# Object-Oriented Style Overloading for Haskell

Mark Shields Simon Peyton Jones
Microsoft Research Cambridge
{markshie, simonpj}@microsoft.com

## 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 explore various encodings of object-oriented classes into Haskell, demonstrate precisely where Haskell's existing type class system is unsatisfactory, and propose two refinements. We proceed in three stages. Firstly, we discuss various ways of accommodating *sub-typing*; we conclude that a simple encoding using Haskell classes is better for our purpose than a more substantial language extension. Second, we introduce a new notion of *closed* class, and show how this enables *improvement* of constraints beyond what is possible in Haskell. Closed classes make it easy to encode the truely ad hoc overloading of object-oriented methods without the need for name mangling or gratuitous type annotations. Thirdly, we allow *overlapping instances*, and define what it means for one instance to be better than another. This mechanism will turn out to mimic the rather complex overloading resolution rules used by object-oriented languages to select the *most-specific* method at a call site.

In the Appendix, we present type checking and inference rules, as well as details of constraint entailment and simplification. However, this workshop paper is somewhat exploratory: the design may shift once we gain experience with an implementation, and we have not devoted any time to showing any formal properties of our system.

## 1 The problem

The purpose of this paper is to make it easy to import libraries from Java[9] or .NET[21], 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.

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 polymor-

This is a greatly extended version of the paper which appeared in the Workshop on Multi-Language Infrastructure and Interoperability (BABEL'01), Florence, Italy, 2001.

phism 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 Section 5. The Appendix 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* [13]. 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 [18]; extensible records [7] [12]; and type-indexed rows [28]. 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 $C^\sharp$ [20] 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 [19]. 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[1]. 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.

[1] A value of type `IO t` is a computation that may perform some side effects before returning a result of type `t`. See [26] for a tutorial.

## 3 Subtyping

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

```
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` [23]. (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`.

### 3.1 Phantom types

One solution is to use so-called "phantom types" [6] [17], thus:

```
newtype C a   -- Note the
newtype D a   -- type parameter

opC :: Int -> C a -> IO Int
opD :: C (D a) -> IO Bool
```

The type `(C a)` is a subtype of the $C^\sharp$ type `C`. The type variable "`a`" is a kind of "hole" that can be filled in by a more specific type. A value that is certainly of class `C` (and not a subtype thereof) has type `C Void`, where the hole is filled in by the arbitrary but fixed type `Void`. A value that is certainly of class `D` (and not a subtype thereof) has type `C (D Void)`; here `C`'s hole has been filled in by `D Void`.

The type of `opC` says that it can accept a value of type `C a`, for any type `a`. In particular, it can accept a value of type `(C Void)`, or of type `(C (D Void))`, and so on. In this way, we can use ordinary parametric polymorphism to encode subtyping polymorphism.

This is a neat trick, but it has several disadvantages. First, types can become large, and types show up in type error messages, so large types are bad. Second, it is somewhat confusing that `C` is a *type constructor* rather than a *type*, and that an instance of class `D` has Haskell type `C (D Void)`. Most seriously, the scheme breaks down when $C^\sharp$ or Java interface types are involved, because classes may now have multiple supertypes. For example, suppose `D` implements interface `I` as well as inheriting from `C`:

```
interface I {
  C opI( bool x )
}

class D : C, I {     /* D extends C and I */
  bool opD() { ... }
  C opI( bool x ) { ... }
}
```

Should an instance of class `D` have Haskell type `I (C (D Void))` or `C (I (D Void))`? But neither is compatible with methods from both `C` and `I`:

```
opC :: Int  -> C a -> IO Int
opI :: Bool -> I a -> IO (C Void)
```

The second argument cannot have both `C` and `I` as its outermost type constructor; the phantom-type trick fails.

### 3.2 The class-per-class mapping

Since we are modelling $C^\sharp$ classes, another obvious possibility would be to model each $C^\sharp$ as a Haskell *type class*, rather than as a Haskell *type*, thus:

```
class C c where
  opC :: Int -> c -> IO Int

class C d => D d where
  opD :: d -> IO Bool
```

Now `opC` has the type

```
opC :: C c => Int -> c -> IO Int
```

Since `C` is a (Haskell) superclass of `D`, any type that is an instance of class `D` is also an instance of `C`, and hence can have `opC` applied to it. So the subtyping "comes for free" — and multiple supertyping works fine too.

On closer inspection, however, the attraction is only superficial:

**No instances.** What types, if any, are *instances* of the type classes `C` and `D`? We can sidestep this question altogether by giving an existential type to functions that return an object:

```
newC :: Int -> (∃c. C c => c)
```

That is, `newC` returns a value of some type `c`, where all we know about `c` is that it lies in class `C`. This is a fine approach, and one that has been used in Mercury [11]. Unfortunately, though Haskell does support existential quantification, it does not allow existentials to be used "anonymously." Instead, a new data constructor must be introduced for each class[2]:

```
data C = forall c . C c => C c
newC :: Int -> C
```

Now we have lost the subtyping on object types. Another possibility is to introduce a new *type* as well as a new type class, as we discuss in Section 3.3.

[2]Haskell spells "`exists`" as "`forall`" in this situation!

**Bogus dictionaries.** Consider the function `opC` again. A Haskell function of type `C c => Int -> c -> IO Int` takes a run-time witness argument corresponding to the constraint `C c`. Typically, this argument contains a tuple of functions, one for each method in the class. But in fact `opC` should dispatch through the vector-table attached to its second argument, not through a separately-passed method suite. So the run-time witness passing is unnecessary; and the implementation of `opC` is not what one would usually expect.

(In addition, overloading is still problematic. Object-oriented languages allow a single method name to be used independently in different classes, and for distinctly-typed methods within a single class. Haskell permits neither of these things. But we will address that in Section 4.)

### 3.3 Encoding subtype constraints using classes

Though the "class-per-class" idea does not work well, a mild variant works much better. 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.

### 3.4 Full-blown subtype constraints

Attractive though the `Sub`$X$ approach is, it is only capable of encoding a limited variant of subtyping. In particular, the constraint `SubC c` encodes the subtype relationship of the form `c :<: C`, where "`:<:`" means "is a subtype of", *and `C` is a fixed type.* It cannot encode subtype relationships like `C :<: c`, where `c` is an arbitrary type, or type variable, nor structural subtypes such as `(C->a) :<: (b->D)`. The former, for example, would be required to express functions with a covariant result type:

$$\forall \texttt{a . A :<: a . Int -> a}$$

Few object oriented languages require such types, but they may in the future. To support covariance, one could try to add a second class, `SupC`, for each type, but that is not enough. For example, consider the type

```
(SubC c, SupC c) => c -> IO Int
```

One would have to add subtype-specific-rules to allow this type to be simplified to

```
C -> IO Int
```

To understand what would be involved, we explored in detail the technical machinery necessary to support fully-fledged structural sub-typing constraints. We extended Haskell's type system with an extra form of constraint, $\tau_1 :<: \tau_2$, where $\tau_1$ and $\tau_2$ are arbitrary types, along with suitable simplification rules. Using these constraints our running example is rendered like this:

```
newtype C
newtype I
newtype D :<: C,I

opC :: (c :<: C) => Int  -> c -> IO Int
opI :: (i :<: I) => Bool -> i -> IO C
opD :: (d :<: D) =>         d -> IO Bool
```

However, the underpinning machinery turned out to be quite substantual, as we show in Appendix A.9. Even the type checking rules (let alone type inference) become much more complicated. This additional complication is not justified by our goal of importing Java or .NET libraries, since they

do not require covariance. Nor does the extra complexity buy the Haskell programmer much, because we did not go as far as to allow the Haskell programmer to introduce new subtypes (using, say, extensible sums or products).

Another approach, used by O'Haskell [24], is to introduce implicit subtyping at every application, but forbid subtyping constraints from appearing within type schemes. This has the advantage of hiding most of the machinery of subtyping from the programmer, who never sees subtype constraints. However, it requires residual subtype constraints accumulated during type inference to be simplified to either `true` or `false`, which in turn destroys completeness of type inference.

Our conclusion, painfully drawn, is that there is a very sharp knee in the power-to-weight-ratio curve: fully-fledged subtyping is a wondrous thing, but comes at a heavy cost. The encoding of Section 3.3 gets us just far enough to encode the type systems we are interested in, and at essentially zero cost, so we adopt that solution for the rest of this paper.

## 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[3]:

```
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 $C^\sharp$. For example, we can write[4]:

```
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 [25]. 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:

[3] 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.

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

```
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` 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" [14]. 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 [27]. An elaboration of type classes, called *functional dependencies*, does support improvement [15]; 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 8).

### 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[5]:

```
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 $\texttt{m} :: \forall\alpha\ .\ (\texttt{m} :: \alpha)\texttt{=>}\alpha$.) 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:

```
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"

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

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, as we shall see, the combination of overlap with multiple arguments, and polymorphism, is rather subtle. A key contribution of this 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 Type inference

Here is the story so far:

**Subtyping:** use an encoding into type classes.

**Ad-hoc overloading:** add the notion of a `closed` class, plus the syntactic sugar of method constraints.

**Overlap:** allow overlapping `instance` declarations.

Type systems are usually described formally in two stages. First, one gives the *type checking rules* that explain how to check whether a typing derivation for a program is correct. Second, one gives a *type inference algorithm* that takes a program and infers a valid typing derivation for it.

In Appendix A.3 we give the type checking rules for our proposed extensions. The differences from standard Haskell, lacking `closed` classes and overlapping `instance` declarations, are not great. Our proposed extensions offer quite a range of choices for the type inference process, however. In the rest of this section we offer an informal description of these choices; Appendix A.5 presents the algorithm formally.

A type inference algorithm tries to infer a typing derivation for the program. It may succeed, in which case the program is definitely well typed. It may fail because the program has no typing derivation. A *complete* type inference algorithm always succeeds if a valid typing derivation exists.

Type inference for Haskell is already incomplete, because of polymorphic recursion. In Haskell, a function may call itself at its polymorphic type provided the programmer supplies a type signature for the function. In the absence of a signature, type inference will fail, even though there is a valid typing derivation for the program. The inference algorithm we describe below is incomplete in other ways, a design choice we shall discuss in Section 5.7.

## 5.1 Type inference for Haskell

We begin with a sketch of type inference for an ordinary Haskell function involving qualified types:

```
palin :: Eq e => [e] -> Bool
palin = \xs -> xs == reverse xs
```

Here, `(==)` has type `Eq a => a -> a -> Bool`. When the compiler performs type inference on the right hand side of `palin`, it emerges with (a) the type of the RHS, `[e] -> Bool`, and (b) a set of constraints, in this case the singleton set {`Eq [e]`}. The contraint arises from the application of the overloaded function `(==)`.

What happens next depends on whether the programmer supplies a type signature for the function. If so, as in this case, the compiler must check (a) that the type of the RHS matches the part after the `=>` in the signature (it does), and (b) that the constraints needed by the RHS (`Eq [e]`) can be deduced from the constraints in the signature (`Eq e`)? Or, put the other round, does `Eq e` *entail* `Eq [e]`? More concretely, can we construct a dictionary for `Eq [e]` from a dictionary for `Eq e`? Yes, we can, using the (global) instance declaration to *discharge* the constraint `Eq [e]`:

```
instance Eq a => Eq [a] where ...
```

If the programmer did not supply a type signature, the compiler can abstract over the constraints, to infer a type

```
palin :: Eq [e] => [e] -> Bool
```

Alternatively, it may also choose to *simplify* the set of constraints, to get an equivalent but simpler set, and the abstract over the simpler set, to get

```
palin :: Eq e => [e] -> Bool
```

Notice that these two ways of typing `palin` will lead to two distinct valid typing derivations for the program.

Remember:

- A type signature forces an entailment check.
- Simplification can be done at any time, yielding an equivalent set of constraints.

## 5.2 Coherence

A single program may have many different typing derivations. For example, consider the expression

```
let id x = x in id 'c'
```

One derivation might give `id` the type `Int->Int`, while another might give it the type ∀`a. a->a`. But it does not matter which derivation we choose; the meaning of the program is independent of the derivation.

Where type classes are involved, however, a typing derivation embodies choices about which instance declaration to use to discharge each type constraint. In the absence of overlapping instances, there is a unique way of constructing (say) a dictionary for `Eq [a]` from a dictionary for `Eq a`. Hence, it is relatively easy to show that, although there may be many valid typing derivations for one program, they all give the same answer when run. We say that the type system is *coherent*. Coherence is a vital property, because it ensures that *the meaning of the program is independent of the details of the typing algorithm.*

## 5.3 Type inference with improvement and overlap

The type inference process is complicated by the addition of improvement, and overlapping instances. The key component is the *simplifier*, which simplifes sets of constraints.

Suppose that the simplifer is considering a particular constraint, the *target constraint*. There are four steps involved:

**Identify the *candidate instances*;** that is, the `instance` declarations that match the target constraint (Section 5.3.1). There may be more than one candidate, either because of overlapping instances, or because the call site has too little type context to distinguish among candidate instances.

**Perform improvement.** If all the candidates agree about how to instantiate a type variable in the target method constraint, then instantiate it (Section 5.3.2). For example, suppose the target constraint is `m :: a -> b`, and the candidate instances are `m :: Int -> Int` and `m :: Int -> Char`. They both agree that the first argument must be `Int`, so we can instantiate `a` to `Int`.

