Babel 2001 Preliminary Version

Object-Oriented Style Overloading for Haskell
(Extended Abstract)

Mark Shields [1] Simon Peyton Jones [2]

Microsoft Research Cambridge
7 JJ Thomson Avenue
Cambridge, CB3 0FB, UK

Abstract

Haskell has a sophisticated mechanism for overloading identifiers with multiple definitions at distinct types. Object-oriented programming has a similar notion of overriding and overloading for methods names. Unfortunately, it is not possible to encode object-oriented overloading directly using Haskell overloading. This deficiency becomes particularly tiresome when Haskell programs wish to call methods imported from an object-oriented library.

We present two refinements of Haskell's type class system: *Closed* classes and *overlapping instances*. We demonstrate how we may exploit the refined system so as to be able to encode object-oriented classes within Haskell. This encoding allows us to mimic, within Haskell, the overloading resolution rules employed by object-oriented languages without the need for additional type annotations or name mangling. As a consequence, object-oriented class libraries are very convenient to import and use within Haskell.

1 The problem

The purpose of this paper is to make it easy to import libraries from Java[9] or .NET[18], into a Haskell program. By "easy" we mean that it should be as easy to use the library from Haskell than from its native language. Indeed, Haskell's higher order features and first-class monadic values make it a powerful glue language, so if we succeed it might even be *easier* to use the library from Haskell than from its native language. However, these advantages will not be persuasive if things that are easy in the native language are clumsy in Haskell. That is the challenge we address here.

[1] `markshie@microsoft.com`
[2] `simonpj@microsoft.com`

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

The idea of mapping an object-oriented library into the Haskell type system is not new [6] — we review it in Section 2. In this paper, we make three new contributions:

Subtyping. Object oriented languages make extensive use of implicit coercions between a subtype and its supertypes, while Haskell lacks the entire notion of subtyping. In our earlier work [6] we described how to use polymorphism to encode subtyping using so-called "phantom types". Alas, this approach breaks down when we encounter the multiple supertyping of interface types. In Section 3 we discuss the design alternatives, and show an alternative encoding for subtyping, using type classes, that is adequate for our purposes.

Ad hoc overloading. While Haskell supports overloading, all the overloaded instances must share a common type pattern. In contrast, many object-oriented languages allow a single method name to be overloaded at *unrelated* types. One can evade this difficulty by using name-mangling to give a distinct name to each distinct overloading of a single method name, but that is extremely unattractive in practice.

In Section 4 we present an extension to Haskell's type class mechanism that smoothly accommodates truely *ad hoc* overloading. To make it work effectively in practice, we introduce the idea of a *closed class*, which in turn allows the type checker to make *improvement* to inferred types, and hence reduce the need for type annotations.

Overlap. Many object-oriented languages also allow a single method name to be overloaded at *overlapping* types; that is, several methods would be well-typed, but one of them is the "best match" for the types at the call site. The definition of "best match" is the subject of subtle, carefully-worded, but informal, passages in the language manual.

Hugs and GHC both support the closely-related notion of *overlapping instance declarations*, but what exactly these mean is even less well specified, and polymorphism makes the setting more complicated than the corresponding object-oriented problem.

In Section 4.4 we tackle this issue head-on, giving a precise story about when and how overloading is resolved in the presence of overlap.

These extensions have subtle implications for type inference, as we discuss in the full paper [24]. The full paper also contains a formal description of type checking and type inference to complement the informal explanations used here.

Our extensions generalise Haskell's existing *qualified types* [12]. For example, Haskell's negation function has type:

```
negate :: (Num a) => a -> a
```

This type says that `negate` can be applied to any type `a` that satisfies the *type* *constraint* `(Num a)`. At run-time, `negate` takes an extra parameter apart from the value of type `a`, namely a *witness* that `(Num a)` indeed holds. In concrete terms, the witness for `Num a` is a tuple, or dictionary, of functions for operating over numeric values, one of which is the negation function.

This approach turns out to have many useful generalisations, each obtained by introducing a new form of type constraint, along with a corresponding new form of witness. Concrete examples include: implicit parameters [15]; extensible records [7] [11]; and type-indexed rows [23]. We take exactly the same approach in this paper.

There is a danger here. Is our work simply "yet another extension of Haskell type classes?" How long can we go on adding new extensions before the whole system becomes unusably complicated? These are good questions. One would like to find a unifying framework into which all these extensions could fit as special cases. Sulzmann and Stuckey propose Constraint Handling Rules as such a framework [8]. In this paper we also also take steps towards a general framework. However, unifying frameworks are easier to design when there is a rich zoo of motivating special cases, and our main purpose here is to work out in detail some extra inhabitants for the zoo.

