Implementing Haskell overloading

Lennart Augustsson
Department of Computer Sciences
Chalmers University of Technology
S-412 96 Göteborg, Sweden
Email: augustss@cs.chalmers.se

Abstract

Haskell overloading poses new challenges for compiler writers. Until recently there have been no implementations of it which have had acceptable performance; users have been adviced to avoid it by using explicit type signatures. This is unfortunate since it does not promote the reusability of software components that overloading really offers.

In this paper we describe a number of ways to improve the speed of Haskell overloading. None of the techniques described here is particularly exciting or complicated, but taken together they may give an order of magnitude speedup for some programs using overloading.

The techniques fall into two categories: speeding up overloading, and avoiding overloading altogether. For the second kind we borrow some techniques from partial evaluation.

There does not seem to be a single implementation technique which is a panacea; but a number of different ones have to be put together to get decent performance.

1 Introduction

Haskell, [Hud92], introduces a new and systematic way of handling overloading through the use of type classes. Haskell style type classes where introduced by [Kae88] and got their current shape in [WB89]. An implementation technique is also suggested in the latter paper where each overloaded function is translated into an equivalent non-overloaded function with an extra argument, the *dictionary*, containing all the operations of the class. Using this translation will result in a program with a lot of higher order functions. Higher order functions are notorious for being hard to handle efficiently because much less is known about them at compile time. This is the main reason for Haskell overloading being slow.

In this paper we will investigate a number of techniques for improving the performance of overloading in Haskell. Each of these techniques are fairly straightforward, but taken together they can have a significant impact on efficiency. The scenario we have in mind for these techniques is a system where modules are compiled separately, and with information possibly flowing "forwards", i.e., if module B depends on module A then information about A may be available when B is compiled, but not vice versa. For a system where the complete program is available at compile time (or with information flow in all directions) we could do much better, and we discuss this briefly at the end of the paper.

Most of the techniques described here have been implemented in the Chalmers HBC Haskell compiler, [AJ89, Aug93].

We will start by a brief recapitulation of Haskell overloading and how it is translated by program transformation. We will then continue by examining this translation and suggesting improvements of it. We continue by discussing how the overloading overhead can be eliminated completely by specializing the program. We finish the paper by some measurements and suggestions of other possible techniques.

2 Type classes

A type class is set of types sharing some operations, called methods. A new type class is introduced by a class declaration which describes what the common operations are (it may also specify a number of superclasses). A type is declared to be in a class through an instance declaration. The instance declaration describes what the methods given in the class declaration do for the particular type.

Figure 1 contains an example that declares the class `Eq` with the two methods `==` and `/=` intended to compare for equality and inequality. The type `Char` and lists are then declared to be instances of this class. The example also contains a function, **`mem`**, which uses the (overloaded) method `==`. The type of this function indicates that it may be applied to any value which has a type that is in the `Eq` class.

3 Simple translation

The full language with overloading is translated into a subset of the language where all overloading has been removed. This translation is usually integrated into the type checking. It was suggested in [WB89] and further explicated in [HHPW92]. We will make a short recapitulation of it here.

Each class declaration C introduces a number of methods m_k. Methods may be given a default definition which is used if an instance declaration does not give a definition for the particular method. Each instance declaration gives rise to a *dictionary* (a tuple) containing all the methods in the class. To select a particular method from the dictionary there are selection functions. The selection function for method m_k will be named m_k in the translated program. The default

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)
instance Eq Char where
    (==) = primEqChar
instance Eq a => Eq [a] where
    []     == []     = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _      == _      = False

mem :: (Eq a) => [a] -> a -> Bool
mem [] y = False
mem (x:xs) y = x == y || mem xs y

val = mem [] 'a'
```

Figure 1: Sample class and instance declaration.

methods will be named $C.m_k$. Note that we will use names like $a.b.c$ where the "." is just part of the name; it has no special significance. The parts between the dots may be identifiers or operators enclosed in "()". This means that "`Eq.Char.(==)`" should be read as a complete symbol; it is just a name.[1]

An instance declaration of class C and type T will translate into two parts: definitions of all the class methods (named $C.T.m_k$) and a dictionary (named $C.T$). The dictionary is a tuple containing all the methods from the instance, as well as the dictionaries for the immediate superclasses of C. All of this is illustrated in figures 1 and 2. If the instance declaration has a context (as is the case for "Eq" on lists in figure 1) the resulting dictionary will itself be "overloaded". To get a real dictionary (a tuple) it has to be applied to dictionaries corresponding to the context.

Each use of an overloaded identifier will be replaced by the selection function applied to an appropriate dictionary. Each overloaded functions will get an additional argument, a dictionary, for each overloading constraint.

The naming scheme makes it possible to determine the name for a particular method or dictionary definition even it is compiled in a separate module; all one need to know is the class, the type, and the method.

The translated program is not quite a Haskell program since it may not be type correct,[2] but this does not matter for the rest of the compilation process.

4 Faster method calls

We will first focus our attention on trying to improve method calls through a dictionary, and later see how we can sometimes avoid them altogether.

[1] We disregard the silly Haskell restrictions that non-constructors must begin with a lower case letter

[2] Consider the following example (due to Konstantin Läufer):