**Identify the best candidate,** provided that a unique best candidate exists (Section 5.3.3).

**Check that this candidate matches the target.** In particular, we must check that choosing it would not instantiate any type variables in the target constraint (Section 5.3.4).

We discuss each of these steps in more detail.

### 5.3.1 Identify the candidates

Consider type-checking the following declarations:

```
instance null :: [a]  -> Bool where ....
instance null :: (Float,Float) -> Bool where ....

foo (x:zs) = (ord x, null zs)
```

From the pattern-match on the LHS of `foo` we can see that `zs` must be a list, and from the expression `ord x` we can see that the element type must be `Char` (assuming `ord :: Char -> Int`). So the method constraint that arises from the call to `null` is `null :: [Char] -> t` where `t` is a fresh type variable. Now consider matching this method constraint against the two instance declarations. The second definitely does not match, but the first does. Notice, though, that to make it match, we must instantiate both the instance declaration (instantiating `a` to `Char`) and the target method constraint (instantiating `t` to `Bool`). In short, *to identify candidate instances, we begin by unifying the target method constraint with the instance types.*

However, it is not enough to treat any `instance` that unifies as a candidate. Consider type-checking the following declaration of `g1`:

```
newtype A; class SubA a; instance SubA a
newtype C; class SubC c; instance SubC C

instance m :: SubA a => a -> Int where ...
instance m :: SubC c => c -> Int where ...

g1 c = m (c::C) + (1::Int)
```

The call to `m` gives rise to a method constraint `(m :: C -> Int)`. What are the candidate `m`-instances? If we ignore the contexts, it seems that both `instance` declarations match, by instantiating `a` or `c` to `C`. But their contexts tell us that the first of these apparent matches is spurious; the first `instance` declaration can only match if `a` is `C`, and hence if `SubA C`, which does not hold.

But wait a minute! `SubA C` might not hold *where `g1` is defined*, but it might hold at the *call site for `g1`*. In principle, we might defer the choice, and instead infer the type:

```
g1 :: (m :: C->Int) => C -> Int
```

But this is not what Haskell does. Consider:

```
foo x = x + (1::T)
```

If `T` is not an instance of class `Num` we do not infer the type

```
foo :: Num T => T -> T
```

Instead, the compiler complains that there is no instance for `Num T`. Our conclusion is this: *it is legitimate to use the absence of an instance as a reason to declare unsatisfiability of a ground constraint.*

More precisely, then, an `instance` declaration is a candidate for a target method constraint if

(a) the type after the "`=>`" unifies with the target constraint, and

(b) the context of the instantiated `instance` *may be satisfiable.*

We return to the question of satisfiability in Section 5.4.

#### 5.3.2 Improvement

At this point, we have a set of candidate instances, and we can use them to perform *improvement*, as we sketched in Section 4.2. In that section we said that if there was just one candidate, we could use it to instantiate type variables in the target constraint, if the class concerned is closed. Method constraints are implicitly closed.

However, in general there may be several candidate instances. Can we do any improvement? Yes, we can: if all the candidates agree that a particular type variable in the target method constraint must be instantiated in the same way, then (again assuming that methods are closed) we can safely instantiate it. Here is an example:

```
instance n :: Int -> Char -> (Char,Int)
instance n :: Int -> Bool -> (Bool,Int)
instance n :: Char -> Bool

h x = n (3::Int) x
```

The target constraint, garnered from the RHS of `h` is `n :: Int -> r -> s`. The first two instances are candiates for this target; the third is not. The first two instances agree that the result must be a pair, and that the second component of the pair is `Int`. So we can improve `s` to `(s1,Int)` where `s1` is a fresh type variables. The inferred type for `h` is:

```
h :: (n :: Int -> r -> (s1,Int)) => r -> (s1,Int)
```

We could, in principle, go a little further. It would also be legitimate to spot that both candidate instances of `n` share a common pattern, namely that the type of the second argument is the same as the first component of the result, and thereby infer a more specific type for `h`:

```
h :: (n :: Int -> r -> (r,Int)) => r -> (r,Int)
```

However, it is significantly harder to spot this kind of common pattern, involving finding the least common generalisation of a set of types, so we choose not to try. Interestingly, this operation shows up in System CT, though in a different way (see Section 8).

To summarise, our plan is to treat method constraints as closed, and to use this information to perform improvement, thereby obtaining earlier commitment and less ambiguity.

#### 5.3.3 Finding the best candidate

Let us return to the main example of the previous section, and consider the function `g2`:

```
class SubA a
class SubA b => SubB b
class SubC c

instance m :: SubA a => a -> Int where ...
instance m :: SubB b => b -> Int where ...
instance m :: SubC c => c -> Int where ...

g2 :: B -> Int
g2 b = m b + 1
```

A simple unsatisfiability test will reject the third instance for `m`, but that still leaves two candidates:

```
instance m :: SubA a => a -> Int where ...
instance m :: SubB b => b -> Int where ...
```

Which should we choose? You may say this is a bad situation, but object-oriented programs do this kind of thing a lot: their answer is "choose the most specific". In this case, the second is more specific than the first, so we choose it. How can we make precise this notion of "more specific"?

We define a partial order over the `instance` declarations, independent of any particular target method constraint. If the set of candidate instances (Section 5.3.1) contains a least member with respect to the partial order, that is the most specific candidate.

The partial order is defined like this. An instance `m :: C1 => T1` is more specific than `m2 :: C2 => T2`, if (a) we can instantiate the latter instance to get `C2' => T1`, and (b) from `C1` we can deduce `C2'`. In our example, instantiate `b->Int` to `a->Int` (by substituting `b` for `a`) and now we can certainly deduce `SubA b` from `SubB b`. All this is formalised in Appendix A.4. The main point here is that we must take contexts into account when defining the partial order.

#### 5.3.4 Checking the match

Now we have identified the best candidate. But the game is not quite over. Consider another example:

```
instance ppr :: Show a => a -> String where ...
instance ppr ::      [Char] -> String where ...
```

The intent here is "use the second instance on arguments of type `[Char]`, and the first instance otherwise". Now consider this definition:

```
sc c = ppr [c,c,c]
```

Here, the method constraint will be `ppr :: [s] -> t`. Both instances are candidates, and improvement will fix `t` to be `String`. Now undoubtedly, the second instance is the more specific. Should we therfore choose it, and fix `s` to be `Char`? Certainly not! The programmer might be surprised to find that `sc` is not applicable to a `Bool`, for example.

The point here is that *only improvement may instantiate type variables in the target constraint.* Improvement gives us `ppr :: [s] -> String`, and that is as far as we can go. So we cannot commit to either `instance` at this point. Instead,

we must abstract over the method constraint, giving

```
sc :: (ppr :: [s]->String) => s -> String
```

At one call site of `sc` we might instantiate `s` to (say) `Int`, in which case we will dischage the `ppr` constraint with the first instance. At another call site we might instantiate `s` with `Char`, in which case we discharge the `ppr` constraint with the second instance. It is simply premature to discharge it here.

### 5.4 Satisfiability

The test for candidature (Section 5.3.1) depends in turn on a test for *satisfiability*. There is a range of possible design choices, for the satisfiability decision. All that is necessary for soundness is that we reject no valid candidate. Within this constraint, a more refined test will reject more candidates but may be more expensive. A less refined test might be cheaper, but risks rejecting a program when it is clearly well typed.

For example, recall our example `g1`, but this time annotated by a type signature:

```
newtype A; class SubA a; instance SubA a
newtype C; class SubC c; instance SubC C

instance m :: SubA a => a -> Int where ...
instance m :: SubC c => c -> Int where ...

g1 :: C -> Int
g1 c = m c + 1
```

At one extreme, we could use the vacuous test that rejected no candidates (i.e. declared all constraints to be satisfiable); when doing type inference for `g1`, we would be unable to make a unique choice between the two `m`-instances to discharge the constraint `m :: C->Int`, and hence we would reject the definition. While rejection is sound, it would be surprising and unwelcome behaviour, because it is plain as a pikestaff that `g1` satisfies the specified type signature.

So a reasonable compiler will certainly look at the context of a candidate. Even then, however, it can choose how hard to work to identify unsatisfiable contexts. Consider this variant of an earlier example:

```
instance op :: (Int,  Int)  -> Int  where ...
instance op :: (Bool, Bool) -> Bool where ...

f4 x = op (x+x, x)
```

From `f4`'s right hand side we get the two constraints `(op :: (a,a) -> b, Num a)`; the first arises from the use of `op`, and the second from the use of `(+)`. Now, we can reason that the second instance of `op` does not match, because that would imply that `a` is `Bool` and hence we would require `Num Bool` — and `Bool` is not an instance of class `Num`. So again, exactly one instance matches, but this time the proof of unsatisfiability is a bit more elaborate, because it involves considering both constraints together.

Our point is this: there a spectrum of design choices in the satisfiability test, but something relatively simple will catch the cases we care about.

### 5.5 Termination

Consider the following `instance` declaration:

```
instance Eq [[a]] => Eq [a] where ...
```

If it were legal, this declaration could send the constraint simplifier into an infinite loop. Suppose the simplifier needs to satisfy `Eq [t]`; since this constraint matches the head (*i.e.*, r.h.s.) of the instance declaration, it will use the `instance` to discharge the constraint, giving the new constraint `Eq [[t]]`. And so on. To ensure termination we must ensure that a constraint in the context (*i.e.*, l.h.s.) of the `instance` declaration is strictly simpler than that in the head. Haskell ensures this by requiring that constraints in the context mention only simple type variables, while the head must mention a type constructor.

Improvement complicates matters, however. Suppose we have a `closed` class `C`, and just one instance declaration for `C`:

```
instance C a => C [a] where ...
```

Suppose the simplifier is faced with the constraint `C b`, where `b` is a type variable. Since there is only one instance declaration for `C`, we can improve `b` to `[b1]`, where `b1` is a fresh type variable, and then use the `instance` declaration to discharge `C [b1]`. But now we are left with the constraint `C b1` where `b1` is a type variable, and the process repeats.

Our solution is to adopt the following syntactic restrictions:

1) A `closed` class cannot be a superclass of another class.

2) The context of an `instance` declaration must not mention a `closed` class.

### 5.6 Ambiguity

Sometimes the type inference algorithm may be unable to make a unique choice of which instance declaration to use to discharge a constraint. Our guiding principle is that, under these circumstances, the constraint simplifier must refrain from making arbitrary choices.

Ambiguity may arise because the constraint matches no `instance` declaration (such as the constraints `(Read a, Show a)` arising from `show (read x)`), or because it matches more than one candidate but no candidate is better than all the others.

Constraints that cannot be uniquely discharged are either generalised or floated outwards, in the hope that the extra context available at the call site, or further "up" the expression, respectively, will resolve the choice. If the constraint cannot be resolved by the time the constraint emerges at the top level of the module, it is reported as ambiguous; the programmer must add extra type information to guide the inference algorithm's choice.

### 5.7 Incompleteness

As we mentioned earlier, type inference is incomplete. There is a choice of improvement algorithms; a cheap one may fail

where an expensive one succeeds.

For example, if we are prepared to look at constraints *together*, we can eliminate method constraints more eagerly. For example, consider:

```
instance foo :: Int  -> Int where ...
instance foo :: Bool -> Int where ...

instance baz :: Int  -> Int where ...
instance baz :: Char -> Int where ...

undefined :: a
undefined = undefined

w :: Int
w = (\x -> foo x + baz x) undefined
```

Arising from the right-hand side of `w` we will get the method constraints `(foo :: a -> Int, baz :: a -> Int)`. If we take these constraints one at a time, there is no way to decide which to choose. Hence, type inference will reject the program, complaining that the annotated type for `w` cannot be shown to be an instance of its inferred type:

```
w :: (foo :: a -> Int, baz :: a -> Int) => Int
```

But if we take these constraints together, we can see that the only consistent choice is to choose the first instance of `foo` and `baz`, because they both need their first argument to be of type `Int`. Hence type inference succeeds. Of course, the difficulty with this test is that it is (potentially) much more expensive to compute.

Does this incompleteness matter? One might say "no, provided the error message explains where to add a type annotation to guide the inference process", because then the programmer can simply add a suitable type signature. In the above example, the programmer need simply annotate `x` with type `Int`. The problem is that it becomes harder to say what is a valid Haskell program. Perhaps a particular program will typecheck fine on one implementation of Haskell, while being rejected by another.

This sounds unpleasant. Yet, to formalise the notion of a Haskell program that is accepted by every implementation one must give a type inference algorithm, or something equivalent. That in turn is undesirable.

In the end, it is a matter of taste. There are many interesting type system developments that stray into territory where type inference is incomplete. Should we reject such developments? Should we accept incompleteness? Should we specify the weakest acceptable inference algorithm? Food for thought!

### 5.8 Summary

Obtaining a system that is both coherent and expressive is a surprisingly subtle business. Candidates must be identified using two-way matching (unification), but without yet committing to any particular instance (Section 5.3.1). Information the candidates agree on (and no more) may be applied to the target constraint (Section 5.3.2). A unique best candidate can then be identified (if one exists) but it can only be used if it matches the target method constraint using one-way matching (Section 5.3.3 and 5.3.4).

