Partial Evaluation for Dictionary-free Overloading

Mark P. Jones
Yale University, Department of Computer Science,
P.O. Box 2158 Yale Station, New Haven, CT 06520-2158.
jones-mark@cs.yale.edu

Research Report YALEU/DCS/RR-959 April 1993

Abstract

It has long been suggested that parametric polymorphism might be implemented by generating a distinct version of the code for each monomorphic instance of a function that is used in a given program. However, most implementations avoid this approach for fear that it could lead to a substantial increase in the size of the compiled program – a so-called *code explosion* – and that it would complicate the process of separate compilation.

An alternative, used in many systems, is to have a single implementation of each polymorphic function and use a uniform (or boxed) representation for all values. This avoids the risk of code explosion but makes it more difficult to support parametric overloading. In addition, the use of boxed representations for even the most primitive values may carry a significant run-time cost.

This paper presents a new approach that lies between these two extremes. The resulting system can be used to obtain an efficient implementation of overloading and to avoid the use of boxed representations of values in common cases (by reducing it to a particular form of overloading). We describe a compiler for a Haskell-like language that supports overloading using a system of type classes. The compiler generates distinct implementations for each overloaded function that appears in a given program but only a single version of the code for pure polymorphic values. This avoids many of the problems associated with the use of dictionaries in previous implementations of type class overloading. Furthermore, comparing the output from our compiler with that of an earlier dictionary-based implementation we find that, for realistic applications, there is no code explosion.

The new compiler has been obtained by adding a simple partial evaluator to the dictionary-based system. The results of this paper therefore provide further motivation for including a more general partial evaluation system as part of production quality compilers for such languages.

We do not attempt to deal with the interaction between partial evaluation and separate compilation although our results certainly suggest that further work in that area would be beneficial.

1 Introduction

The inclusion of a polymorphic type system in the design of a programming language recognizes the fact that many useful algorithms can be described in a uniform manner without full knowledge about the kind of values on which they operate. For example, if all lists are implemented in the same way within a particular programming language, then the process of calculating the length of a list is independent of the type of values that it contains.

In his seminal paper on polymorphism in programming languages [14], Milner used a polymorphic type system as a means of identifying a large class of programs in an untyped λ-calculus whose execution would not "go wrong". Objects in the underlying semantic model can have many different types; for example, there is a unique identity function $\lambda x.x$ that serves as the identity for all types. With this in mind, it is quite natural to consider an implementation in which all versions of a particular polymorphic function are implemented by the same section of program code. This approach is simple and has worked well in many practical implementations, including those which support separate compilation. Unfortunately, it is difficult to support parametric overloading in this framework; for example, we may need to rely on run-time tags [1] or additional parameters such as dictionaries in Haskell [7, 21]. In addition, it is necessary to use a uniform representation for run-time values, typically a single machine word. Objects that do not fit into this space must be allocated storage elsewhere and represented by a pointer to that value. This is known as a *boxed* representation and can cause substantial performance overheads.

Another possibility is to use a simply-typed λ-calculus as the underlying model, with a semantic domain D_τ for each type τ. This leads to an implementation in which distinct sections of code are used to implement different instances of a polymorphic function. For example, the polymorphic identity function mentioned above is treated as a family of functions, indexed by type, with a different implementation, $\lambda x : \tau.x$, for each type τ. This is by no means a new idea; similar suggestions have been made many times in the past both as an implementation technique (for example, in [15]) and as a theoretical tool for describing the semantics of ML-style polymorphism [16]. It is easy to deal with overloading and to avoid the need for uniform representations using sets of values indexed by type; there is no need to insist that the indexing set includes all types or that the implementations are always substitution instances of a single simply-typed term.

There are two reasons why this approach has not been widely adopted in concrete implementations; first that it might lead to an explosion in the size of the compiled program, second that it does not interact well with the use of separate compilation. It is not particularly difficult to demonstrate how a code explosion might occur. For example, suppose that we define a sequence of functions in a Haskell program using:

$$\begin{array}{lcl}
f_1\ (x,y)\ (u,v) & = & f_0\ x\ u\ \ \&\&\ \ f_0\ y\ v \\
f_2\ (x,y)\ (u,v) & = & f_1\ x\ u\ \ \&\&\ \ f_1\ y\ v \\
f_3\ (x,y)\ (u,v) & = & f_2\ x\ u\ \ \&\&\ \ f_2\ y\ v \\
f_4\ (x,y)\ (u,v) & = & f_3\ x\ u\ \ \&\&\ \ f_3\ y\ v \\
 & \vdots &
\end{array}$$

where f_0 has type $a \to a \to Bool$. It is easy to show that, in the general case, a call to f_i will involve 2^{i-j} different versions of f_j for each $0 \leq j \leq i$. However, this worst-case exponential behaviour may not occur in practice. For example, if the only instance of f_0 that is actually used has type $Int \to Int \to Bool$, then it is only necessary to generate one version of the code for each of the functions f_i.

This paper describes our results with a compiler that uses a combination of the typed and untyped models described above, generating distinct versions of the code for overloaded functions, but only a single version for all other functions. Input programs for the compiler are written in a Haskell-like language with support for user-defined overloading based on an extended system of type classes [7, 11, 21]. All current Haskell systems make use of dictionary values at run-time to implement type class overloading and this can have a significant effect on run-time performance. As a result, some Haskell programmers have resorted to including explicit type information in programs to avoid the use of overloading in performance-sensitive applications, but loosing much of the flexibility that overloading can provide. By generating distinct versions of the code for overloaded operators we have the potential to avoid many of these problems because dictionary values are no longer involved. In effect, we can offer the performance of an explicitly typed program without the need for explicit type signatures, compromising the flexibility of type class overloading.

The dictionary-free implementation is obtained by specializing the output of an previous version of the compiler that does rely on the use of dictionaries at run-time. As might be expected, there is a very close connection with standard techniques used in partial evaluation; a production compiler providing dictionary-free overloading in this way would probably use a more general partial evaluation system to achieve the same kind of results.

The principal objective of the work presented in this paper was to assess what kind of impact, if any, the code explosion problems described above might have in realistic programming examples. The results are very promising, suggesting that, for realistic programs the code explosion problem does not occur. Indeed, in all the cases that we tried, the size of the compiled program was actually reduced!

The remainder of this paper is organized as follows. We begin with a brief introduction to the use of type classes in Haskell in Section 2, describing the way in which overloaded functions are defined and extended to work on a range of datatypes. All current implementations of Haskell are based on the dictionary passing style discussed in Section 3. The need to eliminate the use of dictionaries motivates the use of a form of partial evaluation in Section 4 which produces a dictionary-free implementation of programs using type class overloading. We provide some measurements of program size for a collection of 'realistic' programs using both the dictionary passing style and the dictionary-free implementations. Another benefit of a dictionary-free implementation is that it considerably reduces the motivation for the 'dreaded monomorphism restriction' in Haskell, making it possible to eliminate this in future versions of Haskell. This is discussed in Section 5. Another application, described in Section 6, is to allow the use of more efficient (i.e. non-uniform) representations for certain frequently used data types. This is made possible by treating the familiar constructor functions and pattern matching constructs as overloaded values. Finally, Section 7 gives some pointers to further work, in particular to the problems of combining global partial evaluation and separate compilation.

2 Type classes

Type classes were introduced by Wadler and Blott [21] as a means of supporting overloading (ad-hoc polymorphism) in a language with a polymorphic type system. This section gives a brief overview of the use of type classes in a language like Haskell and provides some examples that will be used in later sections. Further details may be found in [6, 7].

The basic idea is that a type class corresponds to a set of types (called the *instances* of the class) together with a collection of *member functions* (sometimes described as *methods*) that are defined for each instance of the class. For example, the standard prelude in Haskell defines the class *Num* of numeric types using the declaration:

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{class}\ (Eq\ a, Text\ a) \Rightarrow Num\ a\ \textbf{where}} \\
\quad (+), (*), (-) & :: & a \to a \to a \\
\quad negate & :: & a \to a \\
\quad abs, signum & :: & a \to a \\
\quad fromInteger & :: & Integer \to a \\
\quad x - y & = & x + negate\ y
\end{array}$$

The first line of the declaration introduces the name for the class, *Num*, and specifies that every instance a of *Num* must also be an instance of *Eq* and *Text*. These are two additional standard classes in Haskell representing the set of types whose elements can be compared for equality and those types whose values can be converted to and from a printable representation respectively. Note that, were it not for a limited character set, we might well have preferred to write type class constraints such as $Num\ a$ in the form $a \in Num$.