```
class C a where
   f ::  b -> (a,b)
instance C Int where
   f x = (1,x)
h x = (f 'a', f True)
```

The translated version of h will become

```
h dictC x = (f dictC 'a', f dictC True)
```

This is ill-typed in the Hindley-Milner sense, since dictC is λ-bound but used with two different types

```
(==) (m, _) = m
(/=) (_, m) = m
Eq.(==) dictEq x y = error "no =="
Eq.(/=) dictEq x y = not ((==) dictEq x y)

Eq.Char.(==) = primEqChar
Eq.Char.(/=) = Eq.(/=) Eq.Char
Eq.Char      = (Eq.Char.(==), Eq.Char.(/=))

Eq.List.(==) dictEq []     []     = True
Eq.List.(==) dictEq (x:xs) (y:ys) =
    (==) dictEq x y && Eq.List.(==) dictEq xs ys
Eq.List.(==) dictEq _      _      = False
Eq.List.(/=) dictEq xs     ys     =
    Eq.(/=) (Eq.List dictEq) xs ys
Eq.List dictEq =
    (Eq.List.(==) dictEq, Eq.List.(/=) dictEq)

mem dictEq [] y = False
mem dictEq (x:xs) y =
    (==) dictEq x y || mem dictEq xs y

val = mem Eq.Char [] 'a'
```

Figure 2: Translated sample class and instance declaration.

4.1 Dictionary representation

Representing dictionaries as tuples and having a selector function pick out a component has some drawbacks. Consider the translation of equality on list (figure 2):

```
Eq.List dictEq =
    (Eq.List.(==) dictEq, Eq.List.(/=) dictEq)
```

A typical use of this dictionary would be translating "`[x] == [y]`" into

```
(==) (Eq.List dict) x y
```

Here we notice that the entire dictionary is constructed (exactly how much depends on the compiler in question) and then one component is selected and the others are thrown away. This is clearly useless work and we would like to avoid it.

An alternative is to represent a dictionary as a pair consisting of a vector of functions and a vector of dictionaries. To call the mth method the mth element in the vector of functions is called and as an extra argument it will get the vector of dictionaries. The second vector corresponds to the dictionary arguments needed to construct the actual dictionary. See figure 3 for some examples. The pair could be thought of as a partial application of the overloaded dictionary (the vector of functions) to its dictionary arguments (the vector of dictionaries). Using an explicit representation instead of a partial application allows the components to be accessed in a way that allows faster calls. With this representation a method call costs, with our G-machine ([Joh87]) implementation, about 15 memory references[3] which should

[3] For todays fast RISC machines the bulk of the time is spend doing memory references

```
(==) (f,d) = f 0 d
(/=) (f,d) = f 1 d
Eq.(==) dictEq x y = error "no =="
Eq.(/=) dictEq x y = not ((==) dictEq x y)

Eq.Char.(==) = primEqChar
Eq.Char.(/=) = Eq.(/=) Eq.Char
VEC.Eq.Char k= case k of { 0 -> \() -> Eq.Char.(==);
                           1 -> \() -> Eq.Char.(/=) }

Eq.List.(==) dictEq []     []     = True
Eq.List.(==) dictEq (x:xs) (y:ys) =
    (==) dictEq x y && Eq.List.(==) dictEq xs ys
Eq.List.(==) dictEq _      _      = False
Eq.List.(/=) dictEq xs     ys     =
    Eq.(/=) (VEC.Eq.List, (dictEq)) xs ys
VEC.Eq.List k =
    case k of { 0 -> \(dictEq) -> Eq.List.(==) dictEq;
                1 -> \(dictEq) -> Eq.List.(/=) dictEq }

mem dictEq [] y = False
mem dictEq (x:xs) y =
    (==) dictEq x y || mem dictEq xs y