We make all these things precise in the Appendix. Meanwhile, one might reasonably ask "does the programmer need to understand all this to use the system". Fortunately, the answer is "no". Firstly, because the system is coherent, the details of the typing algorithm do not affect the meaning of the program: either the program is rejected, or it is typed with a unique meaning. Secondly, our framework supports a range of "cleverness", as the numerous design choices above attest. One could leave out improvement altogether, or have a degnerately incomplete unsatisfiability test, and still have a sound system. The effect would be that the system would more often reject the program saying "I cannot tell which of the following `m`-instances you intended to use". The programmer can easily solve this problem by adding type annotations. The only thing that does depend on the typing algorithm is the exact boundary between when the system will make a unique choice and when it will reject the program.

## 6 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.

### 6.1 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

```
/* 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

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

where the `Click` interface is defined thus:

```
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... }
}
```

```
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

($C^\sharp$ 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() }
}
```

```
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

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.

### 6.2 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.

## 7 Futher directions

In this section we collect random thoughts that don't seem to belong in the main thread.

### 7.1 Haskell records

Our approach to ad-hoc overloading alows us to steal the encoding of records used by System O [25].

In Haskell, every record type must use distinct field labels. For example, the following text is illegal in Haskell 98:

```
data V2 = V2 { x,y   :: Float }
data V3 = V3 { x,y,z :: Float }
```

The two records cannot share the common fields `x` and `y`. With our extension, as in System O, we could redeclare them like this:

```
data V2 = V2 { v2_x,v2_y      :: Float }
data V3 = V3 { v3_x,v3_y,v3_z :: Float }

instance x :: V2 -> Float where x = v2_x
instance y :: V2 -> Float where y = v2_y
instance x :: V3 -> Float where x = v3_x
instance y :: V3 -> Float where y = v3_y
instance z :: V3 -> Float where z = v3_z
```

Indeed, one could imagine these instance declarations being automatically generated, just as the selector functions themselves are, so that the field names `v2_x` etc were never exposed to the programmer. Once this is done, we can write examples like this:

```
length :: V2 -> Float
length v = x v + y v
```

There's a problem with record update, however. A heavier encoding might solve it (adding a "set" method as well as a "get" function), but it makes the story less compelling.

### 7.2 Class methods

We can link Haskell's "traditional" type classes with our ad-hoc overloading. For example, the `Num` class is defined like this:

```
class Num a where
  (+) :: a -> a -> a
  negate :: a -> a
  ...etc...
```

This brings into scope the functions

```
(+) :: Num a => a -> a -> a
negate :: Num a => a -> a
...etc...
```

So we cannot use `(+)` for a function of type `Int -> Float -> Float`, since it does not fit the template in the `Num` declaration. But with our new method constraints, we could instead specify that the `class` declaration brings into scope the following functions and `instance` declarations:

```
num1 :: Num a => a -> a -> a  -- (+)
num2 :: Num a => a -> a       -- negate

instance (+) :: Num a => a -> a -> a where
   (+) = num1
instance negate :: Num a => a -> a   where
   negate = num2
```

Now we are free to add new instances for `(+)`, thus:

```
instance (+) :: Int -> Float -> Float where
   (+) = plusIF
```

In effect, the arbitrarily-named `num1`, `num2` etc are simply record selectors very like `v2_x`, `v2_y` in the Section 7.1; the `instance` declarations attach them to their familiar names, `(+)` and `negate`.

# 8 Related work

## 8.1 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.

Lastly, there are CT programs that we cannot express. System CT can type the following program, whereas we cannot:

```
let
    g = let f = <..rhs1..> :: Int->Int
            f = <..rhs2..> :: Char->Char
        in
        f
in
(g 1, g 'c')
```

CT will assign `g` the type

```
g :: {f :: a->a}. a -> a
```

*even though the definitions of `f` are not lexically visible at `g`'s call site.* We do not fully understand this aspect of System CT, but it certainly affects the complexity of the translation into witness-passing form.

## 8.2 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.

### 8.3 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] Henglein, F., and Rehof, J. The complexity of subtype entailment for simple types. In *Proceedings of the Twelth Annual IEEE Symposium on Logic in Computer Science (LICS'97), Warsaw, Poland* (June 1997), IEEE Computer Society Press, pp. 352–361.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] Lassez, J.-. L., Maher, M. J., and Marriott, K. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. M. Kaufnann Publishers, 1988, ch. 15, pp. 587–625.

[17] Leijen, D., and Meijer, E. Domain specific embedded compilers. In *Proceedings of the Second USENIX Conference on Domain-Specific Languages (DSL'99), Austin, Texas* (Oct. 1999), USENIX Association, pp. 109–122. Also appears in *ACM SIGPLAN Notices 35*, 1, (Jan. 2000).

[18] 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.

[19] 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.

[20] 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.

[21] 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 $x \in$ { `I_Architecture`, `II_Metadata`, `III_CIL`, `IV_Library`, `V_Annexes` }, 2001.

[22] Mitchell, J. C. Type inference with simple subtypes. *Journal of Functional Programming 1*, 3 (1991), 245–285.

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

[24] Nordlander, J. Pragmatic subtyping in polymorphic languages. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98), Baltimore, Maryland* (Sept. 1998), ACM Press, pp. 216–227.

[25] 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).

[26] 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.

[27] 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).

[28] 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.

[29] Wadler, P., and Blott, S. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (POPL'89), Austin, Texas* (Jan. 1989), ACM Press, pp. 60–76.

# A Formal Development

In the remainder of the paper we present $\lambda^{\mathsf{O}}$, a small lambda-calculus of type-based overloading. We intend $\lambda^{\mathsf{O}}$ to be the kernel language for Haskell extended with support for OO-style overloading. However, for brevity, we have ellided much of the structure needed for full Haskell: base types, recursive datatypes, higher kinds, recursion, Haskell-style newtypes, case discrimination and higher-ranked polymorphism.

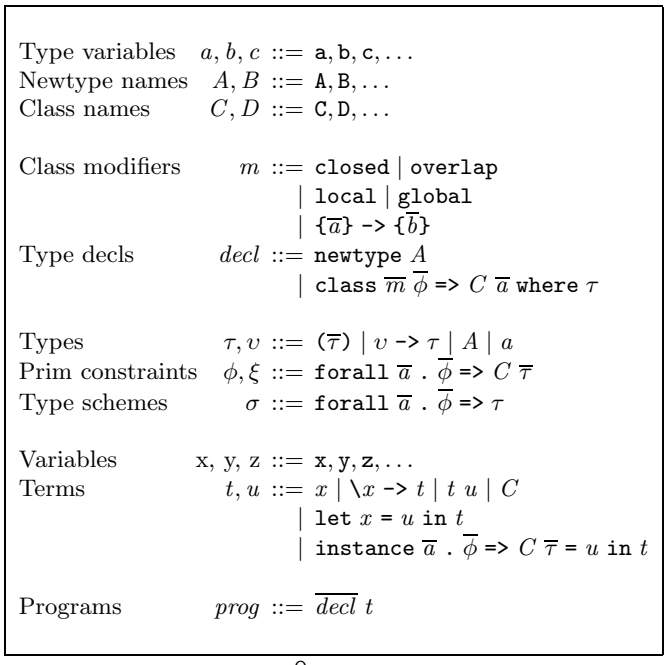


**Figure 4:** Syntax of $\lambda^{\mathsf{O}}$ types and source-level terms

In this workshop paper we won't demonstrate any properties of $\lambda^{\mathsf{O}}$, in particular, soundness and (weak) completeness of type inference with respect to type checking. This is not to say we believe these proofs to be straightforward, or even that we believe the following presentation to be without error! Rather, we would like to first gain some experience with an implementation to check the feasibility of this design before investing a lot of effort in showing the correctness of something which may well change.

We shall present the system in two stages. In the first stage we assume subtyping is implemented by the encoding scheme given in Section 3.3. Though very inexpressive, this approach seems sufficient for encoding the subtyping constraints required for methods from .NET classes. In the second stage, we build subtyping constraints directly into the calculus. This second approach is much more expressive, and supports structural subtyping and covariant methods. However, this comes at the cost of a considerably more complex system.

In a few places, $\lambda^{\mathsf{O}}$ is a little more expressive than required for the subject of this paper. For example, we have included local instance declarations and functional dependencies [15]. This is to demonstrate how easily these features coexist with closed classes.

## A.1 Source Syntax

Figure 4 presents the syntax of type declarations, types, constraints, and source terms.

For brevity we shall exploit a sequence notation throughout. We write $\overline{X}$ to denote a sequence $X_1, \ldots, X_n$ for some unspecified, and possibly zero $n$. We shall systematically abuse

this notation. For example, given $\overline{\tau}$ and $\overline{\upsilon}$, we write $\overline{\tau \coloneqq \upsilon}$ to denote $\tau_1 \coloneqq \upsilon_1, \ldots, \tau_n \coloneqq \upsilon_n$. Notice that this implicitly constrains $\overline{\tau}$ and $\overline{\upsilon}$ to be of the same length. If $\tau$ is a scalar and $\overline{\upsilon}$ a sequence, we write $\overline{\tau \coloneqq \upsilon}$ to denote $\tau \coloneqq \upsilon_1, \ldots, \tau \coloneqq \upsilon_n$. By even further abuse, we write $\Delta \vdash \overline{\tau\ \mathsf{type}}$ to denote $(\Delta \vdash \tau_1\ \mathsf{type}) \wedge \ldots \wedge (\Delta \vdash \tau_n\ \mathsf{type})$. We shall occasionally treat a sequence $\overline{X}$ as the set $\{X_1, \ldots, X_n\}$. The empty sequence is written as $\cdot$.

The type declaration `newtype` $A$ introduces an *uninterpreted type constant* $A$. In $\lambda^{\mathsf{O}}$, newtypes shall be used exclusively to represent .NET class types, and hence a value of type $A$ is a .NET object created by a call to a .NET constructor. There is no way to construct such values within $\lambda^{\mathsf{O}}$ alone. In practice, we should allow newtypes to be assigned a body type, as in Haskell. Newtypes which are intended to denote an external class-type would then be given a uniform body type such as `ObjRef`.

The type declaration `class` $\overline{m}\ \overline{\xi}$ `=>` $C\ \overline{a}$ `where` $\tau$ introduces a *type class* $C$, paramaterized by the type variables $\overline{a}$. Informally, we may think of this as introducing an identifier, $C$, which may be bound to an as yet unspecified number of values at distinct types. Furthermore, we can be sure each such type is an instance of $\tau$ for which the constraints in $\overline{\xi}$ are satisfied. We shall call $\overline{\xi}$ the *superclass constraints* of $C$, even though in stage two such constraints need not be just class constraints.

More precisely, such a declaration introduces a family of primitive *class constraints* of the form $C\ \overline{\upsilon}$. The constraint $C\ \overline{\upsilon}$ arises when $C$ is used within a term at type $[\overline{a \mapsto \upsilon}]\ \tau$. Such a constraint is satisfied when an *instance declaration* has been supplied for $C$ at types $\overline{\upsilon}$. In this case, we say that the instance declaration provides a *witness* for the satisfaction of $C\ \overline{\upsilon}$. In $\lambda^{\mathsf{O}}$, instance declarations belong within terms rather than as top-level declarations, and so shall be discussed shortly.

Class declarations may include modifiers which allow the $\lambda^{\mathsf{O}}$ constraint simplifier to exploit invariants amongst instance declarations. These modifiers may be divided into three groups.

If class $C$ includes the `closed` modifier, then the programmer must ensure all of the instance declarations for $C$ are in scope whenever a constraint involving $C$ may arise. In return for this restriction, the $\lambda^{\mathsf{O}}$ constraint simplifier is able to *improve*, and, in many situations, elliminate altogether, constraints involving $C$. If class $C$ *does not* include the modifier `overlap`, then the programmer must ensure all instance declarations for $C$ are mutually non-unifiable. In return, the simplifier is able to elliminate class constraints more eagerly than if instances declarations were allowed to overlap. The `closed` modifier subsumes the `overlap` modifier.

The modifier `{`$\overline{a}$`} -> {`$\overline{b}$`}` on a class $C$ signals that a *functional dependency* exists between the parameters of $C$ in positions $\overline{a}$ and those in position $\overline{b}$. ($\overline{a} +\!\!+ \overline{b}$ must be a permutation of the type variable parameters for $C$.) In particular, the programmer must ensure that there are no instance declarations for $C$ which agree on types in positions $\overline{a}$ but disagree on those in positions $\overline{b}$. Again, this restriction allows the simplifier to agressively collapse class constraints. We refer the reader to the work of Jones [15] for a full exposition. A class may contain zero, one, or more functional dependencies. If $\overline{a}$ is empty, only a single instance declaration for $C$ is possible. If $\overline{b}$ is empty, the functional dependency is trivial and may be elliminated.

The `local` modifier signals that a class $C$ may have *local instance declarations*, and hence class constraints using $C$ must not be inherited from an outer context. In this way, any local class constraints within a let-bound term will be generalized, and propogate to each specialization point for the let-bound variable. The constraint may thus be satisifed by instance declarations which are otherwise not in the let-binding's scope. Implicit parameters [18] are a degenerate example of local classes with a functional dependency. Dually to `local`, a `global` modifier signals that class constraints using $C$ must be inherited, and thus may never appear within a type scheme. Clearly, it makes no sence for a class to be both `local` and `global`.