2 Mapping OOP into Haskell

Given a Java or .NET library, how can we map it into Haskell's world? More precisely, given the definition of a Java or .NET class, we want to specify the interface of a Haskell module whose implementation is that class. For the sake of definiteness we will use $C^\sharp$ [17] as the representative language in which the library is written, but everything we say applies unqualified to other .NET libraries, and with very minor qualifications to Java libraries.

We do not address the question about how the interface might be *implemented*. A possible route for .NET would be to compile Haskell to the .NET intermediate language; a possible route for Java would be to use the Java Native Interface [16]. In this paper, however, we focus on the design of the interface.

We begin by briefly reviewing the approach described in [6] for mapping an object-oriented library into Haskell. Consider the following $C^\sharp$ class:

```
class C {
  C( int x ) { ... } ;      /* Constructor */
  static int s( int x );    /* Static method */
  int m( bool b, int x );   /* Instance method */
}
```

This class would be mapped into the following Haskell types and functions:

```
newtype C              -- An abstract type
newC :: Int -> IO C
s    :: Int -> IO Int
m    :: (Bool,Int) -> C -> IO Int
```

The $C^\sharp$ class `C` is mapped to an abstract Haskell type `C`. We write it here `newtype` without a right hand side, because its representation is (of course) hidden. The constructor is called `newC`, takes the appropriate arguments, and returns a result of type `C`. More precisely, it returns a result of type `IO C`, because creating a new value of type `C` is a side effect [3]. The static method `s` has the expected type, again remembering that it may have a side effect.

The instance method `m` takes a "self" parameter of type `C` as its *second* argument, with the ordinary arguments, in a tuple, as its *first* argument. One might expect the self parameter to be first, but putting it last allows a neat coding trick [6]. Suppose we have `x::C`; then we can write the OO-like call

```
x # m (True,3)
```

to call `x.m`, where the infix operator `#` is defined as reverse application, thus:

```
x # f = f x
```

Recalling that, in Haskell, function application binds more tightly than anything else, we have

```
x # m (True,3)    =    m (True,3) x
```

One could equally well choose to have the self parameter as the first argument; it does not affect anything else in this paper.

Why are the arguments to `m` tupled? Again, this is a design choice. Our intuition is that OO methods are not designed with currying in mind, and so are likely to be called with all their arguments. Given this, we are likely to get less confusing error messages if the arguments are uncurried, especially by the time we have added ad-hoc overloading.

Lastly, one might ask whether all methods need be in the `IO` monad; after all, some will be purely-functional, and need not be. Indeed so, and perhaps some kind of pragma or meta-data could express this fact. If so, it is readily accommodated by omitting the `IO` monad from the type of the pure method. We do not consider the question further here.

3 Subtyping

Consider the following $C^\sharp$ class declarations:

[3] A value of type `IO t` is a computation that may perform some side effects before returning a result of type `t`. See [21] for a tutorial.

```
class C {
  int opC( int x ) { ... }
}

class D : C {     /* D extends C */
  bool opD() { ... }
}
```

In $C^\sharp$, if `d` has type `D`, then one can write `d.opC(3)`, because any value of type `D` *is also* a value of type `C`. This is called *inclusion subtyping* because no coercion is needed to convert a `D` to a `C` [19]. (In future, we will often write "`d::D`" as short for "`d` has type `D`" even when the variables and types are those of $C^\sharp$.)

How does this look in Haskell? If we simply expose the types and operations in Haskell as earlier described, we get this:

```
newtype C
newtype D

opC :: Int -> C -> IO Int
opD :: D -> IO Bool
```

The trouble is, of course, that `opC` is not applicable to a value of type `D`, because Haskell does not understand the subtype relationship between `C` and `D`.

This problem is not new. One solution is to use "phantom types" [6] to encode the class heirarchy, but this fails for classes with more than one superclass. Another approach is to use first-class existentials and class constraints [10], but Haskell only allows existentials to be introduced by a data constructor, which defeats the purpose of this encoding. In the full paper [24] we consider these options more fully, and explain why they are inadequate for our purposes. We also consider adding full subtyping to Haskell.