val = mem (VEC.Eq.Char,()) [] 'a'
```

Figure 4: Alternative translated sample class and instance declaration.

be compared to 6 references for calling a known function. Figure 4 gives the translation using these alternative dictionaries. In this translation the first vector is represented by a function (with the index as argument), and the second as a tuple. It is, of course, possible to devise a much better low level representation. The method selector functions ((==) and (/=)) are just shown for clarity; in a real translation they would be unfolded.

Together with the next transformation the tail calls inside the case expression can be made quite efficient.

A more radical approach for avoiding construction of dictionaries is taken in [Jon]. With this approach all expressions yielding a dictionary as the result (e.g. "Eq.List Eq.Char") will have constants components, and can thus be computed at compile time instead of at run time. Unfortunately this approach requires a generalization of the Haskell type system, so we have not adopted this approach. A function like

```
f x = [x] == [x]
```

would in Jones' system get the type `f :: (Eq [a]) => a -> Bool` instead of `f :: (Eq a) => a -> Bool`.

4.2 Forcing method arity

Calling "unknown" (i.e., an application where the function is not a supercombinator) functions is more costly than calling a "known" function. One reason for this is that with an unknown call there has to be a check that the number of arguments supplied is sufficient. There may also have to be some handling of excess arguments (this is the case for the G-machine, but not the STG-machine [PJS89]). Even if the function is not completely known, knowing the arity of it improves efficiency.[4]

[4] At least with all implementation methods known to me.

A method call is a call to a completely unknown function, and thus quite costly. But the compiler does know the type of the function, and the type suggest a "natural" arity of the function, namely the number of top level arrows in the type. The arity of a method can be known by enforcing that all actual methods in the instance declarations have their natural arity. Doing so is quite easy, it just involves a possible η-expansion of the method definition.

This is a transformation that may loose full lazyness, but is beneficial in general. (Full lazyness is not guaranteed by the HBC compiler anyway.)

When the arity is known the calls that arise in the method vector (see figure 4) turn into very simple tail calls with the correct number of arguments. These can be implemented by a single jump instruction.

4.3 Numeric literals

Numeric literals are a problem in all programming languages with many numeric types. What is the type of "1"? In Haskell this is solved by saying that "1" really means "`fromInteger 1`", where `fromInteger` is a method for converting from an `Integer` to any numeric type. All numeric literals without a decimal point are really of type `Integer` and gets an automatic conversion by `fromInteger` inserted, and correspondingly all literals with a decimal points are of type `Rational` and gets converted by `fromRational`. This solves the problem, to a large extent, from a programming point of view. But it causes efficiency problems!

Consider the function

```
inc :: (Num a) => a->a
inc x = x + 1
```

and making the automatic conversion explicit

```
inc x = x + fromInteger 1
```

One might think that the cost of computing, say "inc 3.2::Double", is roughly the cost of the addition plus some overhead for the overloading. This is not the case, the biggest cost, by far, is the cost of computing the constant! Converting an `Integer` to a `Double` is quite complicated and takes many instructions. Even worse is `fromRational` which if taken from the Prelude involves many `Integer` multiplications and a division. The matter is made even worse by the fact that the two most common numeric types in a Haskell program are `Int` and `Double` due to the default mechanism for resolving ambiguous overloading. Converting literals to these types is very costly since they are stored as `Integer` and `Rational`.

We clearly need to improve of this if we ever plan to use overloaded literals. Our solution to this is to store preconverted values for the common types. We add a new method `_fromInteger` (hidden from the user) to the `Num` class which takes a triple with the `Integer`, `Int`, and `Double` value of the literal. The new method and the additions to the instance declarations are described in figure 5.

A literal "1" is now translated into "`_fromInteger (1::Integer, 1::Int, 1::Double)`". If this literal is used with types `Int` or `Double` the value is obtained simply by selecting a tuple component, otherwise a conversion will take place (just as before).

An additional advantage of using the `_fromInteger` method is that as the compiler has complete control over all uses of this method it knows that the triple and the parts

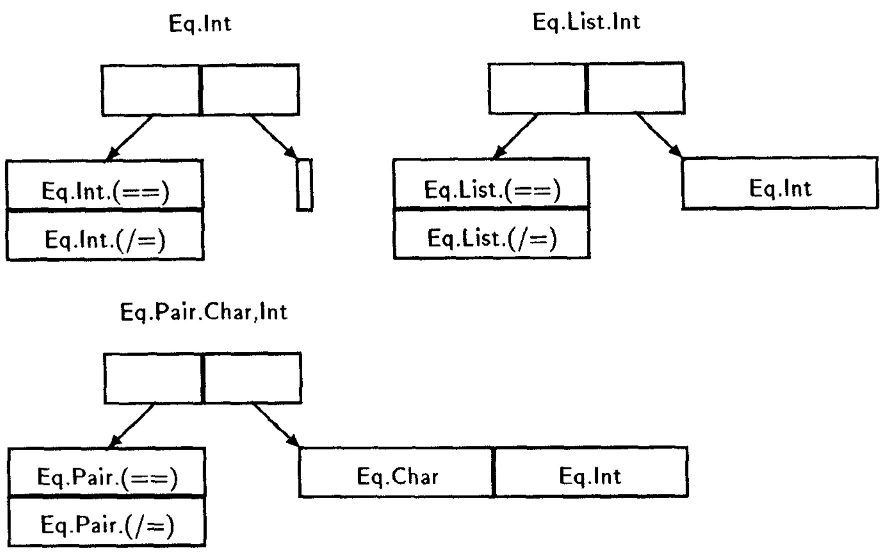

Figure 3: Some sample dictionaries: equality on Int, [Int], and (Char,Int).