The remaining lines specify the operations that are specific to numeric types including simple arithmetic operators for addition $(+)$, multiplication $(*)$ and subtraction $(-)$ and unary negation *negate*. The *fromInteger* function is used to allow arbitrary integer value to be coerced to the corresponding value in any other numeric type. This is used primarily for supporting overloaded integer literals as will be illustrated below. Notice the last line of the declaration which gives a default definition for subtraction in terms of addition and unary negation. This definition will only be used if no explicit definition of subtraction is given for particular instances of the class.

Instances of the class *Num* are defined by a collection of instance declarations which may be distributed throughout the program in distinct modules. The program is free to extend the class *Num* with new datatypes by providing appropriate definitions. In the special case of the built-in type *Integer* (arbitrary precision integers or bignums), the instance declaration takes the form:

$$\begin{array}{lll} \textbf{instance}\ \mathit{Num}\ \mathit{Integer}\ \textbf{where} & & \\ \quad (+) & = & \mathit{primPlusInteger} \\ & \vdots & \\ \quad \mathit{fromInteger}\ x & = & x \end{array}$$

This assumes that the implementation provides a built-in function *primPlusInteger* for adding two *Integer* values. Note that, for this special case, the implementation of *fromInteger* is just the identity function. The Haskell standard prelude also defines a number of other numeric types as instances of *Num* including fixed precision integers and floating point numbers. The definition of *fromInteger* is typically much more complicated for examples like these.

Other useful datatypes can be declared as instances of *Num*. For example, the following definition is a simplified version of the definition of the type of complex numbers in Haskell:

$$\textbf{data}\ \mathit{Complex}\ a \ = \ a :\!+ a$$

$$\begin{array}{lll} \textbf{instance}\ \mathit{Num}\ a \Rightarrow \mathit{Num}\ (\mathit{Complex}\ a)\ \textbf{where} & & \\ \quad (x :\!+ y) + (u :\!+ v) & = & (x+u) :\!+ (y+v) \\ & \vdots & \\ \quad \mathit{fromInteger}\ x & = & \mathit{fromInteger}\ x :\!+ \mathit{fromInteger}\ 0 \end{array}$$

We can deal with many other examples such as rational numbers, polynomials, vectors and matrices in a similar way.

As a simple example of the use of the numeric class, we can define a generic *fact* function using:

$$\begin{array}{lll} \mathit{fact}\ n & = & \textbf{if}\ n == 0 \\ & & \quad \textbf{then}\ 1 \\ & & \quad \textbf{else}\ n * \mathit{fact}\ (n-1) \end{array}$$

Any integer constant m appearing in a Haskell program is automatically replaced with an expression of the form *fromInteger* m so that it can be treated as an overloaded numeric constant, not just an integer. If we make this explicit, the definition of *fact* above becomes:

$$\begin{array}{lll} \mathit{fact}\ n & = & \textbf{if}\ n == \mathit{fromInteger}\ 0 \\ & & \quad \textbf{then}\ \mathit{fromInteger}\ 1 \\ & & \quad \textbf{else}\ n * \mathit{fact}\ (n - \mathit{fromInteger}\ 1) \end{array}$$

As a result, the *fact* function has type $\mathit{Num}\ a \Rightarrow a \rightarrow a$ indicating that, if n is an expression of type a and a is an instance of *Num*, then *fact* n is also an expression of type a. For example:

$$\begin{array}{lll} \mathit{fact}\ 6 & \Longrightarrow & 720 \\ \mathit{fact}\ (6.0 :\!+ 0.0) & \Longrightarrow & 720.0 :\!+ 0.0 \end{array}$$

The type system ensures that the appropriate definitions of multiplication, subtraction etc. are used in each case to produce the correct results. At the same time, an expression like *fact* $'f'$ will generate a type error since there is no declaration that makes *Char*, the type of characters, an instance of *Num*.

3 Dictionary passing implementation

This section outlines the technique of *dictionary passing* and explains some of the reasons why it is so difficult to produce efficient code in the presence of dictionary values.

The biggest problem in the implementation of overloading is that of finding an efficient and effective way to deal with *method dispatch* – selecting the appropriate implementation for an overloaded operator in a particular situation. One common technique is to attach a tag to the run-time representation of each object; each overloaded function is implemented by inspecting the tags of the values that it is applied to and, typically using some form of lookup table, branching to the appropriate implementation.

Apart from any other considerations about the use of tags, this approach can only deal with certain kinds of overloading. In particular, it cannot be used to implement the *fromInteger* function described in the previous section; the implementation of *fromInteger* depends, not on the type of its argument, but on the type of the value it is expected to return!

An elegant solution to this problem is to separate tags from objects, treating tags as data objects in their own right. For example, we can implement the *fromInteger* function by passing the tag of the result as an extra argument. This amounts to passing type information around at run-time but is only necessary when overloaded functions are involved. There are, in fact, several different choices for the kind of values that are used as tags (corresponding to the notion of *evidence* in the theory of qualified types [8, 9]). For example, Thatte [20] suggests using types themselves as tags, extending the implementation language with a **typecase** construct that supports 'pattern matching' on types to deal with method dispatch.

A more concrete approach – based on the use of *dictionary* values – was introduced by Wadler and Blott [21] and has been adopted in all current Haskell systems. A dictionary consists of a tuple that contains the implementations for each of the member functions in a given class. Superclasses can be dealt with by storing a pointer to the appropriate class in the dictionary. For example, Figure 1 illustrates the structure of a dictionary for the *Num* class, including the auxiliary dictionaries for the superclasses *Eq* and *Text*.

Specific instances of this structure are constructed as necessary using the instance declarations in a program. We use the names of the member functions as selectors that can be applied to a suitable dictionaries to extract the corresponding implementation. For example, if *dictNumInteger* is the dictionary corresponding to the instance declaration for *Num Integer* given in Section 2, then:

$$\begin{array}{lll} (+)\ \mathit{dictNumInteger}\ 2\ 3 & \Longrightarrow & \mathit{primPlusInteger}\ 2\ 3 \\ & \Longrightarrow & 5 \\ \mathit{fromInteger}\ \mathit{dictNumInteger}\ 42 & & \\ & \Longrightarrow & 42 \end{array}$$

Notice that these overloaded primitive functions are dealt with by adding an extra dictionary parameter. The same technique can be used to implement other overloaded functions. For example, adding an extra dictionary parameter to the *fact* function given above and using member functions

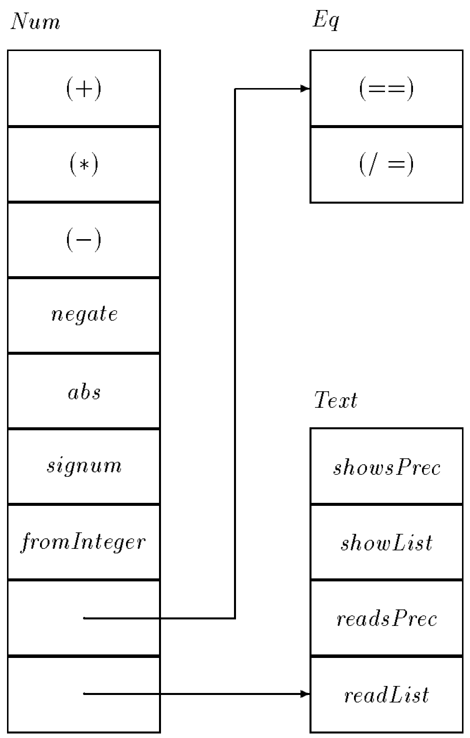


Figure 1: Dictionary structure for the *Num* class

as selectors, we obtain:

$$\begin{array}{l} fact\ d\ n \\ \quad = \ \textbf{if}\ (==)\ (eqOfNum\ d)\ n\ (fromInteger\ d\ 0) \\ \qquad\quad \textbf{then}\ fromInteger\ d\ 1 \\ \qquad\quad \textbf{else}\ (*)\ d\ n\ (fact\ d\ ((-)\ n\ (fromInteger\ d\ 1))) \end{array}$$

(writing *eqOfNum* for the selector function which extracts the superclass dictionary for *Eq* from the dictionary for the corresponding instance of *Num*.) Further details about the translation process are are given by [2, 17, 19, 21].

The dictionary passing style is reasonably simple to understand and implement and is well-suited to separate compilation; only the general structure of a dictionary and the set of instances of a particular class (both of which can be obtained from module interfaces) are needed to compile code that makes use of operators in that class. Unfortunately, the use of dictionaries also causes some substantial problems:

- Unused elements in a dictionary cause an unwanted increase in code size.
- In the general case, the selectors used to implement method dispatch are higher-order functions. It is well-known that efficient code generation and static analysis are considerably more difficult in this situation.
- The need to construct dictionary values and pass these as additional parameters at run-time adds further overheads.

For the purposes of this paper, we concentrate on the use of dictionaries although many of our comments apply more generally to any implementation of type class overloading which makes use of evidence values at run-time, whatever concrete form that may take.