The encoding we adopt is as follows. For each $C^\sharp$ class `C` we generate (a) a Haskell *type* `C`, and (b) a Haskell *type class* `SubC`. Thus:

```
newtype C
class SubC c where {}
instance SubC C
opC :: SubC c => Int -> c -> IO Int
```

We are back to the simple situation in which there is a Haskell type `C` that models the $C^\sharp$ class (= type!) `C`. A type is an instance of `SubC` if the corresponding $C^\sharp$ type is a subtype of class `C`. So the Haskell type `C` is certainly an instance of `SubC`. Finally, `opC` accepts a value of any type that is in `SubC` – i.e. is a subtype of `C`.

Now we can add the encodings of the sub-class `D` and interface `I`:

```
newtype I
class SubI i where {}
instance SubI I
opI :: SubI i => Bool -> i -> IO (C Void)

newtype D
class (SubI d, SubC d) => SubD d where {}
instance SubC D
instance SubD D
instance SubI D
opD :: SubD d => d -> IO Bool
```

Each comes with a new type `D` and `I`, and a class, `SubD` and `SubI`. The new type `D` is an instance of `SubC` as well as `SubD` and `SubI`, and hence `opC` and `opI` can be applied to a value of type `D`.

The superclass relation embodies the expected subtyping properties. For example, consider this function:

```
h :: SubD d => d -> IO Int
h d = do { n <- opC 3 d ;
           b <- opD d   ;
           return n }
```

The call to `opC` generates the constraint `SubC d`, but it is entailed by the constraint `SubD d` arising from the call to `opD`, so the type of `h` has just the single constraint we expect.

Notice that the `SubX` classes have no methods — we use them solely to model the subtype relationship. Since they have no methods, we need pass no evidence for them, so they have no run-time overhead. (Haskell allows the "`where {}`" of a class declaration to be omitted when there are no methods, and we will do so in future.)

If the class hierarchy becomes deep, one may have to write a large number of instance declarations, because each new type must be made an instance of all its superclasses. However, we expect the encoding to be carried out by an automatic tool that reads .NET meta-data and spits out the encoding, so we are not too worried. Of course, the soundness of this encoding depends on the programmer getting the subtype instances right, and not arbitrarily adding new instance declarations.

4 Ad hoc overloading

We accommodated subtyping without extending Haskell, but we will not be so fortunate in the case of ad-hoc overloading. Consider the following $C^\sharp$ class declaration:

```
class C {
  int m( int x );
  bool m( bool b );
}
```

Following the simple approach of Section 2, we would get two Haskell functions, both called `m`:

```
m :: Int -> C -> IO Int
m :: Bool -> C -> IO Bool
```

But Haskell does not permit two distinct functions to have the same name. One alternative is to use name-mangling:

```
m_Int  :: Int -> C -> IO Int
m_Bool :: Bool -> C -> IO Bool
```

From the point of view of Joe Programmer, this is a big step backwards, especially as OO libraries typically make heavy use of this sort of overloading. (The overloading of constructors for the class is another example.) Worse, one must either invent simple rules for name mangling that give very long names, or else have complicated rules that usually give shorter names. There just does not seem to be a good point in this design space.

4.1 Degenerate classes

A more promising possibility is to employ Haskell's type classes in a rather stylised way[4]:

```
class Has_m a where
  m :: a

instance Has_m (Int -> C -> IO Int) where
  m = m_Int

instance Has_m (Bool -> C -> IO Bool) where
  m = m_Bool
```

The name-mangled functions `m_Int` and `m_Bool` still exist behind the scenes, but the programmer never thinks about them. She simply calls `m`, which has type

```
m :: (Has_m a) => a
```

and with a bit of luck the local type constraints will be enough to figure out which instance declaration to use. After all, they are enough in a $C^\sharp$ program! Even if the type constraints don't specify which instance to use, the type system can *abstract* over the constraint, which is *more* than is possible in $C^\sharp$.

[4] Haskell experts will notice that the instances for `Has_m` go beyond the Haskell 98 standard, but we do not want to labour the point here since we intend to discard this approach.

For example, we can write[5]:

```
mlist :: (Has_m (a->b->IO c)) => a -> [b] -> IO [c]
mlist a cs = mapM (m a) cs
```

By abstracting over the constraint, we defer its choice to the call site of `mlist`; in exchange we pay a modest run-time penalty, by passing the method as a parameter to `mlist`.

You might wonder whether we could make the class `Has_m` a little less degenerate thus:

```
class Has_m self arg res where
   m :: arg -> self -> IO res
```

However, the same method name `m` may be used for static methods (which lack a `self` parameter), and for purely-functional methods (whose result type is not in the `IO` monad), so there is virtually no useful common structure.

This class-per-method approach is reminiscent of System O [20]. However, unlike System O, we cannot require all instances of `Has_m` be distinguished by the type of the method's first argument.

