Dictionary-free Overloading by Partial Evaluation

Mark P. Jones
Yale University, Department of Computer Science,
P.O. Box 208285, New Haven, CT 06520-8285.
jones-mark@cs.yale.edu

Abstract

One of the most novel features in the functional programming language Haskell is the system of *type classes* used to support a combination of overloading and polymorphism. Current implementations of type class overloading are based on the use of *dictionary* values, passed as extra parameters to overloaded functions. Unfortunately, this can have a significant effect on run-time performance, for example, by reducing the effectiveness of important program analyses and optimizations.

This paper describes how a simple partial evaluator can be used to avoid the need for dictionary values at run-time by generating specialized versions of overloaded functions. This eliminates the run-time costs of overloading. Furthermore, and somewhat surprisingly given the presence of multiple versions of some functions, for all of the examples that we have tried so far, specialization actually leads to a reduction in the size of compiled programs.

1 Introduction

The functional programming language Haskell [9] supports a flexible combination of overloading and polymorphism based on the use of *type classes* [20]. The standard implementation technique, adopted in all of the Haskell systems to date, involves the use of *dictionary* values that are passed as additional parameters to overloaded functions to resolve the use of overloading at run-time. Unfortunately, it is very difficult to obtain an efficient implementation using this approach because of the overheads of manipulating dictionaries at run-time and because their presence reduces the effectiveness of important program analyses and optimizations.

This paper describes a compiler for a small Haskell-like language that uses a partial evaluator to eliminate run-time dictionaries. Instead of dictionaries, we generate specialized versions of overloaded functions at compile-time, completely avoiding the costs of run-time overloading. While the potential for dictionary-free overloading has been discussed in the past, the idea has not been adopted in practical systems for fear that it could lead to a substantial increase in the size of compiled programs—a so-called *code explosion*. In fact, for all of the programs we have tried so far, we find that the partial evaluator actually gives a reduction in program size! The main reason for this seems to be that the use of partial evaluation allows us to do a better job identifying redundant sections of code that will not be needed at run-time.

Our system fits naturally into the framework of offline partial evaluation with the type class type inference algorithm providing a simple binding time analysis. The results of this paper therefore provide further motivation for including a more general partial evaluation system as part of production quality compilers for languages like Haskell.

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 can be defined and extended to work over a range of datatypes. The dictionary-passing implementation techniques used in all current Haskell implementations are presented in Section 3, with comments outlining some of the obstacles to an efficient implementation. 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. Finally, Section 5 gives some pointers to further and related 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 [20] 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 [8, 9].

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:

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

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:

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{instance}\ Num\ Integer\ \textbf{where}} \\
\quad (+) & = & primPlusInteger \\
 & \vdots & \\
\quad fromInteger\ x & = & x
\end{array}$$

This assumes that the implementation provides a built-in function $primPlusInteger$ for adding $Integer$ 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:

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

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{instance}\ Num\ a \Rightarrow Num\ (Complex\ a)\ \textbf{where}} \\
\quad (x :\!+ y) + (u :\!+ v) & = & (x + u) :\!+ (y + v) \\
 & \vdots & \\
\quad fromInteger\ x & = & fromInteger\ x :\!+ fromInteger\ 0
\end{array}$$

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:

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

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:

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

As a result, the $fact$ function has type $Num\ a \Rightarrow a \rightarrow a$ 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:

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

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 any 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 values, 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.

Another approach – based on the use of *dictionary* values – was introduced by Wadler and Blott [20] and has been adopted in all current Haskell systems. A dictionary is a tuple that contains the implementations for each of the member functions in a given class. Superclasses are represented by pointers to corresponding subdictionaries. 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 $dictNumInteger$

