercury and haskell. http://www.mercury.csse.unimelb.edu.au/information/comparison_with_haskell.html.

[85] van den Brand, M., Klint, P., and Vinju, J. Term rewriting with type-safe traversal functions. *Electronic Notes in Theoretical Computer Science 70*, 6 (2002), 100 – 117. WRS 2002, 2nd International Workshop on Reduction Strategies in Rewriting and Programming - Final Proceedings (FLoC Satellite Event).

[86] van den Brand, M. G. J., Heering, J., Klint, P., and Olivier, P. A. Compiling language definitions: the asf+sdf compiler. *ACM Trans. Program. Lang. Syst. 24* (July 2002), 334–368.

[87] Visser, E. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, C. Lengauer et al., Eds., vol. 3016 of *Lecture Notes in Computer Science*. Spinger-Verlag, June 2004, pp. 216–238.

[88] Wadler, P. *Implementing Functional Languages*. Prentice Hall, 1987, ch. 7.

[89] Wadler, P., and Blott, S. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1989), ACM New York, NY, USA, pp. 60–76.

[90] Wadler, P., et al. The expression problem. *Discussion on the Java-Genericity mailing list* (1998).

[91] Walmsley, P. *XQuery*. O'Reilly Media, March 2007.

[92] Wright, A. K., and Felleisen, M. A syntactic approach to type soundness. *Information and Computation 115* (1992), 38–94.

[93] Yakushev, A. L. R. *Towards Getting Generic Programming Ready for Prime Time*. PhD thesis, 2009.