4.2 Improvement

Unfortunately, this stylised use of Haskell's existing type classes does not work in practice. Assuming the same two `instance` declarations as in Section 4.1, suppose we see the following function definition:

```
f c x = m (x::Int) (c::C)
```

Performing type inference on the right-hand side of `f` will give rise to a class constraint `Has_m (Int -> C -> r)`, for some unknown type `r`, represented by a fresh type variable. Any $C^\sharp$ programmer would expect that once `x` is fixed to have type `Int`, and `c` to have type `C`, there is only one choice for which instance of `m` to choose, namely `m_Int`. But that is not how Haskell works: one cannot instantiate either of the two `instance` declarations for `Has_m` to get `Has_m (Int -> C -> r)`. So Haskell will generalise over the constraint to get:

```
f :: (Has_m (Int->C->r)) => C -> Int -> r
```

This is wonderfully general, because it allows for the possibility that the call site might know about other instances of `Has_m`. But it is really *too* general, and will give rise to all sorts of ambiguity errors. For example, suppose we wrote:

```
do { r <- m (x::Int) (c::C) ;
     print (show r) }
```

If the knowledge of `m`'s argument types does not fix its result type, the `show`

[5] The standard function `mapM` has type `(a->IO b) -> [a] -> IO [b]`.

does not know what type its argument will be, and the compiler will reject the program as ambiguous. This is really no good.

Instead, the type inference system must perform what Mark Jones calls "improvement" [13]. Given the class constraint `Has_m (Int -> C -> r)` *there is only one instance for `m` that matches this constraint*, namely:

```
instance Has_m (Int->C->IO Int) where m = m_Int
```

Since there is exactly one choice, we should make it now, *and that in turn fixes `r` to be `Int`.* Hence we get the expected type for `f`:

```
f :: C -> Int -> IO Int
```

It is this additional unification step that constitutes the "improvement". Now the class constraint can be discharged (fixing which instance of `m` to call), and inference can proceed.

What is the justification for doing this improvement? Answer: it is simply a design choice, and one based on the idea that the class `Has_m` is *closed.* We might declare it like this:

```
class closed Has_m a where
  m :: a
```

By "`closed`", we mean that we allow the type inference algorithm to commit to which instance of `Has_m` to use based on the instances that are currently in scope. In contrast, for type classes, it seems generally better to defer such choices, as discussed in [22]. An elaboration of type classes, called *functional dependencies*, does support improvement [14]; but the sort of improvement we need for `Has_m` constraints cannot be modelled by functional dependencies.

This notion of closedness has appeared elsewhere in the guise of closed kinds [5]. System CT [3] also makes a similar closed-world assumption (Section 6).

4.3 Method constraints

So far, we have seen how to extend Haskell's type-class mechanism to support ad-hoc overloading, by adding the idea of a `closed` class. From a programming point of view, though, using it seems rather a heavyweight approach. We have to invent a new class for each method name, and there may be no obvious place to declare the class. (The method name may be used in multiple sibling libraries.) Indeed, having to declare the class at all seems cumbersome. Lastly, the `Has_m` class must somehow be declared as "closed".

Instead, we provide direct syntactic support by introducing a new form of type constraint, a *method constraint.* For example, we can write the type of `mlist` like this:

```
mlist :: (m :: a->b->IO c) => a -> [b] -> IO [c]
```

We have simply *identified* the degenerate class `Has_m` with the overloaded function m. Corresponding to the new form of constraint is a new form of instance declaration[6]:

```
instance m :: Int -> C -> IO Int where
    m = m_Int

instance m :: Bool -> C -> IO Bool where
    m = m_Bool
```

There is no need to declare a class. The function m is brought into scope by any instance declaration for m, and has the type:

```
m :: (m::a) => a
```

At first sight this type may look confusing, but it simply says this: the function m has any type a that satisfies the method constraint m::a. (Recall that in Haskell all types are universally quantified over their free type variables, so this type for m means m :: $\forall\alpha$. (m :: α)=>α).) We are using the same name, "m", for both the *function* m and the *method constraint* m, but functions and method contraints live in different name spaces, so there is no confusion — compare the type of m in Section 4.1.

The function m can be exported and imported by name, just like any other function.

Overloaded functions can be polymorphic without any difficulty. For example:

```
instance op :: [a] -> [a]            where op = op1
instance op :: Bool -> Bool -> Bool where op = op2
```

Here, the first overloading of op is polymorphic, while the second is not. As before, we simply pick the one that matches the method constraint. For example, the call op [1,2,3] matches the first instance, but not the second, so we can safely commit to the first.