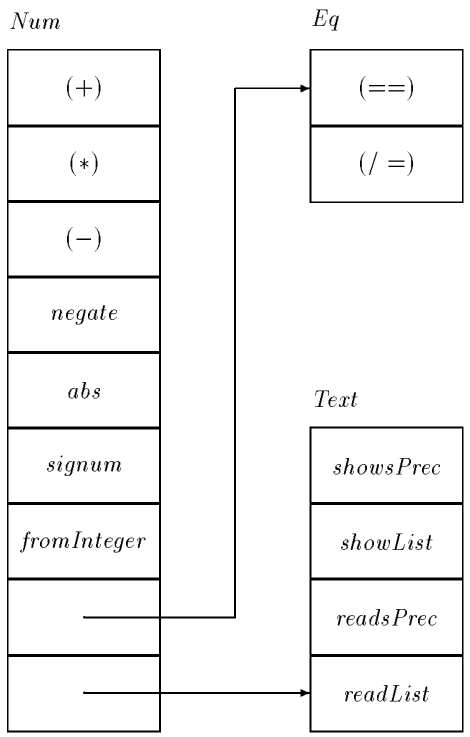

Figure 1: Dictionary structure for the *Num* class

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 as selectors, we obtain:

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

This example uses a function *eqOfNum* to extract the superclass dictionary for *Eq* from the dictionary for the corresponding instance of *Num*. Further details about the translation process are are given by [1, 17, 19, 20].

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.

The following sections discuss each of these points in a little more detail.

3.1 Dictionaries increase program size

In an attempt to reduce the size of executable programs, many compilers 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. However, we will see that the use of dictionaries can reduce the benefits of tree shaking.

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:

$$\mathit{pyth}\ x\ y\ z \quad = \quad x*x+y*y == z*z$$

might be assigned the type:

$$(\mathit{Eq}\ a, \mathit{Plus}\ a, \mathit{Mult}\ a) \Rightarrow a \to a \to a \to \mathit{Bool}$$

rather than the simpler:

$$\mathit{Num}\ a \Rightarrow a \to a \to a \to \mathit{Bool}.$$

A disadvantage of grouping together methods like this is that it becomes rather more difficult to identify unnecessary sections of code. 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 if a program uses only *Int* arithmetic, the need to include a *fromInteger* function as part of the *Num Int* 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]:

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

After the translation to include dictionary parameters this becomes:

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

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 [10] for further details).

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

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:

$$\mathit{fact}\ \ ::\ \ \mathit{Int} \to \mathit{Int}.$$

The translation of *fact* can then be simplified to:

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

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 can sometimes give 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.

Of course, there would be no need for overly restrictive type signatures if the programmer could rely on the compiler to generate efficient, type specific versions of overloaded functions.

4 A dictionary-free implementation

From the programmer's perspective, Haskell type classes have proved to be a valuable extension of the Hindley-Milner type system used in languages like ML. The standard prelude for Haskell included in [9] 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 [20], together with the observation that this might result in an exponential growth in code size. This was illustrated by considering the function:

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

which has type:

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

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.

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 fairly modest increases. To our surprise, we found 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 [10] 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 $x = \lambda v.M$ where v denotes a possibly empty sequence of dictionary parameters) in which no variable x has more than one binding. The set of variables x bound in B will be written $dom\ B$. 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 [10] allows only single bindings in local definitions; of course, an expression of the form $\textbf{let}\ x = M\ \textbf{in}\ M'$ in that system can be represented as $\textbf{let}\ \{x = M\}\ \textbf{in}\ M'$ 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 $f\ d \leadsto f'$ 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:

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

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:

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

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.

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

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

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

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

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

Figure 2: Specialization algorithm

Note that there are two rules for dealing with variables. The first, ($\mathit{var\text{-}let}$), 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, ($\mathit{var\text{-}\lambda}$), 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.

The hypothesis $e = d$ in the rule ($\mathit{var\text{-}let}$) implies 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:

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

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 $S', S' \vdash B \leadsto B'$.