For technical reasons, we disallow class declarations whose superclass constraint contains non-trivial instance schemes (see below) or class constraints for closed or overlapping classes.

The language of types includes tuples, functions, newtypes and type variables.

In stage one, $\lambda^{\mathsf{O}}$ primitive constraints are just *instance schemes*. An instance scheme of the form `forall` $\overline{a}$ `.` $\overline{\xi}$ `=>` $C\ \overline{\tau}$ may be thought of as the type of an instance declaration. It is witnessed by a polymorphic function from witnesses to $\overline{[\overline{a \mapsto \upsilon}]\ \xi}$ to a witness for the class constraint $C\ \overline{[\overline{a \mapsto \upsilon}]\ \tau}$. We shall identify the trivial instance scheme `forall` $\cdot$ `.` $\cdot$ `=>` $C\ \overline{\tau}$ with the class constraint $C\ \overline{\tau}$.

Constraints are conjunctions (written as a sequence) of primitive constraints. The empty constraint, $\cdot$, is typically written as `true`. We write `false` to denote some fixed, unsatisfiable primitive constraint.

Type schemes are defined as usual. In common with instance schemes, we identify the trivial type scheme `forall` $\cdot$ `.` $\cdot$ `=>` $\tau$ with the type $\tau$. It is important to distinguish instance schemes, which are inhabited by witness building functions, from type schemes, which are inhabited by ordinary functions.

Terms include the usual abstraction, application and polymorphic let-binding, along with variables and class names. A class declaration such as `class` $\overline{m}\ \overline{\xi}$ `=>` $C\ \overline{a}$ `where` $\tau$ in effect introduces an identifier $C$ with polymorphic type `forall` $\overline{a}$ `.` $C\ \overline{a}$ `=>` $\tau$. In a significant departure from Haskell, $\lambda^{\mathsf{O}}$ terms also include instance declarations. The term `instance` $\overline{a}$ `.` $\overline{\xi}$ `=>` $C\ \overline{\tau}$ `=` $u$ `in` $t$ introduces a new way of satisfying primitive constraints involving $C$. In particular, it introduces, within the context of $t$, a witness, $W$, to the instance scheme `forall` $\overline{a}$ `.` $\overline{\xi}$ `=>` $C\ \overline{\tau}$. Given witnesses to $\overline{[\overline{a \mapsto \upsilon}]\ \xi}$, the witness $W$ produces a witness to $C\ \overline{[\overline{a \mapsto \upsilon}]\ \tau}$ which contains the value $u$. Only classes containing the `local` modifier may have instance declarations which are not at the top-level of the program.

It is straightforward to translate Haskell class and instance declarations into $\lambda^{\mathsf{O}}$. Rather than describe this formally, Figure 5 simply presents an example translation for the

**Haskell Declarations**

```
class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

instance Monad [] where
  return a = [a]
  bind ma f = concat (map f ma)
```

**$\lambda^{\mathsf{O}}$ Declarations**

```
data MonadD m =
  MonadD (forall a . a -> m a)
         (forall a b . m a -> (a -> m b) -> m b)
class Monad m where MonadD m

return :: forall a . Monad m => a -> m a
return = case Monad of MonadD r _ -> r
bind :: forall a b .
          Monad m => m a -> (a -> m b) -> m b
bind = case Monad of MonadD _ b -> b

instance Monad [] =
  MonadD (\a -> [a])
         (\ma f -> concat (map f ma))
```

**Figure 5:** Encoding Haskell class and instance declarations in $\lambda^{\mathsf{O}}$ (extended by datatypes and higher-ranked polymorphism)

`Monad` class. This translation assumes $\lambda^{\mathsf{O}}$ to be augmented by datatypes with polymorphic data constructors.

Our informal presentation adopted some syntactic sugar for method declarations, which we now explain. We assume the first instance declaration of a method `m` introduces the $\lambda^{\mathsf{O}}$ class declaration:

```
class closed Has_m a where a
```

(As mentioned in Section 4.3, it is not feasible to require the programmer to include such a declaration explicitly, since there is no unique module for it to live within.) A method instance declaration such as:

```
instance m :: a :<: A => a -> IO Int
```

is represented as the $\lambda^{\mathsf{O}}$ instance declaration:

```
instance a . a :<: A => Has_m (a -> IO Int) = stub
```

where *stub* stands for whatever stub code is required to marshal the arguments, make the appropriate call, and unmarshal the result.

A variation of this encoding is to make all method constraint classes `global`:

```
class closed global Has_m a where a
```

This would prevent a method constraint from ever appearing within a type scheme, which in turn would force the type inference system to reject any program containing a method call which cannot be resolved by local type context. This encoding may be more palatable to programmers accustomed to the method overloading resolution rules of object-oriented languages.

Witness vars $w ::= \mathsf{w}, \ldots$
Constraint contexts $\Phi, \Xi ::= \overline{w : \phi}$
Witness bindings $B ::= \overline{w = W}$

Witness terms

| | | |
|---|---|---|
| $W ::= w$ | | |
| | $\mid C\ \overline{W}\ T$ | Construct dictionary |
| | $\mid \mathsf{letw}\ B\ \mathsf{in}\ W'$ | Bind witnesses |
| | $\mid \mathsf{Proj}_i^C\ W$ | Extract superclass witness |
| | $\mid \Lambda\overline{a}\ .\ W \mid W\ \overline{\tau}$ | Type abs. and app. |
| | $\mid \lambda\overline{w}\ .\ W \mid W\ \overline{W'}$ | Witness abs. and app. |

Run-time terms

| | | |
|---|---|---|
| $T, U ::= x \mid \lambda x\ .\ T \mid T\ U$ | | |
| | $\mid \mathsf{let}\ x = U\ \mathsf{in}\ T$ | |
| | $\mid \Lambda\overline{a}\ .\ T \mid T\ \overline{\tau}$ | Type abs. and app. |
| | $\mid \mathsf{letw}\ B\ \mathsf{in}\ T$ | Bind witnesses |
| | $\mid \mathsf{Proj}_{\mathsf{V}}^C\ W$ | Extract instance impl. |
| | $\mid \lambda\overline{w}\ .\ T \mid T\ \overline{W}$ | Witness abs. and app. |

**Figure 6:** Syntax of $\lambda^{\mathsf{O}}$ run-time terms

We write $\overline{a} \vdash \tau\ \mathsf{type}$ to denote the judgement that $\tau$ is a well-formed type. In $\lambda^{\mathsf{O}}$, this reduces to checking all free type variables of $\tau$ are within $\overline{a}$. Well-formedness for constraints ($\overline{a} \vdash \overline{\xi}\ \mathsf{constraint}$) and type schemes ($\overline{a} \vdash \sigma\ \mathsf{scheme}$) is defined in the obvious way.

## A.2 Run-time syntax

Every satisfied constraint has a coresponding witness. Polymorphic terms may give rise to constraints which, though satisfiable, are not yet satisfied. Such constraints appear within type schemes, and are propogated by type inferere-nce. When such a constraint is instantiated to be satisfied, the coresponding witness must be conveyed from the point of specialization to the point of use. One way of achieving this is to tranform source programs into a *run-time language* in which these witnesses are passed as implicit arguments to every polymorphic function with constraints. This is achieved using the well-known *dictionary transformation* [29].

It shall be the job of type inference to convert source terms to run-time terms with explicit witness passing. In keeping with modern practice, this translation shall also make all type passing explicit, even though no type information is actually required at run-time.

Figure 6 defines the syntax of these run-time terms and witnesses. We prefer to seperate witnesses from run-time terms, even though they share much structure. This is so that, if $\lambda^{\mathsf{O}}$ were to be extended by recursion, we could be sure that witness calculations always terminate.

Given the declaration `class` $\overline{m}\ \overline{\xi}$ `=>` $C\ \overline{a}$ `where` $\tau$, a witness to $C\ \overline{v}$ is of the form $C\ \overline{W}\ T$, where $\overline{W}$ are witnesses to

$$\boxed{\overline{a} \mid \Phi \mid \Gamma \vdash t : \tau \hookrightarrow T}$$

$$\frac{(\texttt{class}\ \overline{m}\ \Xi\ \texttt{=>}\ C\ \overline{b}\ \texttt{where}\ \tau) \in \mathit{decls} \quad \overline{a} \vdash \overline{v}\ \mathsf{type} \quad \Phi \vdash^{e} C\ \overline{v} \hookrightarrow W}{\overline{a} \mid \Phi \mid \Gamma \vdash C : [\overline{b \mapsto v}]\ \tau \hookrightarrow \mathsf{Proj}^{C}_{\mathsf{V}}\ W}\ \textsc{class}$$

$$\frac{(x : \texttt{forall}\ \overline{b}\ .\ \overline{\xi}\ \texttt{=>}\ \tau) \in \Gamma \quad \overline{a} \vdash \overline{v}\ \mathsf{type} \quad \Phi \vdash^{e} \overline{[b \mapsto v]\ \xi} \hookrightarrow W}{\overline{a} \mid \Phi \mid \Gamma \vdash x : [\overline{b \mapsto v}]\ \tau \hookrightarrow x\ \overline{v}\ \overline{W}}\ \textsc{var}$$

$$\frac{\overline{a} \vdash v\ \mathsf{type} \quad \overline{a} \mid \Phi \mid \Gamma, x : v \vdash t : \tau \hookrightarrow T}{\overline{a} \mid \Phi \mid \Gamma \vdash \backslash x\ \texttt{->}\ t : (v\ \texttt{->}\ \tau) \hookrightarrow \lambda x\ .\ T}\ \textsc{abs}$$

$$\frac{\overline{a} \mid \Phi \mid \Gamma \vdash t : (v\ \texttt{->}\ \tau) \hookrightarrow T \quad \overline{a} \mid \Phi \mid \Gamma \vdash u : v \hookrightarrow U}{\overline{a} \mid \Phi \mid \Gamma \vdash t\ u : \tau \hookrightarrow T\ U}\ \textsc{app}$$

$$\frac{\begin{array}{c} x \in \mathit{fv}(t) \quad \overline{a} \cap \overline{b} = \emptyset \quad \overline{a} +\!\!+ \overline{b} \vdash \Xi\ \mathsf{constraint} \\ \sigma = \texttt{forall}\ \overline{b}\ .\ \mathit{anon}(\Xi)\ \texttt{=>}\ v \quad \mathit{gens}(\Xi) = \Xi \\ \overline{a} +\!\!+ \overline{b} \mid \mathit{inhs}(\Phi) +\!\!+ \Xi \mid \Gamma \vdash u : v \hookrightarrow U \\ \overline{a} \mid \Phi \mid \Gamma, x : \sigma \vdash t : \tau \hookrightarrow T \end{array}}{\begin{array}{l} \overline{a} \mid \Phi \mid \Gamma \vdash \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \tau \hookrightarrow \\ \quad \mathsf{let}\ x = \Lambda \overline{b}\ .\ \lambda \mathit{names}(\Xi)\ .\ U\ \mathsf{in}\ T \end{array}}\ \textsc{let}$$

$$\frac{\begin{array}{c} (\texttt{class}\ \overline{m}\ \overline{\xi'}\ \texttt{=>}\ C\ \overline{c}\ \texttt{where}\ v') \in \mathit{decls} \\ C \in \mathit{fcv}(t) \quad \overline{a} \cap \overline{b} \cap \overline{c} = \emptyset \quad w\ \text{fresh} \\ \Xi = \mathit{named}(\overline{\xi}) \quad \overline{a} +\!\!+ \overline{b} \vdash \Xi\ \mathsf{constraint} \quad \overline{a} +\!\!+ \overline{b} \vdash \overline{v\ \mathsf{type}} \\ \phi = \texttt{forall}\ \overline{b}\ .\ \overline{\xi}\ \texttt{=>}\ C\ \overline{v} \quad \mathit{gens}(\Xi) = \Xi \\ (\Gamma \neq \cdot) \vee (\mathit{inhs}(\Xi) \neq \Xi) \Longrightarrow \texttt{local} \in \overline{m} \\ \mathit{inhs}(\Phi) +\!\!+ \Xi \vdash^{e} \overline{[c \mapsto v]\ \xi'} \hookrightarrow W \\ \overline{a} +\!\!+ \overline{b} \mid \mathit{inhs}(\Phi) +\!\!+ \Xi \mid \Gamma \vdash u : [\overline{c \mapsto v}]\ v' \hookrightarrow U \\ \Phi' = \mathit{extend}(\Phi \mid w : \phi)\ \text{well-defined} \\ \overline{a} \mid \Phi' \mid \Gamma \vdash t : \tau \hookrightarrow T \end{array}}{\begin{array}{l} \overline{a} \mid \Phi \mid \Gamma \vdash \\ \quad \texttt{instance}\ \overline{b}\ .\ \overline{\xi}\ \texttt{=>}\ C\ \overline{v}\ \texttt{=}\ u\ \texttt{in}\ t : \tau \hookrightarrow \\ \quad\quad \mathsf{letw}\ w = \Lambda \overline{b}\ .\ \lambda \mathit{names}(\Xi)\ .\ C\ \overline{W}\ U\ \mathsf{in}\ T \end{array}}\ \textsc{inst}$$