Nor is there any difficulty if the overloaded function has a context in its type. For example, we can add a third instance for op:

```
instance op :: Num a => Maybe a -> a where op = op3
```

Now, if we encounter the call op (Just 3) we again know exactly which instance to pick, in this case driven by the type of the first argument.

4.4 Overlapping instances

Consider the following $C^\sharp$ class declarations:

[6] Others have suggested that a better keyword might be "overloaded" rather than "instance".

```
class B : A { ... }

class C {
  int m( A x ) { ... } ;
  int m( B x ) { ... } ;
}
```

Now consider a call `c.m( b )` where `b::B` and `c::C`. Both `m` methods are applicable to `b`, but the second is a better "fit" to the argument type. On the other hand, given the call `c.m( a )`, where `a :: A`, the second method is not applicable, so the first is used. The sections of the $C^\sharp$ language specification that describe exactly what "best fit" means are carefully written, but still informal.

What is the corresponding problem in Haskell? The above declarations will be rendered thus:

```
class SubA b => SubB b

instance m :: (SubA a, SubC c) => a -> c -> IO Int
instance m :: (SubB b, SubC c) => b -> c -> IO Int
```

The overlap problem is that anything that matches the second instance declaration will also match the first. Overlapping instance declarations are not permitted in standard Haskell 98, but are present in various experimental extensions. However, we are not aware of any precise description of the type system of Haskell together with overlapping instance declarations. Indeed the combination of overlap with multiple arguments, and polymorphism, is rather subtle. A key contribution of the full paper is to give a precise account of how they interact. In particular, we establish a partial ordering on instance declarations which resembles the instantiation ordering on type schemes, and specify that a method constraint may be resolved to a particular instance only when it is the least amongst all candidate instances.

There is one difference beteen our approach and that taken by existing Haskell implementations that support overlapping instances. Both GHC and Hugs prohibit instance declarations that unify without overlapping. For example:

```
instance Eq a => Wuggle (Int, a) where ...
instance Eq a => Wuggle (a, Int) where ...
```

These two instance declarations would be rejected, because the constraint

```
Wuggle (Int,Int)
```

matches both of them, yet neither is more specific than the other. In this paper, we advocate allowing the instance declarations, raising an error only if the constraint `Wuggle (Int,Int)` acutally comes up in practice. (If it does, there will be two candidate instances, and we will report an ambiguity error.) But it may not come up! Instead we may encounter the constraint `Wuggle (Int,Char)`, which matches only one of the intances, or

`Wuggle (Bool,Int)`, which matches only the other instance. In short, the instance declarations are innocent, and potentially useful. Our framework allows them, and yet only makes a commitment when there is a unique choice.

Nothing in the above discussion is specific to imported .NET or Java libraries. Ad-hoc, overlapping overloading can usefully be deployed in native Haskell programs.

5 Encoding class hierarchies

In Figure 1 we show a larger $C^\sharp$ class hierarchy. (As before, we use $C^\sharp$ as our prototypical foreign language.) The corresponding Haskell interfaces are given in Figure 2, while Figure 3 shows some well-typed Haskell programs that use these interfaces.

Notice that there is one `instance` declaration for each *call pattern* of a method. By call pattern, we mean the actual bytecode sequence to invoke the appropriate method. This can be a little confusing. For example, the virtual method `o` in class `E` is overridden in class `F`. Even though method `o` has two implementations, there is only one calling pattern for `x.o`, since virtual method dispatch is through the vtable associated with `x`. Hence, there is only one instance declaration for `o`. Similarly, method `m` (on integers) in interface `I` has no implementation *per se*, but any class which implements interface `I` must supply such an implementation. Again, the same calling pattern applies to each implementation, and thus there is a single instance declaration for `m` (on integers). By contrast, when method `n` is overridden in class `F`, the calling pattern changes, and so we supply a new instance declaration.

5.1 Sub-classing and call-backs

This paper discusses how to *import* classes from $C^\sharp$, but it does not discuss how to *export* classes to $C^\sharp$. We provide no way to define a completely new $C^\sharp$ class in Haskell, or even to create a sub-class of an existing $C^\sharp$ class. If we wanted to allow this, we would have to make much more substantial changes to the language; the MLj compiler exemplifies this approach [1].