```
class Num a where
    ...
    fromInteger :: Integer -> a
    _fromInteger :: (Integer, Int, Double) -> a
    _fromInteger (i,_,_) = fromInteger i
instance Num Int where
    ...
    _fromInteger (_,i,_) = i
instance Num Double where
    ...
    _fromInteger (_,_,d) = d
```

Figure 5: Additions to speed up literals.

of it are all evaluated, so the runtime test for this can be omitted.

The fromRational method is handled in an analogous manner.

5 Avoiding method calls

In this section we will discuss different ways of avoiding the overloading overhead completely.

5.1 Flattening the superclass structure

The suggested translation of classes with superclasses is to have a reference to each superclass dictionary in the subclass dictionary. An example is shown in figure 6.

When a method from a superclass is needed it is selected in the usual way, but the superclass is first accessed via the superclass selector (named like the superclass). An example:

```
f :: (Ord a) => a -> Bool
f x = x == x

f dictOrd x = (==) (Eq dictOrd) x x
```

```
class (Eq a) => Ord a where
    (<), (<=) :: a -> a -> Bool
instance Eq Char where
    (<)  = primLtChar
    (<=) = primLeChar
```

```
(<)  (m, _, _) = m
(<=) (_, m, _) = m
Eq   (_, _, d) = d
Ord.(<)  = error "no <"
Ord.(<=) = error "no <="

Ord.Char.(<)  = primLtChar
Ord.Char.(<=) = primLeChar
Ord.Char.Eq = (Ord.Char.(<), Ord.Char.(<=), Eq.Chai
```

Figure 6: Sample subclass and instance declaration with translation.

```
(<)      (m, _, _, _, _) = m
(<=)     (_, m, _, _, _) = m
Eq       (_, _, d, _, _) = d
Ord.(==) (_, _, _, m, _) = m
Ord.(/=) (_, _, _, _, m) = m

Ord.Char.(<)  = primLtChar
Ord.Char.(<=) = primLeChar
Ord.Char.Eq   = (Ord.Char.(<), Ord.Char.(<=),
                 Eq.Char, Eq.(==), Eq.(/=))

f dictOrd x = Ord.(==) dictOrd x x
```

Figure 7: Sample subclass and instance declaration with translation.

This double method selection can be avoided by flattening the superclass structure. Then all dictionaries contain both the methods of the class and of all its superclasses, an example is shown in figure 7. Note that the superclass dictionary is kept as well as flatting it, because sometimes the whole superclass dictionary is needed and not just one of the methods. Example:

```
g :: (Ord a) => a -> Bool
g x = [x] == [x]

g dictOrd x = (==) (Eq.List (Eq dictOrd)) x x
```

5.2 Constant folding (or Simple partial evaluation)

An obvious transformation is to eliminate the method selection operation when the dictionary is known. This occurs when the type of an expression does not involve any overloading. For example the function

```
inc :: Int -> Int
inc x = x + 1
```

is translated into

```
inc x = (+) Num.Int x (fromInteger Num.Int 1)
```

Since the dictionary Num.Int is known the method selection can be performed at compile time instead of at run time. Note that the contents of the dictionary is known, due to the naming conventions, even if the instance declaration is in a separately compiled module. We simply translate "m_k $C.T$" to "$C.T.m_k$". Doing the method selection in the example above we get