**Figure 7:** Well-typed $\lambda^{\mathsf{O}}$ terms

$$\mathit{inheritable}(\texttt{forall}\ \overline{a}\ .\ \overline{\phi}\ \texttt{=>}\ C\ \overline{\tau}) \Longleftrightarrow \texttt{local} \notin \overline{m}$$
$$\quad \text{where } (\texttt{class}\ \overline{m}\ \overline{\xi}\ \texttt{=>}\ C\ \overline{b}\ \texttt{where}\ v) \in \mathit{decls}$$

$$\mathit{inhs}(\Phi) = \{w : \xi \mid (w : \xi) \in \Phi, \mathit{inheritable}(\xi)\}$$

$$\mathit{generalizable}(\texttt{forall}\ \overline{a}\ .\ \overline{\phi}\ \texttt{=>}\ C\ \overline{\tau}) \Longleftrightarrow \texttt{global} \notin \overline{m}$$
$$\quad \text{where } (\texttt{class}\ \overline{m}\ \overline{\xi}\ \texttt{=>}\ C\ \overline{b}\ \texttt{where}\ v) \in \mathit{decls}$$

$$\mathit{gens}(\Phi) = \{w : \xi \mid (w : \xi) \in \Phi, \mathit{generalizable}(\xi)\}$$

$$\mathit{named}(\overline{\phi}) = \overline{w : \phi} \qquad \text{where } \overline{w} \text{ fresh}$$
$$\mathit{anon}(\overline{w : \phi}) = \overline{\phi}$$
$$\mathit{names}(\overline{w : \phi}) = (\overline{w})$$

$$\mathit{matchclasses}(C\ \overline{\tau}, D\ \overline{v}) \Longleftrightarrow \mathbf{ff} \qquad \text{where } C \neq D$$
$$\mathit{matchclasses}(C\ \overline{\tau}, C\ \overline{v}) \Longleftrightarrow$$
$$\quad \exists(\{\overline{a}\}\ \texttt{->}\ \{\overline{a'}\}) \in (\overline{m} \cup (\{\overline{c}\}\ \texttt{->}\ \{\})) .$$
$$\quad\quad \forall b \in \overline{a} .$$
$$\quad\quad\quad [\overline{c' \mapsto \tau}] \circ [\overline{c \mapsto c'}]\ b = [\overline{c' \mapsto v}] \circ [\overline{c \mapsto c'}]\ b$$
$$\quad\quad\quad \text{where } (\texttt{class}\ \overline{m}\ \overline{\xi}\ \texttt{=>}\ C\ \overline{c}\ \texttt{where}\ v') \in \mathit{decls}$$
$$\quad\quad\quad \text{and } \overline{c'} \text{ fresh}$$

$$\mathit{extend}(\cdot \mid w : \phi) = w : \phi$$
$$\mathit{extend}(w' : \xi, \Xi \mid w : \phi) = \mathit{extend}(\Xi \mid w : \phi)$$
$$\quad \text{where } (\texttt{class}\ \overline{m}\ \overline{\xi'}\ \texttt{=>}\ C\ \overline{c}\ \texttt{where}\ v') \in \mathit{decls}$$
$$\quad \text{and } \texttt{local} \in \overline{m}$$
$$\quad \text{and } \xi = \texttt{forall}\ \overline{b}\ .\ \overline{\xi'}\ \texttt{=>}\ C\ \overline{v}$$
$$\quad \text{and } \phi = \texttt{forall}\ \overline{a}\ .\ \overline{\phi'}\ \texttt{=>}\ C\ \overline{\tau}$$
$$\quad \text{and } \overline{a'}, \overline{b'} \text{ fresh}$$
$$\quad \text{and } \mathit{matchclasses}(C\ \overline{[a \mapsto a']\ \tau}, C\ \overline{[b \mapsto b']\ v})$$
$$\mathit{extend}(w' : \xi, \Xi \mid w : \phi) = \mathit{undefined}$$
$$\quad \text{where } (\texttt{class}\ \overline{m}\ \overline{\xi'}\ \texttt{=>}\ C\ \overline{c}\ \texttt{where}\ v') \in \mathit{decls}$$
$$\quad \text{and } \texttt{overlap} \notin \overline{m} \text{ and } \texttt{closed} \notin \overline{m}$$
$$\quad \text{and } \xi = \texttt{forall}\ \overline{b}\ .\ \overline{\xi'}\ \texttt{=>}\ C\ \overline{v}$$
$$\quad \text{and } \phi = \texttt{forall}\ \overline{a}\ .\ \overline{\phi'}\ \texttt{=>}\ C\ \overline{\tau}$$
$$\quad \text{and } \overline{a'}, \overline{b'} \text{ fresh}$$
$$\quad \text{and } \mathit{mgus}(\emptyset \mid \overline{[a \mapsto a']\ \tau} \coloneqq \overline{[b \mapsto b']\ v}) \neq \emptyset$$
$$\mathit{extend}(w' : \xi, \Xi \mid w : \phi) = w' : \xi, \mathit{extend}(\Xi \mid w : \phi)$$

**Figure 8:** Ancillary definitions for type checking

each primitive constraint in $\overline{[a \mapsto v]\ \xi}$, and $T$ is a run-time value of type $\overline{[a \mapsto v]\ \tau}$. $\mathsf{Proj}^{C}_{i}$ projects the witness $W_i$, and $\mathsf{Proj}^{C}_{\mathsf{V}}$ projects the run-time value $T$.

The remaining witness forms and run-time terms are those of System F, augmented by let (for sharing terms) and letw (for sharing witnesses).

## A.3 Type Checking

Figure 7 presents the rules for deciding the well-typing judgement $\overline{a} \mid \Phi \mid \Gamma \vdash t : \tau \hookrightarrow T$. Here $\overline{a}$ is the set of type variables in scope, and $\Gamma$ the type context, for source term $t$. $\Phi$ is a constraint context, and $T$ is the run-time term to which $t$ is translated.

A constraint context is a sequence of named primitive constraints of the form $w : \phi$. The name $w$ is arbitrarily chosen to represent the witness variable which will be bound to a witness to $\phi$ at run-time. These witness variables shall be introduced within $T$ during type checking. The functions *named* and *anon*, given in Figure 8, mediate between constraints and constraint contexts, while the function *names* returns the tuple of witness names of its argument constraint context.

The type checking rules make use of an entailment judgement $\Phi \vdash^{e} \xi \hookrightarrow W$, which is true when constraint $\xi$ is a logical consequence of the constraint context $\Phi$, and is witnessed by $W$. This will be explained shortly.

Rule CLASS checks that a class identifier $C$ is used at a type for which an instance declaration is available. It produces a run-time term which projects the value from the appropriate

witness.

Rule VAR performs specialization of a polymorphic variable $x$. Any constraint on $x$ must be entailed by $\Phi$. The term $x$ is translated to a run-time term in which both the types to which $x$ is specialized, and the witnesses to $x$'s constraints, are passed as tuples.

Rules ABS and APP are standard.

Rule LET type checks polymorphic terms in the usual way. The let-bound term $u$ is type checked using a constraint context extended by the arbitrary constraint $\Xi$, and $t$ is type checked using a type context extended by $x$. The resulting run-time term makes all type and witness abstraction explicit.

This rule has two subtleties. The first is that we reject programs with *redundant* let bindings, *i.e.*, let-bound terms which are unused. This is so that we may avoid having to check for the satisfiability of the constraints of let-bound terms. Since we know $x \in fv(t)$, we can be sure that $\Phi$ entails some instance of $\Xi$. Hence the satisfiability of $\Phi$ guarantees the satisfiability of $\Xi$. Dually, if $\Xi$ is unsatisfiable, then the entire program will be untypable within the empty constraint context. We refer the reader to the work of Jones [14] for a more thorough explanation of this subtlety.

The second subtelty stems from `local` and `global` classes. We must take care to type check $u$ without using any of the local constraints from $\Phi$. This in effect forces any local constraints required by $u$ to appear within the generalized constraint $\Xi$. Dually, we must check that any global constraints required by $u$ are inherited from $\Phi$, and do not appear within $\Xi$. These tests make use of the functions *inhs* and *gens*, defined in Figure 8, which yield just the inheritable (non local) and generalizable (non global) constraints of their argument.

Finally, rule INST type checks an instance declaration. This rule shares much of the structure of rule LET, but is refined in four ways. Firstly, an instance declaration which is not at the top level of the program (*i.e.*, $\Gamma$ is non-empty) must be for a local class. Furthermore, if the instance depends on any local classes, then the instance itself must be for a local class, otherwise the simplifer could change the dynamic binding of witnesses within a let-bound term. Secondly, $u$ cannot be of any type, but must type check at the instance $[\overline{c \mapsto v}]\ v'$ of $v'$ if it is to satisfy $C\ \overline{v}$. Thirdly, witnesses for the superclass constraints $[\overline{c \mapsto v}]\ \overline{\xi'}$ of $C$ must be available in order to construct the witness for $C\ \overline{v}$.

The fourth subtlety of rule INST is to prevent overlapping instances for non-closed and non-overlapping classes, and to allow the shadowing of instance declarations for local classes. In order to type check $t$, $\Phi$ must be extended with the instance scheme $\phi$ coresponding to the instance declaration for class $C$. This is formalized by the function *extend*, defined in Figure 8. If $C$ is a local class, the *extend* function removes from $\Phi$ any instance schemes for $C$ which "match" the new scheme for $C$. If $C$ is a non-closed or non-overlapping class, *extend* is undefined if $\Phi$ contains any instance schemes for $C$ which are unifiable with the new scheme.

The notion of matching class constraints is formalized by the function *matchclasses*, also defined in Figure 8. We consider two classes to match when they are exactly equal on all their argument types, or exactly equal on the argument types in the input positions for one of the classes' functional dependencies. The test for unifiability is implemented by the function *mgus*, defined in Figure 9. Given a set of equality constraints, *mgus* returns the empty set if they are unsatisfiable, or a singleton set of a most general unifier satisfying all the constraints. *mgus* also accepts a set of type variables which should be treated as skolem constants—this shall be important in the sequel.

$$
\begin{aligned}
&mgus(\overline{a} \mid \texttt{true}) = \{\mathbf{Id}\}\\
&mgus(\overline{a} \mid b \coloneqq b, \Phi) = mgus(\overline{a} \mid \Phi)\\
&mgus(\overline{a} \mid b \coloneqq \tau, \Phi) =\\
&\qquad \{\theta \circ [b \mapsto \tau] \mid \theta \in mgus(\overline{a} \mid [b \mapsto \tau]\ \Phi)\}\\
&\qquad \text{where } b \notin fv(\tau) \text{ and } b \notin \overline{a}\\
&mgus(\overline{a} \mid \tau \coloneqq b, \Phi) = mgus(\overline{a} \mid b \coloneqq \tau, \Phi)\\
&mgus(\overline{a} \mid (\overline{\tau}) \coloneqq (\overline{v}), \Phi) = mgus(\overline{a} \mid \overline{\tau \coloneqq v} +\!\!+ \Phi)\\
&mgus(\overline{a} \mid (\tau \texttt{ -> } v) \coloneqq (\tau' \texttt{ -> } v'), \Phi) =\\
&\qquad mgus(\overline{a} \mid \tau \coloneqq \tau', v \coloneqq v', \Phi)\\
&mgus(\overline{a} \mid A \coloneqq A, \Phi) = mgus(\overline{a} \mid \Phi)\\
&mgus(\overline{a} \mid \_) = \emptyset
\end{aligned}
$$

**Figure 9:** The function *mgus*

For example, an implicit parameter may be declared as:

```
class local {} -> {a} Imp_x a where a
```

Now when typing:

```
instance Imp_x Int = 1 in
instance Imp_x Bool = True in
Imp_x
```

the second instance declaration attempts to introduce the class constraint `Imp_x Bool`, which, by the matching rule, shadows the class constraint `Imp_x Int`. Hence, the program has type `Bool`.

It should be noted that local instance declarations are very likely to surprise the programer unless they are strictly controlled using functional dependencies. The combination of `local` with `overlapping` or `closed` is also likely to lead to surprises.

## A.4 Constraint Entailment

Constraint entailment is decided by the four judgements given in Figures 10 and 11. The judgement $\Phi \vdash^p \phi \hookrightarrow W$ is true when the primitive constraint $\phi$ either appears directly within $\Phi$ (rule PID), or appears within the superclass constraints of a class constraint which may itself be projected from $\Phi$ (rule PPROJ). Hence, the witness $W$ is a possibly empty chain of witness projections applied to a witness variable.