However, some $C^\sharp$ library methods (especially those involved with graphical user interfaces) rely on sub-classing to define "callback objects". For example the library method might be

```
class Button {
  void OnClick( Click h )
  ...other methods...
}
```

where the `Click` interface is defined thus:

```
/* NB: parameter names and method bodies
       omitted for the sake of brevity */

class A     { ... }
class B : A { ... }
class C     { ... }
  /* A,B,C have nullary constructors */
class D : C { D(char); ... }
  /* D has an explicit constructor */

interface I {
  int m(int);
}

interface J {
  int m(int, int);
  int m(int, bool);
}

class E : I, J {
  E();     /* Overloaded constructor */
  E(bool);
  int m(int);
  int m(int, int);
  int m(int, bool);
  int n(A, D);
  int n(B, C);
  virtual int o(int);
}

class F : E {
  F();
  new int n(A, C);
  override int o(int);
}

class G {
  G();
  int m(int);
}
```

Figure 1: An example class heirarchy

```
newtype A; class SubA a; instance SubA A
newtype B; class SubB b; instance SubB B
instance SubA B
newA :: IO A; newB :: IO B

newtype C; class SubC c; instance SubC C
newtype D; class SubD d; instance SubD D
instance SubC D
newC :: IO C; newD :: Char -> IO D

-- interface I
newtype I; class SubI i
instance m :: SubI i => Int -> i -> IO Int

-- interface J
newtype J; class SubJ j
instance m :: SubJ j => (Int,Int)  -> j -> IO Int
instance m :: SubJ j => (Int,Bool) -> j -> IO Int

-- class E
newtype E; class SubE e; instance SubE E
instance SubI E; instance SubJ E
instance newE :: IO E;
instance newE :: Bool -> IO E;

instance n :: (SubA a, SubD d, SubE e)
           => (a, d) -> e -> IO Int
instance n :: (SubB b, SubC c, SubE e)
           => (b, c) -> e -> IO Int
instance o :: SubE e => Int -> e -> IO Int

-- class F
newtype F; class SubF f; instance SubF F
instance SubE F; instance SubI F; instance SubJ F
newF :: IO F
-- n is new, so new instance for n
instance n :: (SubA a, SubC c, SubF f)
           => (a, c) -> f -> IO Int
-- o is overridden, so no new instance for o

-- class G
newtype G; class SubG g; instance SubG G
newG :: IO G
instance m :: SubG g => Int -> g -> IO Int
```

Figure 2: Representation of Figure 1 in Haskell

```
f :: J -> Int -> IO Int
f j y = do p <- j # m (y, 1)
           q <- j # m (y, True)
           return p + q

g :: E -> IO Int
g e = do a <- newA
         b <- newB
         c <- newC
         d <- newD
         i <- e # n (a, d)
         j <- e # n (b, c)
         return i + j
```

Figure 3: Example well-typed terms using declarations of Figure 2

```
interface Click {
  void ClickMe()
}
```

The `OnClick` method installs the callback object `h` as a handler to service button clicks. A functional programmer would think of such a callback object as simply a closure, but a $C^\sharp$ programmer must define a sub-class of the `Click` interface, thus:

```
class MyClick : Click {
  OnClick() { ...service a click... }
}
```

($C^\sharp$ also has a notion of *delegates* which is slightly more convenient in this situation, but nevertheless the problem remains.) Since we cannot sub-class in Haskell, does that render the `Button` class useless to the Haskell programmer?

We can solve the problem, albeit slightly clumsily. What we want to do is to give behaviour to the `Click` interface; we do not want to add methods or otherwise extend it. We can write a generic `MyClick` class like this:

```
class HClick : Click {
  private HaskellClosure h;
  new( HaskellClosure h' ) { h = h'; }
  OnClick() { h.run() }
}
```

Defining this class requires knowledge of the representation of Haskell closures in the .NET world. In particular, the `run` method of a `HaskellClosure` will perform its I/O actions. The code for `HClick` could be generated from the interface specification for `Click`, though we have not yet implemented this.

If we now import this class into Haskell, using the mechanisms already defined, we can now create a callback object using `newHClick`:

```
  onClick :: Button -> IO () -> IO ()
  onClick but click_handler
    = do { cb <- newHClick( click_handler ) ;
           but # onClick( cb ) }
```

The bottom line is this: we can create callback objects without too much difficulty, but we cannot create genuinely new classes and export them back to the $C^\sharp$ world.

5.2 A dark corner: The `new` modifier

In fact, Figure 2 is not completely accurate in its representation of $C^\sharp$'s overload resolution. Consider the Haskell call

```
   n (b::B, d::D) (f::F)
```

All three instances for `n` in Figure 2 are candidates, but there is no best fit. So the type inference engine will complain that it cannot choose, and (what is worse) we can't fix the problem by supplying more type information.