3.1 Dictionaries increase program size

In an attempt to reduce the size of executable programs produced by a compilation system, many systems use some form of 'tree shaking' to eliminate unused sections of code from the output program. This is particularly important when large program libraries are involved; the standard prelude in Haskell is an obvious example.

The idea of grouping related operations into a single class is certainly quite natural. In addition, it often results in less complicated types. For example, if Haskell used separate classes *Eq*, *Plus* and *Mult* for each of the operators $(==)$, $(+)$ and $(*)$ respectively, then the function:

$$pyth\ x\ y\ z \quad = \quad x * x + y * y == z * z$$

might be assigned the type:

$$(Eq\ a, Plus\ a, Mult\ a) \Rightarrow a \rightarrow a \rightarrow a \rightarrow Bool$$

rather than the simpler:

$$Num\ a \Rightarrow a \rightarrow a \rightarrow a \rightarrow Bool.$$

A disadvantage of grouping together methods like this is that it becomes rather more difficult to eliminate unwanted code as part of the tree shaking process. For example, any program that uses a dictionary for an instance of *Num* will also require corresponding dictionaries for *Eq* and *Text*. Many such programs will not make use of all of the features of the *Text* class, but it is still likely that large portions of the standard prelude dealing with reading and printing values will be included in the output program. In a similar way, even a program where only *Int* arithmetic is used, the need to include a *fromInteger* function as part of the dictionary may result in compiled programs that include substantial parts of the run-time support library for *Integer* bignums.

Another factor that tends to contribute to the size of programs that are implemented using the dictionary passing style is the need to include additional code to deal with the construction of dictionary values (and perhaps to implement the selector functions corresponding to each member function and superclass).

3.2 Dictionaries defeat optimization

It is well known that the presence of higher-order functions often results in significant obstacles to effective static analysis and efficient code generation. Exactly the same kind of problems occur with the use of dictionaries – the selector functions used to implement member functions are (usually) higher-order functions – except that the problems are, if anything, more severe since many of the most primitive operations in Haskell are overloaded.

To illustrate the problems that can occur, consider the following definition of a general purpose function for calculating the sum of a list of numbers[1]:

$$\begin{array}{lll} sum & :: & Num\ a \Rightarrow [a] \rightarrow a \\ sum\ xs & = & loop\ 0\ xs \\ \textbf{where} & loop\ tot\ [\,] & = \ tot \\ & loop\ tot\ (x:xs) & = \ loop\ (tot + x)\ xs \end{array}$$

After the translation to include dictionary parameters this becomes:

$$\begin{array}{llll} sum\ d\ xs & = & loop\ d\ (fromInteger\ d\ 0)\ xs & \\ \textbf{where} & loop\ d\ tot\ [\,] & = & tot \\ & loop\ d\ tot\ (x:xs) & = & loop\ d\ ((+)\ d\ tot\ x)\ xs \end{array}$$

As the original definition is written, it seems reasonable that we could use standard strictness analysis techniques to discover that the second (accumulating) argument in recursive calls to *loop* can be implemented using call-by-value so that the calculation of the sum runs in constant space. Unfortunately, this is not possible because we do not know enough about the strictness properties of the function $(+)\ d$; even if the implementation of addition is strict in both arguments for every instance of *Num* in a particular program, it is still possible that a new instance of *Num* could be defined in another module which does not have this property. The code for *sum* has to be able to work correctly with this instance and hence the implementation of *sum* will actually require space proportional to the length of the list for any instance of *Num*.

The implementation of *sum* given above also misses some other opportunities for optimization. For example, if we were summing a list of machine integers (values of type *Int*) then the second argument to *loop* could be implemented as an unboxed value, and the addition could be expanded inline, ultimately being implemented by just a couple of low-level machine instructions.

3.3 The run-time overhead of dictionary passing

There are a number of additional run-time costs in an implementation of type class overloading based on dictionaries. The construction of a dictionary involves allocation and initialization of its components. Our experience suggests that the number of distinct dictionaries that will be required in a given program is usually rather small (see Section 4.3 for more concrete details) so the cost of dictionary construction should not, in theory, be too significant. However, there are many examples which show that the same dictionary value may be constructed many times during the execution of a single program. Some of these problems can be avoided by using more sophisticated translation schemes when dictionary parameters are added to Haskell programs, but others cannot be avoided because of the use of *context reduction* in the Haskell type system (see [9] for further details).

There is also a question about whether dictionary construction is implemented lazily or eagerly. In the first case, every attempt to extract a value from a dictionary must be preceded by a check to trigger the construction of the dictionary if it has not previously been evaluated. (Of course, this is entirely natural for a language based on lazy evaluation and standard techniques can be used to optimize this process in many cases.) The second alternative, eager construction of dictionary values, risks wasted effort building more of the dictionary structure than is needed. This is a real concern; with the definitions in the standard prelude for Haskell, the dictionary for the instance *RealFloat* (*Complex Double*) involves between 8 and 16 additional superclass dictionaries, depending on the way in which equivalent dictionary values are shared. With a lazy strategy, all of the member functions for the *RealFloat* class can be accessed after building only a single dictionary.

Finally, there is a potential cost of manipulating the additional parameters used to pass dictionary values. For example, it may be necessary to generate extra instructions and to reserve additional space in a closure (or allocate more application nodes in a graph-based implementation) for dictionaries. However, our experience suggests that these costs are relatively insignificant in practice.

3.4 The use of explicit type signatures

One common optimization in current Haskell systems is to recognize when the dictionaries involved in an expression are constant and to extract the implementations of member functions at compile-time. This often requires the programmer to supply additional type information in the form of an explicit type declarations such as:

$$fact\ ::\ Int \rightarrow Int.$$

(*Int* is a built-in type of fixed-size integers, typically 32 bit values.) The translation of *fact* can then be simplified to:

$$\begin{array}{l} fact\ n \\ \quad = \ \textbf{if}\ primEqInt\ n\ zero \\ \qquad\quad \textbf{then}\ one \\ \qquad\quad \textbf{else}\ primMultInt\ n\ (fact\ (primSubInt\ n\ one)) \end{array}$$

where *primEqInt*, *primMulInt* and *primSubInt* are primitive functions which can be recognized by the code generator and compiled very efficiently, and *zero* and *one* are the obvious constant values of type *Int*.

In current Haskell systems, adding an explicit type signature like this in small benchmark programs which make extensive use of overloading typically gives at least a ten-fold improvement in program execution time! (Of course, the speedups are much more modest for 'real-world' programs.) As a result, it has become quite common to find Haskell programs that are sprinkled with type annotations, not so much to help resolve overloading, but rather to avoid it altogether.

We should also mention that identifying the correct place to insert an explicit type signature to avoid overloading is not always easy. For example, changing the type declaration for the *sum* function in Section 3.2 to:

$$sum\ ::\ [Int] \rightarrow Int$$

does not necessarily avoid the use of dictionaries. According to the standard typing rules, the local function *loop* will still be treated as having the polymorphic overloaded type $Num\ a \Rightarrow a \rightarrow [a] \rightarrow a$ and the corresponding translation is:

$$\begin{array}{llll} sum\ xs & = & loop\ dictNumInt\ zero\ xs & \\ \textbf{where} & loop\ d\ tot\ [\,] & = & tot \\ & loop\ d\ tot\ (x:xs) & = & loop\ d\ ((+)\ d\ tot\ x)\ xs \end{array}$$

[1] Augustsson [2] uses essentially the same example to demonstrate similar problems.

Instead, we have to add a type signature to the local definition:

$$\begin{array}{lll} \mathit{sum}\ \mathit{xs} & = & \mathit{loop}\ 0\ \mathit{xs} \\ \textbf{where}\ \mathit{loop} & :: & \mathit{Int} \rightarrow [\mathit{Int}] \rightarrow \mathit{Int} \\ \quad \mathit{loop}\ \mathit{tot}\ [\,] & = & \mathit{tot} \\ \quad \mathit{loop}\ \mathit{tot}\ (x : \mathit{xs}) & = & \mathit{loop}\ (\mathit{tot} + x)\ \mathit{xs} \end{array}$$

This allows the code generator to use the optimizations described in Section 3.2, but the flexibility and generality of the original *sum* function has been completely lost.

4 A dictionary-free implementation

Haskell type classes have proved to be a valuable extension to the Hindley-Milner type system used in languages like ML. The standard prelude for Haskell included in [7] illustrates this with a range of applications including equality, ordering, sequencing, arithmetic, array indexing, and parsing/displaying printable representations of values. However, it is clear from the comments in the previous section that any implementation based on the use of dictionary passing faces some serious obstacles to good run-time performance.