```
inc x = Num.Int.(+) x (Num.Int.fromInteger 1)
```

This will then generate efficient code if the back end of the compiler knows about Num.Int.(+) etc.

This optimization is a very simple instance of partial evaluation (akin to constant folding) and was suggested already in [WB89], and as far as we are aware all Haskell compilers do it.

5.3 Specializing functions

The preceding specialization is all very well when the overloading has been resolved, but what about functions that we

```
sum :: (Num a) => [a] -> a
sum l = sum' l 0
      where sum' []     a = a
            sum' (x:xs) a = sum' xs (a+x)
```

Figure 8: Definition of the Prelude function sum.

want to be overloaded? Take the definition of sum, figure 8.[5] This definition is overloaded and the code for it will not be particularly good. Note that the attempt to make it efficient by making it tail recursive is completely wasted. Since nothing is known about the strictness properties of (+) the evaluation of a+x cannot be made before the tail call, it has to be suspended and it will thus accumulate space until the end of the list is reached.

A very general sum function is nice to have, but we will probably use it mostly for, say, types Int and Double. For these types we would like to have specialized versions. To accommodate this need we have introduced a pragma (annotation) to tell the compiler what instances we would like. For this particular definition we might add the pragma

```
{-# SPECIALIZE sum::[Int]->Int,[Double]->Double #-}
```

Doing so will force the compiler to make to new functions "sum.1::[Int]->Int" and "sum.2::[Double]->Double" with function bodies identical to sum, but with the specified types. The compiler can then produce good code for these since the type is known. To use these special purpose function the compiler has to check each use of an overloaded function to see if it is used with a type for which a specialized version exists.

The information about what specializations a given function has will be noted in the interface file in case of separate compilation.

The function specialization is easily fooled. There is no runtime test at the entry of the non-specialized version of a function to choose the specialized one if the dictionaries passed happens to be the right ones. One could have such a scheme, but it is not clear that it would pay off since this would incur a runtime cost.

5.4 Automatic function specialization (or Real partial evaluation)

Within a module there is no need to add the specialization annotations since the compiler can determine what particular uses each overloaded function has, and make specialized versions for each use. Say that we have the definition of sum as in figure 8 within the same module as the expression

```
sum (l::[Int])
```

This uses sum with the type [Int]->Int and the compiler can then insert a properly specialized function, just as if there had been an annotation.

A different way of solving the same problem is to have a partial evaluator in the compiler. The use of sum will be translated into

```
sum Num.Int l
```

[5]This is not the definition of sum from the Haskell report, but an attempt to improve it

```
f :: (Eq a) => a -> a
f x = if x == x then x else head (f [x])
```

Figure 9: A function with an unbounded number of specialized versions.

```
sum.1   :: [Int] -> Int
sum.1 l = sum' l 0
          where  sum' []     a = a
                 sum' (x:xs) a = sum' xs (a+x)
```

Figure 10: Definition of a specialized sum.

This use of sum has one static argument, `Num.Int`, and one dynamic, `l`. A partial evaluator will specialize **sum** for the static argument, thus giving a function which is the same as the version of **sum** with type `[Int]->Int`. The partial evaluator would have to be polyvariant to capture all possible specializations. Consider the function "`f :: (Ord a, Eq b) => a -> b -> (a,b)`", this may sometimes be used where the type of `a` is known and sometimes where `b` is known. This means that the static arguments may differ for different uses, thus requiring a polyvariant specializer.

As with ordinary partial evaluation there is a potential termination problem with the specialization. There could be programs for which there is no bounded number of different uses of a function. We conjecture that this is not possible with the current Haskell type system, but with a small extension allowing type signatures to be used during type checking it would be possible to have functions like the one in figure 9.[6] A program containing this function would not have a bounded number of instances of `f` since each each call to it gives rise to a potential call with a more complex type.

The automatic function specialization is in fact almost necessary to get reasonable code. If we annotate the **sum** function from figure 8 to get a special version for `Int` we would get the program in figure 10. This definition is hardly an improvement over the original overloaded one at all. The reason is that within the `sum.1` function there is a local *overloaded* definition of `sum'`.[7] But with automatic specialization we would get a special version of it to operate on `Int`.[8]

5.5 Specializing instances

Specializing functions gets us part of the way, but not far enough. The specialized version of **sum** for the type `Complex Double` is almost as bad as before. The reason is the highly generic way in which complex numbers are defined in Haskell, figure 11 gives a small part. The definition of the method `(+)` on complex numbers has absolutely no knowledge about the actual type of the components except that they must belong to `RealFloat`. The translated version of