The judgement $\Phi \vdash^c \phi \hookrightarrow W$ extends the projection judgement in two ways. Firstly, a class constraint may be satisfied by instantiating an instance scheme (rule CINST). The appropriate instance scheme is found by projection, and witnesses to the scheme's argument constraints are found recursively. The resulting witness records both the types and argument

$$\boxed{\Phi \vdash^p \phi \hookrightarrow W}$$

$$\frac{(w : \phi) \in \Phi}{\Phi \vdash^p \phi \hookrightarrow w}\ \text{PID}$$

$$\frac{(\texttt{class}\ \overline{m}\ \overline{\phi}\ \texttt{=>}\ C\ \overline{a}\ \texttt{where}\ \tau) \in \mathit{decls} \quad \Phi \vdash^p C\ \overline{v} \hookrightarrow W}{\Phi \vdash^p [\overline{a \mapsto v}]\ \phi_i \hookrightarrow \mathsf{Proj}_i^C\ W}\ \text{PPROJ}$$

$$\boxed{\Phi \vdash^c \phi \hookrightarrow W}$$

$$\frac{\Phi \vdash^p \texttt{forall}\ \overline{a}\ .\ \overline{\phi}\ \texttt{=>}\ C\ \overline{v} \hookrightarrow W \quad \Phi \vdash^c \overline{[a \mapsto v']\ \phi} \hookrightarrow W'}{\Phi \vdash^c C\ \overline{[a \mapsto v']\ v} \hookrightarrow W\ \overline{v'}\ \overline{W'}}\ \text{CINST}$$

$$\frac{\overline{a} \cap \mathit{fv}(\Phi) = \emptyset \quad \Xi = \mathit{named}(\overline{\xi}) \quad \Phi +\!\!+ \Xi \vdash^c C\ \overline{\tau} \hookrightarrow W}{\Phi \vdash^c \texttt{forall}\ \overline{a}\ .\ \overline{\xi}\ \texttt{=>}\ C\ \overline{\tau} \hookrightarrow \Lambda\overline{a}\ .\ \lambda \mathit{names}(\Xi)\ .\ W}\ \text{CHO}$$

$$\boxed{\Phi \vdash^e \phi \hookrightarrow W}$$

$$\frac{S = \{\, W \mid (\Phi \vdash^c \phi \hookrightarrow W)\} \quad \{W\} = \mathit{minimals}(\Phi \mid S)}{\Phi \vdash^e \phi \hookrightarrow W}\ \text{ELEAST}$$

**Figure 10:** $\lambda^{\mathsf{O}}$ entailment

witnesses at which the instance scheme is instantiated. We intend this rule to also apply when $\overline{a}$ and $\overline{\phi}$ are empty, in which case the rule projects the desired class constraint directly from $\Phi$.

Secondly, a higher-order instance scheme may be satisfied if an appropriate witness function can be constructed (rule CHO). This is the usual rule of function introduction in intuitionistic logic.

Notice that the $\vdash^p$ and $\vdash^c$ judgements cannot be conflated. Otherwise, given the declaration:

```
class C a => D a where a
```

and the constraint context:

$$\Phi = w : \texttt{forall . . C Int => D Int}$$

then:

$$\Phi \vdash^e \texttt{C Int} \hookrightarrow \mathsf{Proj}_1^{\mathtt{D}}\ (w\ (\mathsf{Proj}_1^{\mathtt{D}}\ (w \ldots)))$$

would be true with an infinite derivation. Similarly, it would allow class constraints to be satisfied by the indirect route of first constructing a witness for a larger constraint, then projecting out the the desired class constraint, which is just silly.

In the absence of overlapping instances, the $\vdash^c$ judgement would be the appropriate notion of entailment. However, when instance schemes may overlap this judgement is nondeterministic. To account for this, the actual entailment judgement $\Phi \vdash^e \phi \hookrightarrow W$ must first collect all possible witnesses for $\phi$ using the $\vdash^c$ judgement, and then test if $W$ is the *least* such witness.

$$\boxed{\Phi \vdash W \leq W'}$$

$$\frac{}{\Phi \vdash w \leq \mathsf{Proj}_j^D\ W}\ \text{OVARPROJ}$$

$$\frac{}{\Phi \vdash w \leq w'\ \overline{\tau'}\ \overline{W'}}\ \text{OVARINST}$$

$$\frac{}{\Phi \vdash \mathsf{Proj}_i^C\ W \leq w'\ \overline{\tau'}\ \overline{W'}}\ \text{OPROJINST}$$

$$\frac{\Phi \vdash W \leq W'}{\Phi \vdash \mathsf{Proj}_i^C\ W \leq \mathsf{Proj}_i^C\ W'}\ \text{OPROJ1}$$

$$\frac{(C, i) \leq (D, j)}{\Phi \vdash \mathsf{Proj}_i^C\ W \leq \mathsf{Proj}_j^D\ W'}\ \text{OPROJ2}$$

$$\frac{\Phi \vdash \overline{W \leq W'}}{\Phi \vdash w\ \overline{\tau}\ \overline{W} \leq w\ \overline{\tau}\ \overline{W'}}\ \text{OINST1}$$

$$\frac{w \neq w' \quad (w : \phi) \in \Phi \quad (w' : \xi) \in \Phi \quad w' : \xi \vdash^e \phi}{\Phi \vdash w\ \overline{\tau}\ \overline{W} \leq w'\ \overline{\tau'}\ \overline{W'}}\ \text{OINST2}$$

$$\mathit{minimals}(\Phi \mid S) = \left\{ W \in S \;\middle|\; \begin{array}{c} \forall W' \in S\ . \\ \Phi \vdash W' \leq W \Longrightarrow \\ \Phi \vdash W \leq W' \end{array} \right\}$$

**Figure 11:** Partial order on witnesses

The ordering on witnesses is decided by the judgement $\Phi \vdash W \leq W'$, whose rules appear in Figure 11. The ordering is particularly weak: we do not attempt to order witness functions, nor witnesses which instantiate projected witness functions. Witnesses such as these only arise in the presence of higher-order instances schemes, which are sufficiently rare that supporting them with overlapping instances seems of dubious benefit.

By inspection of rule ELEAST, and by an easy induction on the rules of Figure 11, only witnesses to the same constraint shall ever be compared. Furthermore, by inspection of rules CINST and CHO, all witnesses to instance schemes must be in $\eta$-normal form. Thus, only class constraint witnesses appear within the conclusions of the rules of Figure 11, and witness functions are incomparable.

There are three forms of witness to a class constraint: a witness variable if the class constraint appears directly within $\Phi$, a projection from another class constraint, or an instantiation of a witness function. Rules OVARPROJ, OVARINST and OPROJINST place projection witnesses before instance witnesses, and place shorted chains of projections before longer ones.

Roughly speaking, projection witnesses and instance witnesses are ordered lexicographically amongst themselves. Rules OPROJ1 and OINST1 deepen the comparison when the topmost witnesses are identical. Rule OPROJ2 imposes an arbitrary (and, for this paper, unspecified) ordering on projection witnesses from different class constraints. This is sound since if two classes share a superclass, their witnesses must similarly share a superclass witness, hence either one may be chosen without changing the semantics of the program. (Showing this is indeed the case, particularly in the presence of local classes, is quite subtle and should be formalized in the full paper.)

Rule OINST2 compares two witnesses which were constructed by the application of two distinct witness functions, bound to $w$ and $w'$. In this case, we attempt to order the instance schemes, $\phi$ and $\xi$, which $w$ and $w'$ themselves witness, by placing the more specific/constrained scheme first. This is accomplished by asking if $\xi$ entails $\phi$.

An example helps motivate this definition. Assume the type definitions:

```
class C a where a
class D a => C a where a
class E a where a
```

and that:

$\Phi = w : \texttt{forall} \cdot \texttt{.}\ \texttt{D Int => E Int}, w' : \texttt{forall a . C a => E a}$

Then there is a derivation of:

$w' : \texttt{forall a . C a => E a} \vdash^c$
$\texttt{forall} \cdot \texttt{.}\ \texttt{D Int => E Int} \hookrightarrow \lambda w'' \,.\, w'\ \texttt{Int}\ (\mathsf{Proj}_1^D\ w'')$

(using rule CHO, then rule CINST twice)

Furthermore, since this is the only such derivation, we have

$w' : \texttt{forall a . C a => E a} \vdash^e$
$\texttt{forall} \cdot \texttt{.}\ \texttt{D Int => E Int} \hookrightarrow \lambda w'' \,.\, w'\ \texttt{Int}\ (\mathsf{Proj}_1^D\ w'')$

Notice, however, that the converse entailment does not hold. Thus, $w$ is strictly less that $w'$ in the witness ordering.

We shall consider the finiteness of entailment derivations in Section A.8.

## A.5 Type Inference

In this section we turn our attention to type inference. As usual, inference proceeds by accumulating a constraint context which is simplified wherever possible. Hence, just as type checking builds upon constraint entailment, type inference shall build upon a notion of constraint simplification.

Like many languages supporting type inference, Haskell allows let-bound terms to be annotated with a user-supplied type signature. It is the responsibility of type inference to check that such a type signature is indeed an instance of the term's principal (*i.e.*, inferred) type. This test is typically left unspecified within formal presentations of type inference systems. However, we feel the complexity of $\lambda^{\mathsf{O}}$ constraint entailment warrants a formal treatment of user-supplied type signatures. Indeed, we found that doing so exposed substantial additional complexity both in the type inference system and the constraint simplifier.

$$\boxed{\overline{a} \mid \Phi_0 \mid \theta \mid \Phi \mid \Gamma \vdash t : \tau \hookrightarrow T}$$

$$\frac{(\texttt{class}\ \overline{m}\ \overline{\xi}\ \texttt{=>}\ C\ \overline{b}\ \texttt{where}\ \tau) \in \mathit{decls} \quad \overline{c}\ \text{fresh}}{\overline{a} \mid \Phi_0 \mid \mathbf{Id} \mid w : C\ \overline{c} \mid \Gamma \vdash C : [\overline{b \mapsto c}]\ \tau \hookrightarrow \mathsf{Proj}_{\mathsf{V}}^C\ w}\ \text{ICLASS}$$

$$\frac{(x : \texttt{forall}\ \overline{b}\ \texttt{.}\ \overline{\xi}\ \texttt{=>}\ \tau) \in \Gamma \quad \overline{c}\ \text{fresh} \quad \Xi = \mathit{named}([\overline{b \mapsto c}]\ \overline{\xi})}{\overline{a} \mid \Phi_0 \mid \mathbf{Id} \mid \Xi \mid \Gamma \vdash x : [\overline{b \mapsto c}]\ \tau \hookrightarrow x\ \overline{c}\ \mathit{names}(\Xi)}\ \text{IVAR}$$

$$\frac{b\ \text{fresh} \quad \overline{a} \mid \Phi_0 \mid \theta \mid \Phi_1 \mid \Gamma, x : b \vdash t : \tau \hookrightarrow T}{\overline{a} \mid \Phi_0 \mid \theta_{\setminus\{a\}} \mid \Phi_1 \mid \Gamma \vdash \texttt{\textbackslash}x\ \texttt{->}\ t : (\theta\ b\ \texttt{->}\ \tau) \hookrightarrow \lambda x\ .\ T}\ \text{IABS}$$

$$\frac{\begin{array}{c}\overline{a} \mid \Phi_0 \mid \theta_1 \mid \Phi_1 \mid \Gamma \vdash t : \upsilon \hookrightarrow T \\ \overline{a} \mid \Phi_0 \mid \theta_2 \mid \Phi_2 \mid \theta_1\ \Gamma \vdash u : \upsilon' \hookrightarrow U \\ b\ \text{fresh} \\ \Phi_3 = \Phi_1 +\!\!+ \Phi_2 +\!\!+ (\upsilon\ \texttt{:=:}\ (\upsilon'\ \texttt{->}\ b))\end{array}}{\overline{a} \mid \Phi_0 \mid \theta_2 \circ \theta_1 \mid \Phi_3 \mid \Gamma \vdash t\ u : b \hookrightarrow T\ U}\ \text{IAPP}$$

$$\frac{\begin{array}{c}x \in \mathit{fv}(t) \\ \overline{a} \mid \mathit{inhs}(\Phi_0) \mid \theta_1 \mid \Phi_1 \mid \Gamma \vdash u : \upsilon \hookrightarrow U \\ (\Phi_2 \mid \overline{b} \mid \Phi_3) = \mathit{gen}(\theta_1\ \Gamma \mid \Phi_1 \mid \upsilon) \\ \mathit{gens}(\Phi_3) = \Phi_3 \\ \sigma = \texttt{forall}\ \overline{b}\ \texttt{.}\ \mathit{anon}(\Phi_3)\ \texttt{=>}\ \upsilon \\ \overline{a} \mid \Phi_0 \mid \theta_2 \mid \Phi_4 \mid (\theta_1\ \Gamma), x : \sigma \vdash t : \tau \hookrightarrow T \\ \Phi_5 = (\theta_2\ \Phi_2) +\!\!+ \Phi_4\end{array}}{\begin{array}{l}\overline{a} \mid \Phi_0 \mid (\theta_2 \circ \theta_1)_{\restriction \mathit{fv}(\Gamma)} \mid \Phi_5 \mid \Gamma \vdash \\ \quad \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \tau \hookrightarrow \\ \quad\quad \mathsf{let}\ x = \Lambda\overline{b}\ .\ \lambda \mathit{names}(\Phi_4)\ .\ U\ \mathsf{in}\ T\end{array}}\ \text{ILET}$$

$$\frac{\begin{array}{c}\overline{a} \mid \Phi_0 \mid \theta_1 \mid \Phi_1 \mid \Gamma \vdash t : \tau \hookrightarrow T \\ \langle \Phi_0 \mid \overline{a} \cup \mathit{fv}(\theta_1\ \Gamma) \cup \mathit{fv}(\tau) \mid \Phi_1 \rangle \rhd^c \langle \theta_2 \mid \Phi_2 \mid B \rangle\end{array}}{\begin{array}{c}\overline{a} \mid \Phi_0 \mid (\theta_2 \circ \theta_1)_{\restriction \mathit{fv}(\Gamma)} \mid \Phi_2 \mid \Gamma \vdash \\ t : \theta_2\ \tau \hookrightarrow \mathsf{letw}\ B\ \mathsf{in}\ T\end{array}}\ \text{ISIMP}$$

**Figure 12:** $\lambda^{\mathsf{O}}$ type Inference (part 1 of 2)

Figures 12 and 13 present rules for deciding the type inference judgement $\overline{a} \mid \Phi_0 \mid \theta \mid \Phi \mid \Gamma \vdash t : \tau \hookrightarrow T$. Here $\overline{a}$ is a set of type variables, and $\Phi_0$ a constraint context, introduced by the user-supplied type signatures in the scope $t$. $\Phi$ is the inferred constraint context, and $\theta$ the inferred substitution to apply to type variables in $\Gamma$. We intend $\Gamma$, $\tau$ and $T$ to be as in the type checking judgement.