The possibility of a dictionary-free implementation was mentioned by Wadler and Blott in the original paper introducing the use of type classes [21], together with the observation that this might result in an exponential growth in code size. This was illustrated by considering the function:

$$\mathit{squares}\ (x, y, z) \quad = \quad (x * x, y * y, z * z)$$

which has type:

$$(\mathit{Num}\ a, \mathit{Num}\ b, \mathit{Num}\ c) \Rightarrow (a, b, c) \rightarrow (a, b, c).$$

Notice that, even if there are only two instances of the class *Num*, there are still eight possible versions of this function that might be required in a given program.

But do examples like this occur in real programs? Other situations where the apparent problems suggested by theoretical work do not have any significant impact on practical results are well known. For example, it has been shown that the complexity of the Damas-Milner type inference algorithm is exponential, but the kind of examples that cause this do not seem to occur in practice and the algorithm behaves well in concrete implementations.

In an attempt to investigate whether expanding programs to avoid the use of dictionaries results in a code explosion, we have developed a compiler for Gofer, a functional programming system based on Haskell, that does not use make use of dictionary parameters at run-time. The compiler is based on an earlier version whose output programs did rely on the use of dictionaries. The main difference is the use of a specialization algorithm, described in Section 4.1, to produce specialized versions of overloaded functions. Not surprisingly, the same results can be obtained using a more general partial evaluation system and we discuss this in Section 4.2. Comparing the sizes of the programs produced by the two different versions of the compiler, we have been able to get some measure of the potential code explosion. We had expected that expanding out all of the definitions of overloaded functions in realistic applications would produce larger compiled programs, but we hoped that our experiments would show that the increases in code size are usually fairly modest. To our surprise, we were somewhat surprised to find that, for all the examples we have tried, the 'expanded' program is actually smaller than the original dictionary based version!

4.1 A formal treatment of specialization

This section describes an algorithm for converting the code for a dictionary-based implementation of a program with overloading to a specialized form that does not involve dictionaries. Although our presentation is rather formal, the algorithm itself is simple enough; starting with the top-level expression in a given program, we replace each occurrence of an overloaded function f, together with the dictionary values d that it is applied to, with a new variable, f'. The resulting expression is enclosed in the scope of a new definition for f' that is obtained by specializing the original definition for f and using the corresponding dictionary arguments d.

4.1.1 Source language

The algorithm takes the translations produced by the type checker [9] as its input; the syntax of these terms is given by the following grammar:

$$\begin{array}{rcll} M & ::= & xe & \textit{variables} \\ & | & M\ M & \textit{application} \\ & | & \lambda x.M & \textit{abstraction} \\ & | & \textbf{let}\ B\ \textbf{in}\ M & \textit{local definitions} \end{array}$$

The symbol x ranges over a given set of term variables, and e ranges over (possibly empty) sequences of dictionary expressions. In addition, B ranges over finite sets of *bindings* (pairs of the form $x = \lambda v.M$) in which no variable x has more than one binding. The symbol v used here denotes a (possibly empty) sequence of dictionary parameters. The set of variables x bound in B will be written $\mathit{dom}\ B$. An additional constraint that is guaranteed by the type system but not reflected by the grammar above is that every occurrence of a variable in a given scope has the same number of dictionary arguments (equal to the number of class constraints in the type assigned to the variable and to the number of dictionary parameters in the defining binding).

Note also that the language used in [9] allows only single bindings in local definitions; of course, an expression of the form **let** $x = M$ **in** M' in that system can be represented as **let** $\{x = M\}$ **in** M' in the language used here. The motivation for allowing multiple bindings is that we want to describe the specialization algorithm as a source to source transformation and it may be necessary to have several specialized versions of a single overloaded function.

4.1.2 Specialization sets

Motivated by the informal comments above, we describe the algorithm using a notion of *specializations* each of which is an expression of the form $f\ d \rightsquigarrow f'$ for some variables f and f' and some sequence of dictionary parameters d. As a convenience, we will always require that f' is a 'new' variable that is not used elsewhere. Since a given program may actually require several specialized versions of some overloaded functions, we will usually work with (finite) sets of specializations. To ensure that these sets are consistent, we will restrict ourselves to those sets S such that:

$$(x\ d \rightsquigarrow x'),\ (y\ e \rightsquigarrow x') \in S \quad \Rightarrow \quad x = y\ \wedge\ d = e.$$

In other words, we do not allow the same variable to represent distinct specializations. This is precisely the condition needed to ensure that any specialization set S can be interpreted as a substitution where each $(x\,d \leadsto x') \in S$ represents the substitution of $x\,d$ for the variable x'. For example, applying the specialization set $\{x\,d \leadsto y\}$ as a substitution to the term $(\lambda y.y)y$ gives $(\lambda y.y)(x\,d)$.

In practice, it is sensible to add the following restrictions in an attempt to reduce the size of specialization sets, and hence the size of compiled programs:

- Avoid duplicated specialization of the same function: if $(x\,d \leadsto x')$, $(x\,d \leadsto y') \in S$, then $x' = y'$.
- Avoid unused specializations: there is no need to include $(x\,d \leadsto x') \in S$ unless x actually occurs with dictionary arguments d in the scope of the original definition of x.

Note however that these conditions are motivated purely by practical considerations and are not required to establish the correctness of the specialization algorithm.

It is convenient to introduce some special notation for working with specialization sets:

- If V is a set of variables, then we defined S_V as:

  $$S_V = \{\, (x\,d \leadsto x') \in S \mid x \notin V \,\}.$$

  In other words, S_V is the specialization set obtained from S by removing any specializations involving a variable in V. As a special case, we write S_x as an abbreviation for $S_{\{x\}}$.
- For any specialization set S, we define:

  $$\mathit{Vars}\ S \;=\; \{\, x \mid (x\,d \leadsto x') \in S \,\}.$$

- We define the following relation to characterize the specialization sets that can be obtained from a given set S, but with different specializations for variables bound in a given B:

  $$\begin{aligned} S' \ \mathit{extends}\ (B,S) \iff \exists S''.\ & \mathit{Vars}\ S'' \subseteq \mathit{dom}\ B\ \wedge \\ & S' = S_{(\mathit{dom}\ B)} \cup S''. \end{aligned}$$

4.1.3 The specialization algorithm

The specialization algorithm is described using judgments of the form $S \vdash M \leadsto M'$ and following the rules in Figure 2. The expression M is the input to the algorithm and the output is a new term M' that implements M without the use of dictionaries and a specialization set S for overloaded functions that appear free in M.

Note that there are two rules for dealing with variables. The first, (*var-let*), is for variables that are bound in a **let** expression or defined in the initial top-level environment; these are the only places that variables can be bound to overloaded values, and hence the only places where specializations might be required. The second rule, (*var-λ*), deals with the remaining cases; i.e. variables bound by a λ-abstraction or variables defined in a **let** expression that are not overloaded. Although it is beyond the scope of this paper, we mention that this distinction can be characterized more formally using the full set of typing rules for the system.

$$(\mathit{var\text{-}let})\qquad \frac{(xd \leadsto x') \in S \qquad e = d}{S \vdash xe \leadsto x'}$$

$$(\mathit{var\text{-}\lambda})\qquad \frac{x \notin \mathit{Vars}\ S}{S \vdash x \leadsto x}$$

$$(\mathit{app})\qquad \frac{S \vdash M \leadsto M' \qquad S \vdash N \leadsto N'}{S \vdash MN \leadsto M'N'}$$

$$(\mathit{abs})\qquad \frac{S_x \vdash M \leadsto M'}{S \vdash \lambda x.M \leadsto \lambda x.M'}$$

$$(\mathit{let})\qquad \frac{S, S' \vdash B \leadsto B' \qquad S' \vdash M \leadsto M' \qquad S'\ \mathit{extends}\ (B,S)}{S \vdash \mathbf{let}\ B\ \mathbf{in}\ M \leadsto \mathbf{let}\ B'\ \mathbf{in}\ M'}$$

Figure 2: Specialization algorithm

The hypothesis $e = d$ in the rule (*var-let*) implies the compile-time evaluation of the dictionary expressions e to dictionary constants d. In order for the specialization algorithm to be part of a practical compiler, we need to ensure that this calculation can always be carried out without risk of non-termination. See Section 4.1.6 for further comments.

We should also mention the judgment $S, S' \vdash B \leadsto B'$ used as a hypothesis in the rule (*let*). This describes the process of specializing a group of bindings B with respect to a pair of specialization sets S and S' to obtain a dictionary-free set of bindings B' and is defined by:

$$\begin{aligned} & S, S' \vdash B \leadsto B' \\ & \iff \\ B' = \{\, x' = N' \mid {} & (x = \lambda v.N) \in B \\ & \wedge\ (xe \leadsto x') \in S' \\ & \wedge\ S \vdash [e/v]N \leadsto N' \,\} \end{aligned}$$

Note that, for each variable bound in B, only those that also appear in $\mathit{Vars}\ S'$ will result in corresponding bindings in B'. Assuming we follow the suggestions in the previous section and do not include unused specializations in S', then the specialization algorithm also provides tree shaking, eliminating redundant definitions from the output program.