```
data (RealFloat a) => Complex a = a :+ a

instance (RealFloat a) => Num (Complex a)  where
    (x:+y) + (x':+y') = (x+x') :+ (y+y')
```

Figure 11: Part of the definition of **Complex**.

the instance declaration gives us the following function for `(+)`:

```
Num.Complex.(+) dictRF (x:+y) (x':+y') =
     ((+) dictRF x x') :+ ((+) dictRF y y')
```

Since the `(+)` method for complex numbers uses the `(+)` method for the underlying numbers the translated method gets overloaded itself.

Again, we would like to specialize this for particular uses of the complex addition. Within a module a partial evaluator would be able to handle this perfectly, but again with separate compilation we can do nothing automatically. But just as for functions we can add specialization annotations:

```
{-# SPECIALIZE instance Num (Complex Double) #-}
```

The meaning of the pragma is the same as an instance declaration of the kind

```
instance Num (Complex Double) where
    ...
```

followed by exactly the same code as the original instance declaration. Haskell does not allow instance declarations of this kind (where `Double` occurs there has to be a type variable), but the type checker and translation mechanism can be easily changed to allow it. The type checker/translator will translate the program to use the most specific instance where several instance declarations apply.

Just as with specialized functions this scheme is easily fooled since there is no runtime test.

Even if we specialize the instance declaration for complex numbers the function to sum a list of them will still be bad, but more about this in section 6.

5.6 Inlining

Inlining (or unfolding) function calls is a good method to improve performance of programs built from many small functions, as functional programs usually are. This seems to be even more the case for method definitions. Almost all method definition in the standard Haskell Prelude are quite small and would benefit from being inlined. This is even more the case for the default methods given in the class declarations, since they almost always involve overloaded methods calls.

Inlining is not without its problems. Recursive methods have to be handled properly (either by unfolding a limited number of times or using fold/unfold transformations). There are also some very pragmatic issues such as what to do when a function to be inlined is defined in another module, but calls functions not exported from that module.

[6] This kind of function is already possible to write using mutually recursive modules

[7] This kind of oversight is quite easy to make, I've seen it in several programs that otherwise had type signatures everywhere.

[8] And we then count on the dead code elimination to remove the unused original version of the function

```
(x:+y) + (x':+y') = (x+x') :+ (y+y')

addComplexDouble (x:+y) (x':+y') =
    (Num.Double.(+) x x') :+ (Num.Double.(+) y y')
```

Figure 12: Complex addition, original and specialized for Double.

6 Haskell Complex is a pain

This section is unrelated to overloading but discusses a language feature, *strict constructors*, that can have a big impact on performance.

If we take a look at the definition of complex addition again, figure 12, we can see that a disaster lurks even in the specialized version. The constructor for complex numbers, `:+`, does not evaluate its arguments (since we have non-strict semantics) so when a complex addition is performed the resulting number will have components that are unevaluated. This will result in a function like `sum` accumulating a lot of unevaluated data until it returns and a component is selected.

A strictness analyser capable of handling data structures and having the whole program available may be able to figure out that it is safe to perform the additions (of `Double`) instead of postponing them. But these kind of analysers seems to be quite expensive to use, and have not found their way into real compilers yet. They would also be unable to handle separately compiled modules.

6.1 Strict data types

We propose to extend the Haskell so we can specify that a constructor is to be strict in certain components. The Clean language, [SNvGP91], as well as LML, [AJ89] has this feature. Using this we could give a different definition of `Complex` (where we use postfix `!` to indicate a strict component):