What happens in $C^\sharp$, given the call `f.n( b, d )`? The semantics of the "`new`" qualifier for method `n` in class `F` is that this definition of `n` "hides" all definitions of `n` in `F`'s sub-classes. So now there is only one candidate to choose.

We can accommodate this in Haskell, but only in a rather brutal way. In Figure 2, interface `I` defines `m` thus:

```
  instance m :: SubI i => Int -> i -> IO Int
```

The subtyping constraint on `i` means that `m` works on values of type `E`, as it should. But we could instead say:

```
  instance m :: Int -> I -> IO Int
```

and *in addition*, when `E` is declared, add

```
  instance m :: Int -> E -> IO Int
```

However, these two `instance` declarations would share a common witness. In effect, we simply copy all inherited methods into each sub-class, with fresh `instance` declarations but common witnesses.

What does this buy us? It allows us to *refrain* from copying the instances of `n` into `F`'s class, so that there just a single instance for `n` with self-parameter `F`:

```
  instance n :: (SubA a, SubC b)
             => (a, b) -> F -> IO Int
```

The $C^\sharp$ design treats the self parameter specially, whereas our system does not.

6 Related work

6.1 System CT

System CT [3] is a Haskell-like type system that supports ad-hoc polymorphism in a similar manner to that described in Section 4. For example, CT will infer the following type for `insert`:

```
insert :: {(==) :: a->a->Bool}. a -> [a] -> [a]
insert a []                   = [a]
insert a (b:xs) | a==b        = b : xs
                | otherwise = b : insert a xs
```

Our syntax differs slightly from CT's, but the method constraint `(==)::a->a->Bool` plays the same role in both systems. However, System CT takes a more radical approach than we do. CT has no `class` or `instance` declarations; instead every `let`-definition introduces a new potentially-overloaded identifier. (To mimic this in our system, one would have an `instance` declaration for every `let`-definition.)

Since every `let`-definition is effectively an instance declaration, System CT must confront and solve the issue of *local `instance` declarations*. That is not something we have tackled in the main body of our paper. It is present in our formal treatment, and while it does not much complicate the typing rules, we believe that it would add signficant complexity to proofs about the system.

On the other hand, we are forced (by our desire to import .NET classes) to confront and solve overlap, whereas CT is not.

6.2 Multi methods

Recall that our encoding of $C^\sharp$ classes lifts all methods out into a single namespace, and relies on ad-hoc overloading to distinguish methods of the same name belonging to distinct classes. Indeed, we don't treat overloading across classes (class C and D both implement a method called m) any differently from overloading within classes (class C implements two methods called m).

In this respect, our approach is very similar to that of "multi-method" based object-oriented languages such as CLOS [4]. In these languages, methods are regarded simply as overloaded functions, and method dispatch is based on the dynamic types of all method arguments instead of just the (implicit) "this" argument.

Bourdoncle and Metz [2] have proposed an ML-like language built upon this notion of multi-methods which has many similarities with the work of this paper. In particular, they use constrained polymorphism and subtype constraints to assign each method a principal type.

However, the language of Bourdoncle and Metz differs from our proposal in the treatment of dynamic dispatch. In their language, every object is wrapped by a tag encoding its type, and every method name has a single entry point which dispatches according to these tags. By contrast, our approach relies on the underlying machinery of .NET to perform dynamic dispatch, and we resolve at compile-time which calling sequence is to be used to invoke a particular method.

None the less, it would be interesting to push this connection further. In particular, we have already seen examples where method constraints escaped into type schemes when insufficient type information was available at compile-time to resolve a call. This suggest the witness passing of our implementation could be used to simulate the dynamic dispatch of multi-method based implementations.

6.3 Constraint handling rules

It is clear that the type-class design space is complicated. Stuckey, Sulzmann and Glynn have proposed *Constraint Handling Rules* as a formal framework for specifying and reasoning about type-class systems [8]. The advantage is that properties like ambiguity and coherence may be expressed in a single uniform way, rather than having to be re-expressed for each extension.

We have not yet worked out whether our types system can be expressed in their framwork.

Acknowledgments

We thank Don Syme, Nick Benton, Andrew Kennedy, and the anonymous reviewers, for many helpful comments.

References

[1] Benton, N., and Kennedy, A. Interlanguage working without tears: Blending SML with Java. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99), Paris, France* (Sept. 1999), ACM Press, pp. 126–137.