It is also worth mentioning that the (*let*) rule can very easily be adapted to deal with recursive bindings (often written using **let rec** in place of **let**). All that is necessary is to replace $S, S' \vdash B \leadsto B'$ with $S', S' \vdash B \leadsto B'$.

The motivation for specialization was to produce a dictionary-free implementation of the input term. It is clear from the definition above that the output from the algorithm does not involve dictionary values, but it remains to show that the two terms are equal. This, in turn, means that we have to be more precise about what it means for two terms to be equal. For the purposes of this paper we will assume only the standard structural equality together with the two axioms:

$$\begin{array}{rcl} \textbf{let } \{x_1 = M_1, \ldots, x_n = M_n\} \textbf{ in } M & & \\ & = & [M_1/x_1, \ldots, M_n/x_n]\, M \\ (\lambda v.M)\, e & = & [e/v]\, M \end{array}$$

The second of these is simply the familiar rule of β reduction, restricted to dictionary values arguments. The careful reader may notice that the statement of this rule uses dictionary parameters and expressions in positions that are not permitted by the grammar in Section 4.1.1. For the purposes of the following theorem, we need to work in the slightly richer language of [9] that allows arbitrary terms of the form Me or $\lambda v.M$.

With this notion of equality, we can establish the correctness of the specialization algorithm as:

Theorem 1 *If $S \vdash M \leadsto M'$, then $M = SM'$.*

Proof: The proof is by induction on the structure of $S \vdash M \leadsto M'$ and is straightforward, except perhaps for the (let) rule. In that case we have a derivation of the form:

$$\frac{S, S' \vdash B \leadsto B' \quad S' \ \mathit{extends}\ (B, S) \quad S' \vdash M \leadsto M'}{S \vdash \textbf{let } B \textbf{ in } M \leadsto \textbf{let } B' \textbf{ in } M'}$$

The required equality can now be established using the following outline:

$$\begin{array}{rcll} \textbf{let } B \textbf{ in } M & = & [\lambda v.N/x]M & \\ & = & [\lambda v.N/x](S'M') & (*) \\ & = & [\lambda v.N/x][xe/x'](SM') & \\ & = & [(\lambda v.N)e/x'](SM') & \\ & = & [[e/v]N/x'](SM') & \\ & = & [SN'/x'](SM') & (*) \\ & = & S([N'/x']M') & \\ & = & S(\textbf{let } B' \textbf{ in } M') & \end{array}$$

(The two steps labeled $(*)$ follow by induction. The other steps are justified by the properties of substitutions.) □

4.1.4 A simple example

This section illustrates the way that the specialization algorithm works by considering how it deals with a simple input term:

$$\textbf{let } \{f = \lambda v.\lambda x.(+)\ v\ x\ x\} \textbf{ in } f\ d\ \mathit{one}$$

where d denotes the dictionary for $\mathit{Num\ Int}$. We begin with the specialization set $S = \{(+)\ d \leadsto \mathit{primPlusInt}\}$. Writing $B = \{f = \lambda v.N\}$, $N = \lambda x.(+)\ v\ x\ x$, $M = f\ d\ \mathit{one}$ and using the rule (let), we need to find B' and M' such that:

$$S \vdash \textbf{let } B \textbf{ in } M \leadsto \textbf{let } B' \textbf{ in } M'$$

where $S, S' \vdash B \leadsto B'$ and $S' \vdash M \leadsto M'$ for some S' such that $S'\ \mathit{extends}\ (B, S)$. Taking $S' = S \cup \{f\ d \leadsto f'\}$, it follows that $M' = f'\ \mathit{one}$. We can also calculate:

$$\begin{array}{rcl} B' & = & \{\, f' = N' \mid S \vdash [d/v]N \leadsto N' \,\} \\ & = & \{\, f' = N' \mid S \vdash \lambda x.(+)\ d\ x\ x \leadsto N' \,\} \\ & = & \{\, f' = \lambda x.\mathit{primPlusInt}\ x\ x \,\} \end{array}$$

Hence the complete specialized version of the original term is:

$$\textbf{let } \{f' = \lambda x.\mathit{primPlusInt}\ x\ x\} \textbf{ in } f'\ \mathit{one}.$$

4.1.5 The treatment of member functions

Specializations involving member functions can be handled a little more efficiently than suggested by the description above. In particular, given a specialization of the form $(m\ d \leadsto x')$ where m is a member function and d is an appropriate dictionary, there is no need to generate an additional binding for x'. Instead we can extract the appropriate value M from d during specialization, find its specialized form M' and use that in place of x' in the rule $(\mathit{var\text{-}let})$. Thus specialization of member functions might be described by a rule of the form:

$$(\mathit{var\text{-}member}) \qquad \frac{m\ e = M \quad S \vdash M \leadsto M'}{S \vdash m\ e \leadsto M'}$$

The expression $m\ e = M$ represents the process of evaluating e to obtain a dictionary d, and extracting the implementation M of the member function for m. This rule is essential for ensuring that the output programs produced by specialization do not include code for functions that would normally be included in dictionaries, even though they are never actually used in the program.

4.1.6 Termination

Were it not for the evaluation of dictionary expressions in rule $(\mathit{var\text{-}let})$ and the specialization of member functions in rule $(\mathit{var\text{-}member})$, it would be straightforward to prove termination of the specialization algorithm by induction on the structure of judgments of the form $S \vdash M \leadsto M'$. To establish these additional termination properties (for Gofer and Haskell), it is sufficient to observe that both the set of overloaded functions and the set of dictionaries involved in any given program are finite and hence there are only finitely many possible specializations. (We assume that a cache/memo-function is used to avoid repeating the specialization of any given function more than once.)

The fact that there are only finitely many dictionaries used in a given program depends critically on the underlying type system. In particular, it has been suggested that the Haskell type system could be modified to allow definitions such as:

$$\begin{array}{lcl} f & :: & \mathit{Eq}\ a \Rightarrow a \rightarrow \mathit{Bool} \\ f\ x & = & x == x\ \&\&\ f\ [x]. \end{array}$$

This would not be permitted in a standard Hindley/Milner type system since the function f is used at two different instances within its own definition. Attempting to infer the type assigned to f leads to undecidability, but this can be avoided if we insist that an explicit type signature is included as part of its definition. The set of dictionaries that are required to evaluate the expression $f\ 0$ is infinite and the specialization algorithm will not terminate with this program. Fortunately, examples like this are usually quite rare. If the Haskell type system is extended in this way, then it will be necessary to use the dictionary passing implementation to deal with examples like this, even if dictionaries can be avoided in most other parts of the program.

4.2 The relationship with partial evaluation

Techniques for program specialization have already been widely studied as an important component of *partial evaluation*. Broadly speaking, a partial evaluator attempts to produce an optimized version of a program by distinguishing static data (known at compile-time) from dynamic data (which is not known until run-time). This process is often split into two stages:

- **Binding-time analysis**: to find (a safe approximation of) the set of expressions in a program that can be calculated at compile-time, and add suitable annotations to the source program.
- **Specialization**: to calculate a specialized version of the program using the binding-time annotations as a guide.

The specialization algorithm described here fits very neatly into this framework. One common approach to binding time analysis is to translate λ-terms into a two-level λ-calculus that distinguishes between dynamic and static applications and abstractions [5]. The dynamic versions of these operators are denoted by underlining, thus M_N denotes an application that must be postponed until run-time, while $M\ N$ can be specialized at compile-time. Any λ-term can be embedded in the two-level system by underlining all applications and abstractions, but a good binding time analysis will attempt to avoid as many underlinings as possible.

For the purposes of the specialization algorithm described here, all the binding time analysis need do is mark standard abstractions and applications as delayed, flagging the corresponding dictionary constructs for specialization with the correspondence:

$$\begin{array}{rl} \begin{array}{r}\text{standard}\\ \text{operations}\end{array} & \left\{\begin{array}{ccc} \lambda x.M & \sim & \underline{\lambda} x.M \\ M\ N & \sim & M\_N \end{array}\right\} \text{delayed} \\[2ex] \begin{array}{r}\text{dictionary}\\ \text{operations}\end{array} & \left\{\begin{array}{ccc} \lambda v.M & \sim & \lambda v.M \\ M\ e & \sim & M\ e \end{array}\right\} \begin{array}{l}\text{eliminated by}\\ \text{specialization}\end{array} \end{array}$$

Thus dictionary specialization could be obtained using a more general partial evaluator, using the distinction between dictionaries and other values to provide binding time information. Even better, we could use this information as a supplement to the results of a standard binding-time analysis to obtain some of the other benefits of partial evaluation in addition to eliminating dictionary values.