For the most part, the rules for type inference mirror those for type checking given in Figure 7.

Rule ILET uses the generalization function *gen*, defined in Figure 14. This function is careful to generalize over non-inheritable constraints, *i.e.*, class constraints for local classes, regardless of their free type variables. The reader will notice that, unlike the work of Jones [15], $\lambda^{\mathsf{O}}$ makes no attempt to exploit functional dependencies during generalization. This is because $\lambda^{\mathsf{O}}$ does *not* follow Haskell's rule for rejecting type schemes where any generalized type variables

$$\frac{\begin{array}{c}\sigma = \texttt{forall}\ \overline{b}\ .\ \overline{\xi}\ \texttt{=>}\ v \\ \overline{a} \cap \overline{b} = \emptyset \quad \Xi = named(\overline{\xi}) \quad x \in fv(t) \\ \overline{a} +\!\!+ \overline{b} \vdash \Xi\ \mathsf{constraint} \quad \overline{a} +\!\!+ \overline{b} \vdash v\ \mathsf{type} \\ gens(\Xi) = \Xi \\ \overline{a} +\!\!+ \overline{b} \mid inhs(\Phi_0) +\!\!+ \Xi \mid \theta_1 \mid \Phi_1 \mid \Gamma \vdash u : v' \hookrightarrow U \\ S = \overline{a} \cup \overline{b} \cup fv(\theta_1\ \Gamma) \cup fv(v') \\ \langle inhs(\Phi_0) +\!\!+ \Xi \mid S \mid v' :\!\!= v, \Phi_1 \rangle \triangleright^c \langle \theta_2 \mid \Phi_2 \mid B \rangle \\ inhs(\Phi_2) = \Phi_2 \\ \exists \overline{c}\ .\quad \overline{\theta_2 \circ \theta_1\ a = c} \wedge \overline{c}\ \mathrm{distinct} \\ \wedge\ \overline{c} \cap (\overline{a} \cup fv(\theta_2 \circ \theta_1\ \Gamma) \cup fv(\Phi_2)) = \emptyset \\ \overline{a} \mid \Phi_0 \mid \theta_3 \mid \Phi_3 \mid \theta_2 \circ \theta_1\ \Gamma, x : \sigma \vdash t : \tau \hookrightarrow T \end{array}}{\begin{array}{l}\overline{a} \mid \Phi_0 \mid (\theta_3 \circ \theta_2 \circ \theta_1)_{\restriction fv(\Gamma)} \mid (\theta_3\ \Phi_2) +\!\!+ \Phi_3 \mid \Gamma \vdash \\ \quad \texttt{let}\ x\ \texttt{::}\ \sigma\ \texttt{=}\ u\ \texttt{in}\ t : \tau \hookrightarrow \\ \qquad \mathsf{let}\ x = (\Lambda \overline{b}\ .\ \lambda names(\Xi)\ .\ \mathsf{letw}\ B\ \mathsf{in}\ U)\ \mathsf{in}\ T\end{array}}\ \textsc{alet}$$

$$\frac{\begin{array}{c}(\texttt{class}\ \overline{m}\ \overline{\phi'}\ \texttt{=>}\ C\ \overline{c}\ \texttt{where}\ v') \in decls \\ \phi = \texttt{forall}\ \overline{b}\ .\ \overline{\xi}\ \texttt{=>}\ C\ \overline{v} \quad \Xi = named(\overline{\xi}) \\ \overline{a} \cap \overline{b} \cap \overline{c} = \emptyset \\ (\Gamma \neq \cdot) \vee (inhs(\Xi) \neq \Xi) \Longrightarrow \texttt{local} \in \overline{m} \\ \overline{a} +\!\!+ \overline{b} \vdash \Xi\ \mathsf{constraint} \quad \overline{a} +\!\!+ \overline{b} \vdash \overline{v\ \mathsf{type}} \\ w, \overline{w'}\ \mathrm{fresh} \quad gens(\Xi) = \Xi \\ \overline{a} +\!\!+ \overline{b} \mid inhs(\Phi_0) +\!\!+ \Xi \mid \theta_1 \mid \Phi_1 \mid \Gamma \vdash u : \tau' \hookrightarrow U \\ S = \overline{a} \cup \overline{b} \cup fv(\theta_1\ \Gamma) \cup fv(\tau') \\ \Xi' = (\tau' :\!\!= \overline{[c \mapsto v]}\ v') +\!\!+ \overline{w' : [\overline{c \mapsto v}]\ \phi'} \\ \langle inhs(\Phi_0) +\!\!+ \Xi \mid S \mid \Xi' +\!\!+ \Phi_1 \rangle \triangleright^c \langle \theta_2 \mid \Phi_2 \mid B \rangle \\ inhs(\Phi_2) = \Phi_2 \\ \exists \overline{c'}\ .\quad \overline{\theta_2 \circ \theta_1\ a = c'} \wedge \overline{c'}\ \mathrm{distinct} \\ \wedge\ \overline{c'} \cap (\overline{a} \cup fv(\theta_2 \circ \theta_1\ \Gamma) \cup fv(\Phi_2)) = \emptyset \\ \Phi_4 = extend(\Phi_0 \mid w : \phi)\ \text{well-defined} \\ \overline{a} \mid \Phi_4 \mid \theta_3 \mid \Phi_3 \mid \theta_2 \circ \theta_1\ \Gamma \vdash t : \tau \hookrightarrow T \end{array}}{\begin{array}{l}\overline{a} \mid \Phi_0 \mid (\theta_3 \circ \theta_2 \circ \theta_1)_{\restriction fv(\Gamma)} \mid (\theta_3\ \Phi_2) +\!\!+ \Phi_3 \mid \Gamma \vdash \\ \quad \texttt{instance}\ \overline{b}\ .\ \overline{\xi}\ \texttt{=>}\ C\ \overline{v}\ \texttt{=}\ u\ \texttt{in}\ t : \tau \hookrightarrow \\ \qquad \mathsf{letw}\ w = (\Lambda \overline{b}\ .\ \lambda names(\Xi)\ . \\ \qquad\qquad \mathsf{letw}\ B\ \mathsf{in}\ C\ \overline{w'}\ U)\ \mathsf{in}\ T\end{array}}\ \textsc{ainst}$$

**Figure 13:** $\lambda^{\mathsf{O}}$ type Inference (part 2 of 2)

$$\begin{array}{l} gen(\Gamma \mid \Phi \mid \tau) = (\Xi_1 \mid \overline{a} \mid \Xi_2) \\ \text{where} \quad \overline{a} = (fv(\tau) \cup fv(\Phi)) \setminus fv(\Gamma) \\ \qquad \Xi_1 = \left\{ w : \xi \;\middle|\; \begin{array}{l}(w : \xi) \in \Phi, \\ fv(\xi) \cap \overline{a} = \emptyset, \\ inheritable(\xi)\end{array} \right\} \\ \qquad \Xi_2 = \left\{ w : \xi \;\middle|\; \begin{array}{l}(w : \xi) \in \Phi, \\ fv(\xi) \cap \overline{a} \neq \emptyset \vee \neg inheritable(\xi)\end{array} \right\} \end{array}$$

**Figure 14:** The generalization function *gen*

appear only within the type scheme's constraint. This syntactic test for ambiguity is, strictly speaking, unnecessary, since any program containing a truly ambiguous type scheme shall be untypable in the constraint context `true`, and hence shall be rejected at the top-level.

Rule IAPP infers the type of an application by introducing an *equality constraint* into the current constraint context. In effect, this pushes the task of finding a most-general unifier of two types onto the simplifier. This is marginally neater than calling a most-general unifier function directly. However, our real motivation for structuring type inference is this way is to ease the introduction of subtype constraints in Section A.9.

Rule ISIMP allows the current constraint context to be simplified at any point during inference. Some simplifications are possible when it is known that a particular type variable appears within the constraint only, and does not "escape" into types or the type context. To support such simplifications, the ISIMP rule passes the set of all escaped type variables to the simplifier.

The two rules ALET and AINST deal with annotated let-expressions and instance declarations respectively. Much of their complexity is for checking that the inferred type matches the annotation. There are two aspects to this test. Firstly, we must make sure that all the constraints arising from type inference for the term are either entailed by the supplied constraint, or may be inherited from the surrounding context. This is checked by calling the simplifier on the inferred constraint context, and checking all residual constraints do not contain any generalized type variables. Secondly, we must make sure all generalized type variables remain free (though possibly renamed) after inference. A generalized type variable which becomes bound during simplification signals that the annotated type scheme is more polymorphic than the inferred type.

Notice that the simplifier must try "as hard as it can" when considering the constraints given to it by rules ALET and AINST, otherwise the program could be rejected. This is in contrast to the invocation of the simplifer in rule ISIMP, in which simplification is "optional." We have not attempted to formalize the distinction between these two simplifer modes.

## A.6 Constraint Simplification

This brings us to the heart of type inference: constraint simplification. The simplifier attempts to rewrite a constraint context to a smaller constraint context which entails the original. At the same time, it attempts to reduce equality constraints down to a simple substitution. The simplifier may rewrite a constraint to `false` should it discover the constraint is unsatisfiable, thus signaling a type error.

The simplifier consists of three layers. At the bottom-most layer is the judgement $\langle \overline{w} \mid \overline{a} \mid \Phi \rangle \triangleright \langle \overline{w'} \mid \theta \mid \Phi' \mid B \rangle$, whose rules are given in Figures 15 and 16. This judgement is true when the constraint context $\Phi$ may be simplified in a single step to the constraint $\Phi'$. The set $\overline{w}$ contains the witness names of constraints in $\Phi$ which derive from user-supplied type annotations. The simplifier uses this set to bias some steps. The set $\overline{a}$ contains all type variables in $\Phi$ which may be subject to further constraints as type inference proceeds. That is, a type variable in $\Phi$ which is *not* in $\overline{a}$ must appear only in $\Phi$. This shall be used to enable some more aggressive simplification steps. The output set $\overline{w'}$ contains the witness names for constraints in $\Phi'$ which are a direct consequence of user-supplied constraints in $\Phi$. The substitution $\theta$ contains any substitution arising from simplifying an equality constraint, and $B$ contains bindings for witness variables in $\Phi$ which were simplified away.

$$\boxed{\langle \overline{w} \mid \overline{a} \mid \Phi \rangle \triangleright \langle \overline{w'} \mid \theta \mid \Phi' \mid B \rangle}$$

**Equality Constraints**

$$\frac{}{\langle \overline{w} \mid \overline{a} \mid w : b \coloneqq: b, \Phi \rangle \triangleright \langle \emptyset \mid \mathbf{Id} \mid \Phi \mid \cdot \rangle} \text{ s1}$$

$$\frac{b \notin fv(\tau)}{\langle \overline{w} \mid \overline{a} \mid w : b \coloneqq: \tau, \Phi \rangle \triangleright \langle \emptyset \mid [b \mapsto \tau] \mid [b \mapsto \tau]\ \Phi \mid \cdot \rangle} \text{ s2}$$

$$\frac{}{\langle \overline{w} \mid \overline{a} \mid w : \tau \coloneqq: b, \Phi \rangle \triangleright \langle \emptyset \mid \mathbf{Id} \mid w : b \coloneqq: \tau, \Phi \mid \cdot \rangle} \text{ s3}$$

$$\frac{\overline{w'} \text{ fresh} \quad S = \mathbf{if}\ w \in \overline{w}\ \mathbf{then}\ \overline{w'}\ \mathbf{else}\ \emptyset}{\begin{array}{c}\langle \overline{w} \mid \overline{a} \mid w : (\overline{\tau}) \coloneqq: (\overline{\upsilon}), \Phi \rangle \triangleright \\ \langle S \mid \mathbf{Id} \mid \overline{w' : \tau \coloneqq: \upsilon} +\!\!+ \Phi \mid \cdot \rangle\end{array}} \text{ s4}$$

$$\frac{\overline{w'} \text{ fresh} \quad S = \mathbf{if}\ w \in \overline{w}\ \mathbf{then}\ \overline{w'}\ \mathbf{else}\ \emptyset}{\begin{array}{c}\langle \overline{w} \mid \overline{a} \mid w : (\tau_1 \texttt{ -> } \tau_2) \coloneqq: (\upsilon_1 \texttt{ -> } \upsilon_2), \Phi \rangle \triangleright \\ \langle S \mid \mathbf{Id} \mid \overline{w' : \tau \coloneqq: \upsilon} +\!\!+ \Phi \mid \cdot \rangle\end{array}} \text{ s5}$$

$$\frac{}{\langle \overline{w} \mid \overline{a} \mid w : A \coloneqq: A, \Phi \rangle \triangleright \langle \emptyset \mid \mathbf{Id} \mid \Phi \mid \cdot \rangle} \text{ s6}$$

$$\frac{}{\langle \overline{w} \mid \overline{a} \mid w : \_ \coloneqq: \_, \Phi \rangle \triangleright \langle \emptyset \mid \mathbf{Id} \mid \texttt{false} \mid \cdot \rangle} \text{ s7}$$

**Figure 15:** Simplifier (part 1 of 3)

Rules s1–s7 simplify equality constraints in the obvious way. The somewhat ugly calculation of $S$ in rules s4 and s5 is actually unnecessary, since constraints appearing within user-supplied type annotations would never contain equality constraints. However, including this calculation shall ease the path to subtype constraints in Section A.9.