```
data (RealFloat a) => Complex a = a! :+ a!
```

This data type definition will guarantee that whenever we have an evaluated `Complex` the parts will also be evaluated (to WHNF). One could argue that this corresponds closer to the mathematical definition of a complex number. If a complex number is supposed to define a point in the complex plane, what point would that be if one of the coordinates is undefined?

Using this definition of complex numbers we can finally get a version of `sum` that is well-behaved when summing `Complex Double`, i.e., it runs in constant space and does not build any closures.

7 Pragmas

We regard separate compilation as an essential part of our compiler, and we also think that breaking up a program into small modules should not be penalized. The compiler can doubtless do a better job when it has the complete program at its disposal, but we do not regard having that as a realistic assumption. When a module imports another module the compiler can do a better job if it is given as much information as possible about the imported module. The Haskell Report defines how a module interface should look. The interface contains the minimum amount of necessary information about a module, namely the what types and values (with their types) that are exported. We have added to this by using annotations in the interface files that conveys arity and strictness information.

For each value exported from a module the interface file describes what specialized versions exists, what the arity is, what the strictness and termination properties are. This information is not only for the functions and values explicitly exported, but also for those generated by class and instance declarations.

8 Measurements

Table 1 contains measurements of the impact of various transformations on some programs. Only some combinations were measured since making the measurements is very tedious. All the various transformations cannot be used one-by-one anymore in the compiler so the programs were transformed by hand to achieve the same effect. All measurements were performed on a SparcStation2. The time does not include time for garbage collection since we want to measure the effect of overloading alone. If GC time was included the improvement would be even bigger since the improved programs allocate less memory.

Some of the benchmark programs are listed in appendix B.

nfib	The old favourite. It is a lousy benchmark program from most respects, but it uses overloading in the extreme. The `nfib` functional has its most general type here, `(Ord a, Num a, Num b)=>a->b`, which means that three dictionaries must be used.
nfibI	This is a small variation on `nfib` where the types has been restricted to the more reasonable type `(Integral a)=>a-a`. Note that this attempt to improve `nfib` just makes it worse unless the superclass structure is flattened, because all methods have to extracted be from superclass dictionaries.
idInt	A program that uses complex numbers and floating point a lot. Using complex numbers means that overloading must be used.
hinc	This program uses overloaded literals a lot.
choleski	This program, written by Steven Bevan and available with the Haskell library, does Choleski factorization. The program is written using overloading where possible to get a very general factorization module.
sched	This is a "real" program for job scheduling. It is taken from the benchmark set described in [HL93], but has been rewritten to look more like a Haskell program. This program has non-overloaded type signatures for all top level functions.

The transformations are

tuple	The old translation scheme with dictionaries as tuples.
flat-super	Dictionaries as tuples, but with the superclass structure flattened.
vectors	The new translation with dictionaries as a pair of vectors.
fast-const	Special handling of literals of type `Int` and `Double`.

	program											
	nfib		nfibl		idlnt		hinc		choleski		sched	
tuples	234	(25)	264	(28)	4.6	(7.1)	-	(-)	-	(-)	-	(-)
flat-super	-	(-)	204	(21)	-	(-)	-	(-)	-	(-)	-	(-)
vectors	107	(11)	105	(11)	3.0	(4.6)	242	(292)	-	(-)	-	(-)
fast-const	87	(9.1)	84	(8.8)	2.9	(4.5)	1.55	(1.9)	19	(1.9)	6.6	(1.03)
spec-inst	87	(9.1)	-	-	2.1	(3.2)	-	(-)	-	(-)	-	(-)
spec-func	87	(9.1)	-	-	1.9	(2.9)	-	(-)	-	(-)	-	(-)
auto-specialize	9.5	(1.0)	9.5	(1.0)	1.0	(1.5)	0.83	(1.0)	10	(1.0)	6.4	(1.0)
strict-complex	-	(-)	-	-	0.65	(1.0)	-	(-)	-	(-)	-	(-)

Table 1: Execution times in seconds (normalized) for various programs and transformations. Untested combinations are indicated by a "-".

spec-inst	Specialised instance declarations for common types in the Prelude.
spec-func	Specialised function declarations for common types in the Prelude.
auto-specialize	Automatic function specialization with a module.
strict-complex	Complex numbers with strict components.

The conclusions that can be drawn from these (and other) measurements is that if overloading is used, then the program can be improved something between moderately and dramatically. If on the other hand the programmer has carefully inserted non-overloaded signatures, there is little to be gained (which was to be expected).

9 Related work

All Haskell compilers probably implement what we have described here to a lesser or greater extent (probably a few more things as well), but descriptions of what has been done are scarce.