4.3 Specialization in practice

The specialization algorithm presented here has been implemented in a modified version of the Gofer compiler, translating input programs to C via an intermediate language resembling G-code.

Figure 3 gives a small sample of our results, comparing the size of the programs produced by the original dictionary-based implementation with those obtained by partial evaluation. For each program, we list the total number of supercombinators in the output program, the number of G-code instructions and the size of the stripped executable compiled with `cc -O` on a NeXTstation Turbo (68040) running NeXTstep 3.0. The figures on the first row are for the dictionary-based implementation and the expressions n/m indicates that a total of n words are required to hold the m distinct dictionaries that are required by the program. The figures in the second row are for the partially evaluated version, and each expression of the form $n \rightsquigarrow m$ indicates that,

Program name	Total number of supercombinators		G-code instrs	Executable size
`anna`	1509	(814/151)	58,371	851,968
	1560	($170 \rightsquigarrow 259$)	56,931	819,200
`veritas`	1032	(105/22)	32,094	499,712
	990	($36 \rightsquigarrow 49$)	30,596	483,328
`infer`	394	(67/13)	6,069	131,072
	361	($29 \rightsquigarrow 43$)	5,210	114,688
`prolog`	256	(76/14)	5,590	114,688
	177	($21 \rightsquigarrow 32$)	3,207	81,920
`expert`	235	(66/12)	5,774	114,688
	141	($23 \rightsquigarrow 28$)	3,315	81,920
`calendar`	188	(46/8)	3,901	90,112
	86	($8 \rightsquigarrow 9$)	1,273	49,152
`lattice`	190	(293/48)	3,880	90,112
	134	($47 \rightsquigarrow 101$)	1,810	57,344

Figure 3: Code size indicators

of the total number of supercombinators used in the program, m supercombinators were generated by specialization from n distinct overloaded supercombinators in the original program.

The programs have been chosen as examples of realistic applications of the Gofer system:

- The largest program, `anna` is a strictness analyzer written by Julian Seward. Including the prelude file, the source code runs to a little over 15,000 lines spread over 30 script files.
- `veritas` is a theorem prover written by Gareth Howells and taken from a preliminary version of the Glasgow `nofib` benchmark suite.
- `infer` is a Hindley/Milner type checker written by Philip Wadler as a demonstration of the use of monads.
- `prolog` is an interpreter for a small subset of Prolog.
- `expert` is an minimal expert system written by Ian Holyer.
- `calendar` is a small program for printing calendars, similar to the Unix `cal` command.
- `lattice` is a program for enumerating the elements of the lattice D_3 where $D_0 = \mathit{Bool}$ and $D_{n+1} = D_n \rightarrow D_n$ as described in [10]. It is included here as an example of a program that makes particularly heavy use of overloading (as the figures indicate, 75% of the supercombinators in the output program are the result of specialization).

The same prelude file was used for all these tests; a version of the Gofer standard prelude modified to provide closer compatibility with Haskell (including, in particular, a full definition of the *Text* class). Some of these programs made use of Haskell-style derived instances. This allows the programmer to request automatically generated instance declarations for standard type classes when defining a new datatype. Our system does not currently support derived instances and hence it was sometimes necessary to add explicit declarations. It is worth mentioning that, in the case of the `anna` benchmark, the code for derived instances caused an increase in the size of the final executable of over 15% for both versions of the compiler.

These figures are of interest in their own right; we are not aware of any previous work to make a quantitative assessment of the degree to which overloading is used in realistic applications. For all of the examples listed here, the output program produced by specialization is smaller than the dictionary-based version; in fact, we have yet to find an example where the dictionary-based version of the code is smaller! Not surprisingly, the benefits are greatest for the smaller programs. But even for the larger examples it seems clear that the ability to eliminate redundant parts of dictionaries and to avoid manipulating dictionary parameters more than 'pays' for the increase in code size due to specialization.

In the special case of the anna the specialization algorithm increases compile-time (i.e. translation to C) by approximately 15%, from 20.3 user seconds for the dictionary passing version to 23.2 when specialization is used. However, the code generator is very simple minded and we would expect that a high quality, optimizing code generator would have a more significant effect. It is also possible that there would be further overheads in the presence of separate compilation; Gofer does not support the use of modules; a program is just a sequence of script files loaded one after the other.

The time required to translate Gofer code to C is only a fraction of the time required to compile the C code. Using anna again as a typical example, translation to C takes only 3% of the total compilation time. Furthermore, the fact that the specialized version of the program is a little smaller than the dictionary-based version means that the total compile-time is actually slightly lower when specialization is used. Clearly, there are much more pressing concerns than the relatively small costs associated with a more sophisticated C code generator.

The run-time performance of our programs is improved only marginally by the use of partial evaluation; our code generator does not carry out any of the optimizations described in Section 3.2; unlike most Haskell systems, the addition of explicit type signatures as described in Section 3.4 does not give any noticeable increase in performance for either the dictionary or specialization based implementations. In addition, the implementation of dictionaries in Gofer is already very efficient, avoiding most of the problems described in Section 3.3. In particular, all dictionary values are constructed before the execution of a program begins so there is no need to produce code for dictionary constructor functions, there are no problems with repeated construction, and dictionary components can be extracted without having to check for unevaluated dictionaries. We would expect much greater increases in run-time performance in a system using the standard Haskell implementation of dictionaries together with a more sophisticated code generator.

5 Goodbye to the monomorphism restriction!

One of the most controversial features of Haskell is the so-called *monomorphism restriction* which limits the combination of polymorphism and overloading in certain cases. The current definition of Haskell imposes a much less severe restriction than earlier versions when it was sometimes referred to informally as the *'dreaded' monomorphism restriction*. Nevertheless, it remains as an added complication to the language and still causes a few surprises for beginners.

The principal motivation for using any form of monomorphism restriction is to avoid problems with the **let** ... **in** ... construct. From a type-theoretic perspective, this construct is intended purely as a means of defining new polymorphic values within a program. However, in practice, the same construct is also used for other purposes such as shared evaluation or creating cyclic data structures. Unfortunately, in an implementation of type class overloading based on the use of dictionaries, these two applications may conflict with one another. It is necessary to compromise either polymorphism or sharing for the benefit of the other.

To illustrate this, the Haskell report [7] suggests the following expression as a typical example:

$$\mathbf{let}\ x = \mathit{fact}\ \mathit{1000}\ \mathbf{in}\ (x, x).$$

The *fact* function used here is assumed to be overloaded with type $Num\ a \Rightarrow a \rightarrow a$. Furthermore, the integer value *1000* is implicitly treated as an overloaded constant of type $Num\ a \Rightarrow a$. Applying an unrestricted form of the standard type checking algorithm to this expression allows us to assign a polymorphic type $Num\ a \Rightarrow a$ to the local definition of x and use this value at two different instances to obtain a type $(Num\ a,\ Num\ b) \Rightarrow (a, b)$ for the complete expression. The corresponding translation is:

$$\begin{array}{ll}\mathbf{let} & x\ d \ = \ \mathit{fact}\ d\ (\mathit{fromInteger}\ d\ 1000) \\ \mathbf{in} & (x\ d_a, x\ d_b)\end{array}$$

where d_a and d_b are two (potentially distinct) dictionaries for the *Num* class. The problem here is that, whereas the user may have expected the definition of x to be shared by the two components of the pair, the translation may actually repeat the evaluation of *fact 1000*, even if the two components of the pair will actually be used at the same type!

In effect, there are situations where the type system must choose between sharing and polymorphism. The problems and confusion occur when this choice differs from what the programmer expects.

In an implementation of type class overloading based on the techniques described in this paper, the need for any form of monomorphism restriction is significantly reduced[2]. For the example above, if the two components of the resulting pair are both subsequently used with distinct types then the first step in the specialization of the translation produces:

$$\begin{array}{lll}\mathbf{let} & x_a & = \ \mathit{fact}\ d_a\ (\mathit{fromInteger}\ d_a\ 1000) \\ & x_b & = \ \mathit{fact}\ d_b\ (\mathit{fromInteger}\ d_b\ 1000) \\ \mathbf{in} & (x_a, x_b) & \end{array}$$

before going on to produce appropriate specializations of *fact* and *fromInteger*. If, on the other hand, both components are used at a single type then the first step in the specialization process gives:

$$\mathbf{let}\ x = \mathit{fact}\ d\ (\mathit{fromInteger}\ d\ \mathit{1000})\ \mathbf{in}\ (x, x)$$

and there is no loss of sharing.

Thus the system is able to choose between sharing and polymorphism (and possibly other alternatives in-between) based on the way in which defined values are actually used in the program under consideration rather than making some, almost arbitrary decision based on the syntactic form of the definitions for those values.