Remember that the goal of program specialization is to produce a dictionary-free implementation of any 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:

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

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 [10] 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:

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

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

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

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

4.1.4 A simple example

To illustrate the way that the specialization algorithm works, consider the simple Haskell expression:

$$\mathbf{let}\ f\ x = x + x\ \mathbf{in}\ f\ \mathit{one}.$$

After type checking and the insertion of dictionary parameters, this becomes:

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

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:

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

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 specialized version of the original term is:

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

4.1.5 The treatment of member functions

Specializations involving member functions can be handled a little more efficiently than other overloaded functions. In particular, given a specialization $(m\ d \leadsto x')$ 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. 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 [6]. 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:

$$\begin{array}{ll}\text{standard} \\ \text{operations}\end{array} \left\{\begin{array}{ccc}\lambda x.M & \sim & \underline{\lambda} x.M \\ M\ N & \sim & M \_ N\end{array}\right\} \text{delayed}$$

$$\begin{array}{ll}\text{dictionary} \\ \text{operations}\end{array} \left\{\begin{array}{ccc}\lambda v.M & \sim & \lambda v.M \\ M\ e & \sim & M\ e\end{array}\right\} \begin{array}{l}\text{eliminated by} \\ \text{specialization}\end{array}$$

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 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 $n \leadsto m$ indicates that, 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.

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

Figure 3: Code size indicators

- `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 $D_{n+1} = D_n \to D_n$ as described in [11]. 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.

Using Gofer, the run-time performance of our programs is improved only marginally by the use of partial evaluation. This is because the Gofer code generator is very simple and does not carry out any of the optimizations described in Sections 3.2, 3.3, and 3.4. We have already argued that these optimizations have the potential to offer significant improvements in run-time performance for specialized code, a claim which is supported by measurements described in [1]. One obvious way to gauge this potential would be to use the specialized programs produced by Gofer as the input to a more sophisticated compiler. Unfortunately, for technical reasons quite unrelated to specialization, this is not possible with the current implementation. Instead, as a very crude indicator, we have run some simple experiments using the following program which makes use of two of the functions described above:

$$
\begin{array}{lll}
\mathit{fact} & :: & \mathit{Num}\ a \Rightarrow a \rightarrow a \\
\mathit{fact}\ n & = & \textbf{if}\ n == 0\ \textbf{then}\ 1 \\
 & & \qquad\qquad \textbf{else}\ n * \mathit{fact}\ (n-1) \\
 & & \\
\mathit{nums} & :: & (\mathit{Ord}\ a,\ \mathit{Num}\ a) \Rightarrow a \rightarrow [a] \\
\mathit{nums}\ n & = & \textbf{if}\ n < 1000\ \textbf{then}\ n : \mathit{nums}\ (n+1) \\
 & & \qquad\qquad \textbf{else}\ [\,] \\
 & & \\
\mathit{sum} & :: & \mathit{Num}\ a \Rightarrow [a] \rightarrow a \\
\mathit{sum}\ xs & = & \mathit{loop}\ 0\ xs \\
\end{array}
$$

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

$$
\begin{array}{lll}
\mathit{main} & :: & \mathit{Dialogue} \\
\mathit{main} & = & \mathit{print}\ (\mathit{sum}\ (\mathit{map}\ \mathit{fact}\ (\mathit{nums}\ (1{::}\mathit{Int}))))
\end{array}
$$

Using the Chalmers Haskell B. compiler on a Sun SparcServer 690MP, this program runs in 13.6 user seconds and allocates approximately 43MB on the heap. By contrast, a hand-specialized version of the same program takes only 2.1 user seconds, less than one sixth of the original time, and allocates 9MB on the heap. This program makes particularly heavy use of overloaded functions, and we would be unlikely to obtain speedup factors as good as this for more typical applications programs. Nevertheless, these example does help both to illuminate some of the performance problems with current implementations of overloading and to demonstrate the potential benefits of specialization.

5 Further and related 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.

We believe that this work demonstrates a successful application of partial evaluation. In a full system, we would expect to obtain further benefits by using a more general binding-time analysis to augment the information produced by distinguishing between dictionary and standard abstractions and applications.

The idea of producing specialized versions of functions as an implementation of polymorphism or overloading is by no means new (see [5, 16], for example) but has not been widely adopted in practical systems. The particular case of specializing to avoid the need for dictionaries in Haskell style overloading was considered in [10] but the techniques proposed there were somewhat ad-hoc, and intertwined with the type inference algorithm. Motivated by the costs of dictionary manipulation, some implementations of Haskell [1] allow the programmer to request specialized versions of specific overloaded functions using source code annotations. Unfortunately, dictionaries are still required in many cases and there is no guarantee that specialized versions of functions will always be used whenever they are available.