Somewhat related, but not the same, is what has been done for object oriented languages. For these languages there is no need for separate dictionaries; each object in some sense carries its own dictionary. There are some very efficient techniques for implementing method calls that could be worth investigating such as those used in C++ [Str] or Smalltalk [DS84].

10 Future work

The only thing described here which is not implemented yet is the inlining; it will be done soon.

A different, but interesting, approach would be to do code generation at run time to produce specialized code for heavily used functions (similar to the object oriented system Self [Cha93]). With a good partial evaluator one could imagine specializing the compiler to a particular function, but with the type for which is supposed to generate code to be supplied at runtime. One would thus compile code not for a function itself, but code that when supplied with a type will generate code for the function. This would allow separate compilation, but still give efficient code (at least the second time a function is called).

By collecting information about how overloaded functions are used in a program it is possible to recompile parts to get specialized versions without user annotation. One could also have a compiler that postpones actual code generation until all modules are available (i.e., link time) and which produces all needed versions then. These kind of systems would be more expensive in terms of compile time, but one could have it as an option when generation the final "production" version of a program.

11 Acknowledgments

As usual I would like to thank the Multi-group for a stimulating environment, and also Wei-Ngan Chin and the National University of Singapore where part of this work was done. I would also like to thank all the people who have commented on this paper and in particular Urban Boquist, Niklas Röjemo, and Jan Sparud.

References

[AJ89] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML Compiler. *The Computer Journal*, 32(2):127–141, 1989.

[Aug93] L. Augustsson. *HBC User's Manual.* Programming Methodology Group, Department of Computer Sciences, Chalmers, S–412 96 Göteborg, Sweden, 1993. Distributed with the HBC compiler.

[Cha93] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object Oriented Programming Languages.* PhD thesis, Computer Science Department, Stanford University, 1993.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings 1984 Symposium Principles of Programming Languages*, pages 297–302, Austin, Texas, 1984.

[HHPW92] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type Classes in Haskell. Report, Department of Computer Science, Glasgow University, 1992.

[HL93] P. Hartel and K. Langendoen. Benchmarking implementations of lazy functional languages. In *Proceedings 1993 Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, 1993.

[Hud92] Paul Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[Joh87] T. Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, February 1987.

[Jon] Mark Jones. A new approach to the implementation of type classes. Appeared on the Haskell mailing list.

[Kae88] Stefan Kaes. Parametric overloading in polymorphic programming languages. In *2nd European Symposium On Programming*, Lecture Notes in Computer Science, pages 131–144. Springer Verlag, 1988.

[PJS89] S. L. Peyton Jones and Jon Salkild. The Spineless Tagless G-machine. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, London, Great Britain, 1989.

[SNvGP91] Sjaak Smetsers, Erik Nöcker, John van Groningen, and Rinus Plasmeyer. Generating efficient code for lazy functional languages. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, Cambridge, Massachusetts, July 1991.

[Str] Bjarne Stroustrup. *C++ Reference Manual.*

[WB89] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc.* In *Proceedings 1989 Symposium Principles of Programming Languages*, pages 60–76, Austin, Texas, 1989.

Appendix

A Subset of the Haskell Prelude

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = if x == y then False else True

class  (Eq a, Text a) => Num a  where
    (+), (-), (*) :: a -> a -> a
    negate        :: a -> a
    abs, signum   :: a -> a
    fromInteger   :: Integer -> a
    fromInt       :: Int -> a

data (RealFloat a) => Complex a = a :+ a
                      deriving (Eq,Binary,Text)

instance  (RealFloat a) => Num (Complex a)  where
    (x:+y) + (x':+y') = (x+x') :+ (y+y')
    (x:+y) - (x':+y') = (x-x') :+ (y-y')
    (x:+y) * (x':+y') = (x*x'-y*y') :+ (x*y'+y*x')
    negate (x:+y)     = negate x :+ negate y
    abs z             = magnitude z :+ 0
    signum 0          = 0
    signum z@(x:+y)   = x/r :+ y/r where r = magnitude z
    fromInteger n     = fromInteger n :+ 0
```

```
sum :: (Num a) => [a] -> a
sum l = sum' l 0
        where   sum' []     a = a
                sum' (x:xs) a = sum' xs (a+x)
```

B Some benchmark programs

```
-- The nfib benchmark
main = prints (nfib 30) "\n"
nfib :: (Ord a, Num a, Num b) => a -> b
nfib n = if n < 2 then 1 else 1 + nfib(n-1) + nfib(n-2)


-- The nfibI benchmark
main = prints (nfib 30) "\n"
nfib :: (Integral a) => a -> a
nfib n = if n < 2 then 1 else 1 + nfib(n-1) + nfib(n-2)


-- The idInt benchmark
main = prints (idInt 2500) "\n"

idInt :: Int -> Int
idInt i = round (realPart (sum [one x | x<-[1..i]]))

one :: Int -> Complex Double
one i = mkPolar 1 (2*pi/fromInt i) ^ i


-- The hinc benchmark
main = prints (do 100000 hinc 0.0) "\n"

do :: Int -> (a->a) -> a -> a
do 0 f x = x
do n f x = do (n-1) f (f x)

hinc :: (Fractional a) => a -> a
hinc x = x + 0.5
```