[2] I am indebted to Martin Odersky for this observation

6 Specialized representation

The decision to implement all instances of a polymorphic function by a single section of code forces us to use a uniform representation for those arguments whose type may vary between different calls of that function. One common approach is to represent objects by a pointer to some part of the heap where the object is stored; this is often referred to as a *boxed* representation, while the object that is pointed to is called an *unboxed* value. The amount of storage required for the unboxed representation of a value (and of course, its interpretation) will vary depending on the type of the object. On the other hand, by using boxed values in the implementation of polymorphic functions, all objects are represented in the same way, independently of their type.

Unfortunately, the use of boxed representations for polymorphic functions makes it more difficult to use unboxed representations when the types of values are fully determined at compile-time. Extracting values from their boxed representation to carry out some operation and then boxing the result have significant overheads. For example, creating a new boxed value will often involve storage allocation. In addition, the use of boxed values may limit the usefulness of standard compiler optimizations such as passing function arguments and results in registers. Some solutions to this problem have been proposed but require, either that the language is extended to deal explicitly with boxed/unboxed representations as in [18], or that we use a more sophisticated type system to discover where coercions between boxed and unboxed representations are required as in [12, 13].

This section describes a new approach to these problems, avoiding the need for uniform boxed representations by generating specialized versions of polymorphic functions. Obviously, this fits in very closely with the work in the first part of the paper; indeed, we show how this can be described as a particular example of overloading using a system of parametric type classes [4].

We should point out that, unlike the work described in the previous sections of this paper, the ideas described here have not yet been implemented. A second caveat is that the use of non-uniform representations is likely to be more useful in a strict language than in a non-strict language. This is a result of the fact that types in the latter provide less information about the run-time representation of values than those in the former. For example, a value of type *Int* in a strict programming language can always be guaranteed to be an integer. In a non-strict language we must also allow for the possibility that the value is an unevaluated expression which must be evaluated before the corresponding integer value is obtained.

6.1 Datatypes and boxed values

We start by recalling how new datatypes are introduced in Haskell. For example, one standard way of defining a type of lists is given by:

$$\textbf{data}\ List\ a \ = \ Nil \mid Cons\ a\ (List\ a)$$

This is a compact way of defining several new related values, the most obvious of which are the unary type constructor *List* and the constructor functions:

$$\begin{array}{lcl} Nil & :: & List\ a \\ Cons & :: & a \rightarrow List\ a \rightarrow List\ a \end{array}$$

that can be used to build values of the new type. In addition, we must also provide some means of supporting pattern matching, for example, using a function:

$$caseList \ :: \ List\ a \rightarrow b \rightarrow (a \rightarrow List\ a \rightarrow b) \rightarrow b.$$

This last component uses a standard technique for encoding datatypes in the λ-calculus; the intended semantics of *caseList* can be described by:

$$\begin{array}{lcllcl} caseList\ d\ n\ c & = & \textbf{case}\ d\ \textbf{of} & Nil & \rightarrow & n \\ & & & Cons\ x\ xs & \rightarrow & c\ x\ xs \end{array}$$

For example, the familiar *map* function defined by:

$$\begin{array}{lcl} map & :: & (a \rightarrow b) \rightarrow (List\ a \rightarrow List\ b) \\ map\ f\ Nil & = & Nil \\ map\ f\ (Cons\ x\ xs) & = & Cons\ (f\ x)\ (map\ f\ xs) \end{array}$$

might be implemented by translating it to:

$$\begin{array}{l} map\ f\ ys \\ \quad = \ caseList\ ys\ Nil\ (\backslash x\ xs \rightarrow Cons\ (f\ x)\ (map\ f\ xs)) \end{array}$$

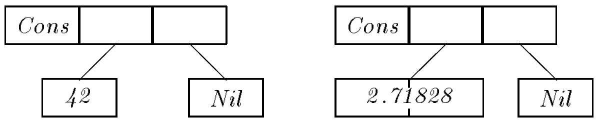
(Note that we would not expect or require the programmer to make explicit use *caseList*; functions like this are intended for internal use only.)

Some authors do not consider functions like *caseList* to be one of the values defined by a datatype definition, but this is only possible because they have a particular concrete implementation of constructor functions and **case** expressions in mind. For example, the unboxed representations for expressions of the form *Nil* and *Cons x xs* might be:

Nil

Cons	*x*	*xs*

where each box represents a single machine word. If we use this representation for all types of list then the two components in a *Cons* value must be boxed values. Hence the singleton lists containing the values *42* :: *Int* and *2.71828* :: *Double* would be represented by:



respectively, assuming that floating point values of type *Double* require two machine words while integers of type *Int* require only one. (A concrete implementation may attach tags to the numeric values in these examples, but this has no real bearing on the current discussion.)

6.2 Specialized representations using overloading

A slightly more efficient way to represent the two lists above would be to use unboxed representations of the form:

ConsI	*42*	*Null*

ConsD	*2.71828*	*Null*

allowing the size of a *Cons* value to vary and using a 'null pointer', *Null*, as the representation of *Nil*. The problem with this idea is that there is no simple rule for determining the size of the x field or the starting position of the xs field in a list of the form *Cons* x xs and hence it is difficult to implement functions like *map* that work with both kinds of list. One possibility would be to use different tags such as the *ConsI* and *ConsD* tags above and allow the implementation of *map* to obtain the required information by inspecting these tags. However, it would certainly be better if we could avoid this interpretive overhead.

A better way to deal with this problem is to treat the constructor functions and the associated pattern matching constructs as overloaded values:

$$
\begin{array}{l}
\textbf{class } \textit{list} :: \textit{List } a \textbf{ where} \\
\quad \begin{array}{lll}
\textit{Nil} & :: & \textit{list} \\
\textit{Cons} & :: & a \rightarrow \textit{list} \rightarrow \textit{list} \\
\textit{caseList} & :: & \textit{list} \rightarrow b \rightarrow (a \rightarrow \textit{list} \rightarrow b) \rightarrow b.
\end{array}
\end{array}
$$

Note that parametric type classes [4] are necessary here because the implementation of a constructor function will, in general depend both on the datatype itself (in this case, *List*) and on the type of the arguments that it is applied to. These dependencies cannot be expressed using either standard Haskell type classes [7] or constructor classes [11]. Parametric type classes can be implemented using the same dictionary passing style described in Section 3, and hence the specialization techniques presented in this paper can be used to ensure that we do not incur any run-time costs from the use of overloading.

With this framework in mind, the definition of the *map* function in terms of *caseList* is unchanged, but the type of *map*, at least for the compiler's purposes becomes:

$$
\begin{array}{l}
\textit{map} :: (\textit{la} :: \textit{List } a, \textit{lb} :: \textit{List } b) \Rightarrow (a \rightarrow b) \rightarrow (\textit{la} \rightarrow \textit{lb}) \\
\textit{map } f \; ys \\
\quad = \textit{caseList } ys \; \textit{Nil} \; (\backslash x \; xs \rightarrow \textit{Cons } (f \; x) \; (\textit{map } f \; xs))
\end{array}
$$

Note that this type reflects the fact that the representation for the two lists involved in this function. The implementation of *caseList* used in the definition depends on the representation of *la*, while the interpretation of *Cons* and *Nil* will depend on *lb*.

As we have already indicated, we would expect the use of overloading to deal with specialized representations to be largely hidden from the programmer. The information that would normally be obtained from class and instance declarations can be generated automatically within the compiler and the whole system can make use of many of the mechanisms already included in the compilation system for supporting the use of type classes.

6.3 Another chance for code explosion?

Some experimental work is required to assess whether code explosion problems are likely when specialized representations are used. Almost every function in a typical program makes use of some simple kind of data structure, so some degree of code explosion seems very likely.

Although we are not yet in a position either to confirm or refute this behaviour, we have carried out some preliminary tests to investigate the kind of code explosion that would occur in the most extreme case when every distinct monomorphic instance of a polymorphic function used in a given program is implemented by a distinct section of code. The results for some of the larger benchmarks are shown in Figure 4. These figures show the number of distinct binding

Program	poly	mono	ratio	lists	pairs
`anna`	1,653	3,207	1.94	153	136
`veritas`	1,128	1,646	1.46	44	47
`infer`	186	434	2.33	14	25
`prolog`	138	255	1.85	23	25
`expert`	146	239	1.64	14	8

Figure 4: Code explosion with full monomorphization

groups that are actually used in the original program (poly) and the number of monomorphic binding groups that would be required in a fully specialized version of the same program (mono). Note that these figures do not take account of the use of dictionaries or of the introduction of new binding groups during compilation, for example, in the implementation of list comprehensions. The figures suggest a typical twofold increase in the number of binding groups but it is not clear how this will relate to code size. For example, the most commonly used function in `anna` is *map* with 187 distinct instances. But the code for this function is fairly small and some optimizing compilers might normally choose to expand its definition inline so that there will actually be very little noticeable change in code size for this example. The number of instances of other functions in the program are substantially lower than this.