[2] Bourdoncle, F., and Merz, S. Type-checking higher-order polymorphic multi-methods. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris, France* (Jan. 1997), ACM Press, pp. 302–315.

[3] Camarao, C., and Figueiredo, L. Type inference for overloading without restrictions, declarations or annotations. In *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan* (Nov. 1999), LNCS 1722, Springer-Verlag, pp. 37–52.

[4] DeMichiel, L. G., and Gabriel, R. P. The Common Lisp Object System overview. In *European Conference on Object-Oriented Programming (ECOOP'87), Pairs, France* (June 1987), LNCS 276, Springer-Verlag, pp. 151–170.

[5] Duggan, D., Cormack, G., and Ophel, J. Kinded type inference for parametric overloading. *Acta Informatica 33*, 1 (1996), 21–68.

[6] Finne, S., Leijen, D., Meijer, E., and Peyton Jones, S. Calling Hell from Heaven and Heaven from Hell. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99), Paris, France* (Sept. 1999), ACM Press, pp. 114–125.

[7] Gaster, B. R., and Jones, M. P. A polymorphic type system for extensible records and variants. Tech. Rep. NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, Nov. 1996.

[8] Glynn, K., Stuckey, P., and Sulzmann, M. Type classes and constraint handling rules. In *First Workshop on Rule-Based Constraint Reasoning and Programming* (July 2000).

[9] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification.* Addison-Wesley, 1996.

[10] Jeffery, D., Dowd, T., and Somogyi, Z. MCORBA: a CORBA binding for Mercury. In *First International Workshop on Practical Aspects of Declarative Languages (PADL'99), San Antonio, Texas* (1999), LNCS 1551, Springer-Verlag, pp. 211–227.

[11] Jones, M., and Peyton Jones, S. L. Lightweight extensible records for Haskell. In *Proceedings of the 1999 Haskell Workshop, Paris, France* (Oct. 1999). Available as Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht.

[12] Jones, M. P. *Qualified Types: Theory and Practice.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[13] Jones, M. P. Simplifying and improving Qualified Types. Tech. Rep. YALEU/DCS/RR-1040, Computer Science Department, Yale University, New Haven, Connecticut, June 1994. Shorter version appears in FPCA'95, 160–169.

[14] Jones, M. P. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming, (ESOP 2000), Berlin, Germany* (Mar. 2000), LNCS 1782, Springer-Verlag.

[15] Lewis, J., Shields, M., Meijer, E., and Launchbury, J. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00), Boston, Massachusetts* (Jan. 2000), ACM Press, pp. 108–118.

[16] Meijer, E., and Finne, S. Lambada: Haskell as a better Java. In *Proceedings of the 2000 Haskell Workshop, Montreal, Canada* (Sept. 2000). Available as

Nottingham University Department of Computer Science Technical Report NOTTCS-TR-00-1.

[17] Microsoft Corp. Draft C# language specification. Working document for ECMA TC39/TG2, available at `http://msdn.microsoft.com/net/ECMA/WD05-Review.pdf`, Mar. 2001.

[18] Microsoft Corp. Draft Common Language Infrastructure (CLI). Working document for ECMA TC39/TG3, available at `http://msdn.microsoft.com/net/ECMA/Partition_`x`.pdf` for $x \in \{\texttt{I\_Architecture}, \texttt{II\_Metadata}, \texttt{III\_CIL}, \texttt{IV\_Library}, \texttt{V\_Annexes}\}$, 2001.

[19] Mitchell, J. C. *Foundations of Programming Languages.* The MIT Press, 1996.

[20] Odersky, M., Wadler, P., and Wehr, M. A second look at overloading. In *Proceedings of the ACM SIGPLAN Conference on Functional Programming Languages and Computer Architecture (FPCA'95), La Jolla, California* (1995).

[21] Peyton Jones, S. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction, Marktoberdorf Summer School 2000* (2001), R. Steinbrueggen, Ed., NATO ASI Series, IOS Press.

[22] Peyton Jones, S., Jones, M., and Meijer, E. Type classes: exploring the design space. In *Proceedings of the 1997 Haskell Workshop, Amsterdam, The Netherlands* (June 1997).

[23] Shields, M., and Meijer, E. Type-indexed rows. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01), London, England* (Jan. 2001), ACM Press, pp. 261–275.

[24] Shields, M., and Peyton Jones, S. Object-oriented style overloading for Haskell. Tech. rep., Microsoft Research, Cambridge, Aug. 2001. Available at `http://www.cse.ogi.edu/~mbs/pub/overloading/overloading.ps`.