The work described in this paper is closely related to previous work using specialization techniques to improve the performance of method dispatch in object-oriented languages [2, 13, 21]. In fact, although the setting is a little different, the dictionary based implementation of overloading can be considered as a particular form of method dispatch. In this respect, the most interesting aspect of the work described here is the surprising effect of specialization on program size. However, we are unlikely to see the same effect in object-oriented dynamic languages since it is more difficult to predict the form of an object at compile-time, and hence to eliminate redundant sections of code.

There are two particular areas where further work would be desirable; the treatment of separate compilation and the extension of our techniques to the specialization of polymorphism.

5.1 Interaction with separate compilation

The biggest outstanding problem with the work described here is its interaction with separate compilation. For example, we need to deal with cases where the need for a specialized version of a function may not be known when the module containing its definition is compiled. Our prototype implementation avoids these problems by requiring all of the source modules in a program to be supplied to the compiler at the same time time. Clearly, this would not be acceptable for large scale program development.

The same problems occur in any system where partial evaluation or program optimization across module boundaries is required. There has already been some work in this area; for example, Consel and Jouvelot [4] give an algorithm that can be used to avoid repeated computation of binding time information for frequently used libraries of functions. However, much remains to be done.

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.

5.2 Specialization of polymorphism

The results presented in this paper suggest quite strongly that we can expect to obtain a viable and efficient implementation of overloading using program specialization. As a more radical proposal, we might question whether it is possible to obtain a realistic implementation of ML-style polymorphism in a similar manner, producing specialized monomorphic versions of polymorphic functions as necessary for a given program.

Once again, the biggest problems in such a system are likely to be the risk of code explosion and the interaction with separate compilation. However, we can also expect some significant benefits. For example, many implementations of ML-style polymorphism require the arguments of polymorphic functions to be packaged as *boxed* values—a uniform representation that is independent of the type of values involved. Converting between boxed and unboxed representations can be quite expensive and there have been a number of attempts to find techniques for reducing or eliminating unnecessary conversions [14, 18, 15, 7]. The alternative suggested here is to reduce polymorphic definitions to a collection of monomorphic versions, each of which can be implemented using an appropriate representation for the values that it manipulates. This will probably be most effective in languages with strict semantics where there is no need to distinguish between values and delayed computations. Some preliminary experiments in this direction are reported in [12], and further investigation is already in progress.

Acknowledgments

This work was supported in part by a grant from ARPA, contract number N00014-91-J-4043. Thanks also to Martin Odersky, Kung Chen, John Peterson, Paul Hudak, and the referees for their comments on an earlier version of this work and to Julian Seward for encouraging me to take my original experiments a little further and for providing me with my biggest benchmark, `anna`.

References

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

[2] C. Chambers and D. Ungar. Customization: optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. *Programming Language Design and Implementation*, Portland, Oregon, ACM SIGPLAN notices, volume 24, number 7, July 1989.

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

[4] C. Consel and P. Jouvelot. Separate polyvariant binding-time analysis. Oregon Graduate Institute, Department of Computer Science, Technical report CS/E 93-006, March 1993.

[5] D. Gries and N. Gehani. Some ideas on data types in high-level languages. Communications of the ACM, Volume 20, Number 6, June 1977.

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

[7] F. Henglein and J. Jørgensen. Formally optimal boxing. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994.

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

[9] 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.

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

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

[12] M.P. Jones. Partial evaluation for Dictionary-free overloading. Yale University, Department of Computer Science, Research Report YALEU/DCS/RR-959. April 1993.

[13] S.C. Khoo and R.S. Sundaresh. Compiling inheritance using Partial Evaluation. Yale University, Department of Computer Science, Research Report YALEU/DCS/RR-836. December 1990.

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

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

[16] 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.

[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] 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.

[21] D. Weise and S. Seligman. Accelerating object-oriented simulation via automatic program specialization. Department of Electrical Engineering and Computer Science, Stanford University, Technical Report CSL-TR-92-519, April 1992.