The most frequently used data structures in all of these programs are lists and pairs and we have included the number of distinct instances of each in the table above. The `infer` program is a small exception; two of the monads defined in this program have 15 and 18 distinct uses respectively in comparison to the 14 instances of lists.

Further work is needed before we can judge whether the techniques for using specialized representations described here will be useful in practical systems. In the meantime, it is worth mentioning that, to the best of our knowledge, there does not seem to be any other work studying of the extent to which polymorphism is actually used in real programs. This kind of information would be useful in its own right, for example, helping to identify where program optimization is likely to have the greatest impact.

7 Further work

Haskell type classes provide a useful extension to a language with a polymorphic type system but the dictionary-passing style used in all current Haskell systems can incur substantial overheads. Expanding the definitions of all overloaded functions in a given program to avoid the problems caused by the use of dictionaries can, in theory, result in an exponential increase in the size of the program code. However, our experience with an implementation of type classes based on this approach suggests very strongly that this does not occur in realistic programs.

The biggest outstanding problem with the work described here is its interaction with separate compilation. In many systems, the source code for a program may be spread over several modules. Each program module can be compiled separately. The standard approach is to parse, validate and type check each module and compile it to some appropriate intermediate language. This is then passed to a code generator to obtain an object code file. Once the object code files for all of the source modules in a given program have been produced, they can be passed to a linker that resolves inter-module references and generates the executable program. The complete process is illustrated in Figure 5.

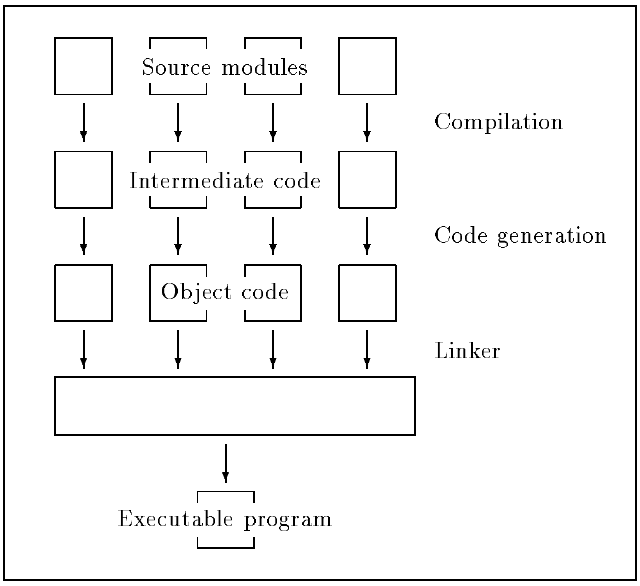


Figure 5: Standard approach to separate compilation

One of the most important benefits of this approach is that, if we make a change to the source code, only those modules that are affected must be recompiled. The full set of object files is required to build the final executable program, but the cost of linking is usually fairly small. A second benefit is that it is not necessary to have all of the source code available when the program is linked; only the object code is required.

To make use of program specialization, we need to postpone code generation as illustrated in Figure 6. This requires a more sophisticated linker that works at a higher level – with the intermediate language rather than the object code. The main problem now is that, when source code changes, the high-level linker will produce a new program to be specialized and the complete specialized program will be passed through the code generator. As we have already mentioned in Section 4.3, code generation in our current implementation is by far the most time-consuming part of the compilation process. The result is that the cost of making even a simple change to the source code is very high.

There are a number of ways that this might be avoided; for example, by adopting some of the techniques used to support incremental compilation or further ideas from partial evaluation. Another interesting possibility would be to postpone some code generation until even later than suggested by Figure 6, so that specialized versions of some parts of the program could be generated dynamically at run-time. Similar techniques have been used with considerable success in the implementation of the object-oriented language Self [3].

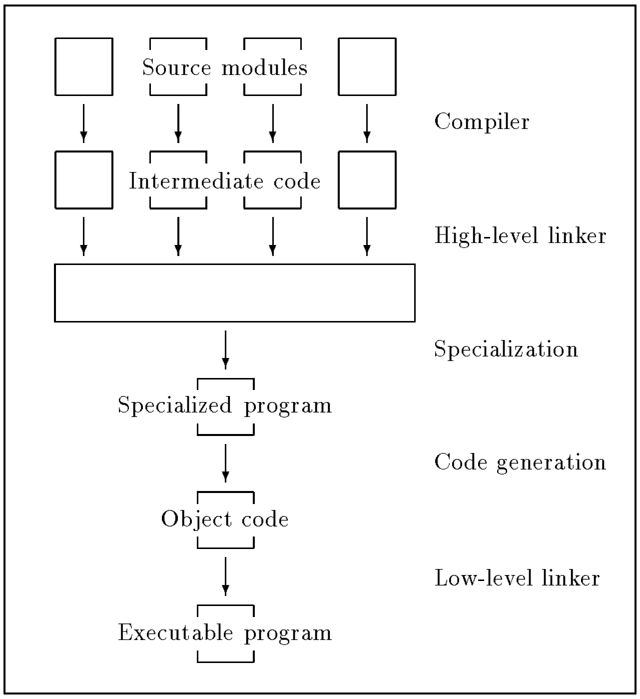


Figure 6: Separate compilation with specialization

We should also point out that, despite the problems associated with dictionary passing, the level of performance provided by current Haskell systems is already good enough for a lot of program development work. We might therefore expect to use dictionary-based implementations during the prototyping and development stages, viewing specialization more as a way of improving the performance of the final product.

The current module system in Haskell has been criticized as one of the weakest parts of the language and there have been suggestions that future versions of Haskell might adopt a more powerful system. With the comments of this paper in mind, one of the factors that should strongly influence the choice of any replacement is the degree to which it supports optimization, analysis and specialization across module boundaries.

Acknowledgments

This work was supported in part by grants from DARPA, contract number N00014-91-J-4043, and from NSF, contract number CCR-9104987. Thanks to Martin Odersky and Kung Chen for their comments on this work, to Julian Seward for encouraging me to take my original experiments a little further and for providing me with my biggest benchmark, anna. Thanks also to Paul Hudak for getting me thinking about the relationship between datatype definitions and type classes.

References

[1] A. Appel. *Compiling with continuations.* Cambridge University Press, 1992.

[2] L. Augustsson. Implementing Haskell overloading. To appear in *Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993.

[3] C. Chambers, D. Ungar and E. Lee. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. *Lisp and symbolic computation*, 4, 3, 1991.

[4] K. Chen, P. Hudak, and M. Odersky. Parametric type classes (Extended abstract). *ACM conference on LISP and Functional Programming*, San Francisco, California, June 1992.

[5] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1, 1, January 1991.

[6] P. Hudak and J. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN notices*, 27, 5, May 1992.

[7] P. Hudak, S.L. Peyton Jones and P. Wadler (eds.). Report on the programming language Haskell, version 1.2. *ACM SIGPLAN notices*, 27, 5, May 1992.

[8] M.P. Jones. A theory of qualified types. In *European symposium on programming.* Springer Verlag LNCS 582, 1992.

[9] M.P. Jones. Qualified types: Theory and Practice. D. Phil. Thesis. Programming Research Group, Oxford University Computing Laboratory. July 1992.

[10] M.P. Jones. Computing with lattices: An application of type classes. *Journal of Functional Programming*, Volume 2, Part 4, October 1992.

[11] M.P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993.

[12] X. Leroy. Efficient data representation in polymorphic languages. INRIA research report 1264, July 1990.

[13] X. Leroy. Unboxed objects and polymorphic typing. In *ACM Principles of Programming Languages*, New York, January 1992.

[14] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 3, 1978.

[15] R. Morrison, A. Dearle, R.C.H. Connor and A.L. Brown. An ad-hoc approach to the implementation of polymorphism. *ACM Transactions on Programming Languages and Systems*, 13, 3, July 1991.

[16] A. Ohori. A simple semantics for ML polymorphism. In *4th International Conference on Functional Programming Languages and Computer Architecture*, Imperial College, London, September 1989. ACM Press.

[17] J. Peterson and M. Jones. Implementing Type Classes. *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, June 1993.

[18] S.L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture*, Cambridge, MA, Springer Verlag LNCS 582, August 1991.

[19] S.L. Peyton Jones and P. Wadler. A static semantics for Haskell (draft). Manuscript, Department of Computing Science, University of Glasgow, February 1992.

[20] S. Thatte. Typechecking with ad hoc polymorphism (preliminary report). Manuscript, Department of mathematics and computer science, Clarkson University, Potsdam, NY. May 1992.

[21] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *16th ACM annual symposium on Principles of Programming Languages*, Austin, Texas, January 1989.