The remaining rules tackle class constraints. These make use of the ancillary definitions given in Figure 18, which we shall explain as required.

Rule s8 attempts to collapse identical class constraints, or satisfy a class constraint by projection from another class constraint. The function *closesups* calculates the closure under superclass projection of all class constraints of its argument. For example, given the declaration:

```
class D a => C a where a
```

then the constraint context:

$$w : \texttt{C Int}, w' : \texttt{D Int}$$

may be simplified to $w$ : `C Int` with the binding $w' = \mathsf{Proj}_1^C\ w$.

This rule, in common with most of the rules for class constraints, uses the set $\overline{w}$ to avoid simplifying away a class constraint originating from a user-supplied type annotation. However, it is perfectly valid for inferred class constraints to be satisfied by a class constraint or instance scheme originating from such an annotation.

**Class Constraints**

$$\frac{w \notin \overline{w} \quad (W : C\ \overline{\tau}) \in closesups(\Phi)}{\langle \overline{w} \mid \overline{a} \mid w : C\ \overline{\tau}, \Phi \rangle \triangleright \langle \emptyset \mid \mathbf{Id} \mid \Phi \mid w = W \rangle} \text{ s8}$$

$$\frac{\begin{array}{c}(\texttt{class}\ \overline{m}\ \overline{\xi}\ \texttt{=>}\ C\ \overline{b}\ \texttt{where}\ \upsilon) \in decls \\ \texttt{closed} \in \overline{m} \quad w \notin \overline{w} \\ improve(\overline{w} \mid \overline{a} \mid \Phi \mid C\ \overline{\tau}) = \texttt{false}\end{array}}{\langle \overline{w} \mid \overline{a} \mid w : C\ \overline{\tau}, \Phi \rangle \triangleright \langle \emptyset \mid \mathbf{Id} \mid \texttt{false} \mid \cdot \rangle} \text{ s9}$$

$$\frac{\begin{array}{c}(\texttt{class}\ \overline{m}\ \overline{\xi}\ \texttt{=>}\ C\ \overline{b}\ \texttt{where}\ \upsilon) \in decls \\ \texttt{closed} \in \overline{m} \quad w \notin \overline{w} \\ \Xi = improve(\overline{w} \mid \overline{a} \mid \Phi \mid C\ \overline{\tau}) \neq \texttt{false}\end{array}}{\langle \overline{w} \mid \overline{a} \mid w : C\ \overline{\tau}, \Phi \rangle \triangleright \langle \emptyset \mid \mathbf{Id} \mid \Xi +\!\!+ (w : C\ \overline{\tau}, \Phi) \mid \cdot \rangle} \text{ s10}$$

$$\frac{\begin{array}{c}(\texttt{class}\ \overline{m}\ \overline{\xi'}\ \texttt{=>}\ C\ \overline{a'}\ \texttt{where}\ \tau') \in decls \\ (\texttt{closed} \in \overline{m}) \vee (\texttt{overlap} \notin \overline{m}) \vee (fv(\overline{\tau}) = \emptyset) \quad w \notin \overline{w} \\ \{w'\} = mininst(\overline{w} \mid \overline{a} \mid \Phi \mid C\ \overline{\tau}) \\ (w' : \texttt{forall}\ \overline{b}\ .\ \overline{\xi}\ \texttt{=>}\ C\ \overline{\upsilon}) \in \Phi \quad \overline{b'} \text{ fresh} \\ \{\theta\} = mgus(fv(\overline{\tau}) \mid \overline{\tau \coloneqq: [b \mapsto b']\ \upsilon}) \\ \Xi = named(\theta \circ [\overline{b \mapsto b'}]\ \overline{\xi})\end{array}}{\begin{array}{c}\langle \overline{w} \mid \overline{a} \mid w : C\ \overline{\tau}, \Phi \rangle \triangleright \\ \langle \emptyset \mid \mathbf{Id} \mid \Xi +\!\!+ \Phi \mid w = w'\ \overline{\theta\ b'}\ names(\Xi) \rangle\end{array}} \text{ s11}$$

$$\frac{\begin{array}{c}w \notin \overline{w} \quad fv(\overline{\tau}) = \emptyset \\ \emptyset = candidates(\overline{w} \mid \overline{a} \mid \Phi \mid C\ \overline{\tau}) \\ (W : C\ \overline{\tau}) \notin closesups(\Phi)\end{array}}{\langle \overline{w} \mid \overline{a} \mid w : C\ \overline{\tau}, \Phi \rangle \triangleright \langle \emptyset \mid \mathbf{Id} \mid \texttt{false} \mid \cdot \rangle} \text{ s12}$$

$$\frac{\begin{array}{c}w \notin \overline{w} \quad (W : C\ \overline{\upsilon}) \in closesups(\Phi) \\ (\texttt{class}\ \overline{m}\ \overline{\xi}\ \texttt{=>}\ C\ \overline{a'}\ \texttt{where}\ \tau') \in decls \\ (\{\overline{b}\}\ \texttt{->}\ \{\overline{b'}\}) \in \overline{m} \quad \overline{c} \text{ fresh} \\ \theta = [\overline{c \mapsto \tau}] \circ [\overline{a' \mapsto c}] \quad \theta' = [\overline{c \mapsto \upsilon}] \circ [\overline{a' \mapsto c}] \\ \forall b'' \in \overline{b}\ .\ \theta\ b'' = \theta'\ b'' \\ \Xi = named(\{\theta\ b'' \coloneqq: \theta'\ b'' \mid b'' \in \overline{b'}\})\end{array}}{\langle \overline{w} \mid \overline{a} \mid w : C\ \overline{\tau}, \Phi \rangle \triangleright \langle \emptyset \mid \mathbf{Id} \mid \Xi +\!\!+ \Phi \mid w = W \rangle} \text{ s13}$$

$$\frac{\begin{array}{c}w \notin \overline{w} \quad \overline{c} \text{ fresh} \quad w' \text{ fresh} \\ \Xi = named([\overline{b \mapsto c}]\ \overline{\xi}) \\ \Phi' = \{w'' : \phi \mid (w'' : \phi) \in \Phi, w'' \in \overline{w}\} \\ \langle \overline{w} +\!\!+ names(\Xi) \mid \overline{a} \mid w' : C\ \overline{[b \mapsto c]\ \upsilon}, (\Xi +\!\!+ \Phi') \rangle \triangleright^* \\ \quad \langle \theta \mid \Xi +\!\!+ \Phi' \mid B \rangle\end{array}}{\begin{array}{c}\langle \overline{w} \mid \overline{a} \mid w : \texttt{forall}\ \overline{b}\ .\ \overline{\xi}\ \texttt{=>}\ C\ \overline{\upsilon}, \Phi \rangle \triangleright \\ \langle \emptyset \mid \theta \mid \Phi \mid w = \Lambda \overline{c}\ .\ \lambda names(\Xi)\ .\ \mathsf{letw}\ B\ \mathsf{in}\ w' \rangle\end{array}} \text{ s14}$$

**Figure 16:** Simplifier (part 2 of 3)

Rules s9 and s10 both deal with *improvement* of a target class constraint for a closed class. For example, given the declaration:

```
class closed C a where a
```

and the constraint context:

$$\Phi = \begin{array}{l} w_1 : \texttt{C (Int, Bool)}, w_2 : \texttt{C (Int, Char)}, \\ w_3 : \texttt{C Int}, w_4 : \texttt{C (a, b)} \end{array}$$

$$\boxed{\langle \overline{w} \mid \overline{a} \mid \Phi \rangle \triangleright^* \langle \overline{w'} \mid \theta \mid \Phi' \mid B \rangle}$$

$$\frac{}{\langle \overline{w} \mid \overline{a} \mid \Phi \rangle \triangleright^* \langle \emptyset \mid \mathbf{Id} \mid \Phi \mid \cdot \rangle}\ \textsc{sdone}$$

$$\frac{\langle \overline{w} \mid \overline{a} \mid \Phi \rangle \triangleright \langle \overline{w'} \mid \theta \mid \Phi' \mid B \rangle \quad \langle \overline{w} \cup \overline{w'} \mid \overline{a} \cup \bigcup_{a \in \overline{a}} fv(\theta\ a) \mid \Phi' \rangle \triangleright^* \langle \overline{w''} \mid \theta' \mid \Phi'' \mid B' \rangle}{\langle \overline{w} \mid \overline{a} \mid \Phi \rangle \triangleright^* \langle \overline{w'} \cup \overline{w''} \mid \theta' \circ \theta \mid \Phi'' \mid B' +\!\!+ B \rangle}\ \textsc{sstep}$$

$$\boxed{\langle \Phi_0 \mid \overline{a} \mid \Phi_1 \rangle \triangleright^c \langle \theta \mid \Phi_2 \mid B \rangle}$$

$$\frac{\begin{array}{c}\overline{w} = names(\Phi_0) \\ \langle \overline{w} \mid \overline{a} \mid \Phi_0 +\!\!+ \Phi_1 \rangle \triangleright^* \langle \overline{w'} \mid \theta \mid \Phi_3 \mid B \rangle \\ \Phi_4 = \{ w : \xi \mid (w : \xi) \in \Phi_3, w \notin \overline{w} \cup \overline{w'} \}\end{array}}{\langle \Phi_0 \mid \overline{a} \mid \Phi_1 \rangle \triangleright^c \langle \theta \mid \Phi_4 \mid B \rangle}\ \textsc{simp}$$

**Figure 17:** Simplifier (part 3 of 3)

where $w_1$, $w_2$ and $w_3$ are in $\overline{w}$, then improvement allows the class constraint `C (a, b)` to be improved to `C (Int, b)`. This involves three steps.

Firstly, all of the *candidate instance schemes* which could possibly satisfy `C (a, b)` are found, and their instantiations collected using the function *candidates*, defined in Figure 18. This function determines if a particular instance scheme may match the target class constraint by recursively invoking the simplifier and rejecting those instance schemes whose constraints may be simplifed to `false`. Notice that since the simplifier is incomplete for unsatisfiable constraints the set of candidate instance schemes may be approximated by a larger set than necessary, but this is sound. For the example, this yields the two candidates `C (Int, Bool)` and `C (Int, Char)`. Notice also that we need only look in $\Phi$ for candidate instance schemes, and we need not consider the instances schemes available by projection. This is because no class may inherit from a closed class.

Secondly, the *least-common generalization* [16] of all the candidates is found using the function *lcg*, defined in Figure 19. For the example, this yields `C (Int, c)`, for fresh type variable `c`.

(The careful reader will notice that *lcg* does not yield a least generalization when types contain common subexpressions. For example:

$$lcg(\{\texttt{(Int, Int)}, \texttt{(Bool, Bool)}\}) = \texttt{(a, b)}$$

instead of `(a, a)`. Thus rule S9 won't always improve a constraint as much as possible. Thankfully, however, the simplifier correctness depends only on the result being *some* generalization, and not the least such.)

Finally, the least common generalization type is unified with the target class constraint.

These three steps are brought together by the function *improve* of Figure 18. This function either yields `false` if no candidates were found (and thus the target class constraint cannot be satisfied by any known instance schemes), or a set

$$\begin{array}{l} closesup(W : C\ \overline{\tau}) = \\ \qquad closesups(\{\mathsf{Proj}_i^C\ \ W : [\overline{a \mapsto \tau}]\ \phi_i \mid 1 \le i \le n\}) \\ \qquad \cup \{ W : C\ \overline{\tau} \} \\ \qquad\qquad \text{where } (\texttt{class}\ \overline{m}\ \overline{\phi}\ \texttt{=>}\ C\ \overline{a}\ \texttt{where}\ \upsilon) \in decls \\ closesup(W : \phi) = \{ W : \phi \} \end{array}$$

$$closesups(\Phi) = \bigcup \{ closesup(W : \phi) \mid (W : \phi) \in \Phi \}$$

$$candidates(\overline{w} \mid \overline{a} \mid \Phi \mid C\ \overline{\tau}) = \left\{ (w \mid \overline{\theta\ \tau}) \;\middle|\; \begin{array}{l} (w : \texttt{forall}\ \overline{a}\ \texttt{.}\ \overline{\xi}\ \texttt{=>}\ C\ \overline{v}) \in \Phi, \\ w \in \overline{w}, \\ \overline{b} \text{ fresh}, \\ \Xi = named([\overline{a \mapsto b}]\ \overline{\xi}) +\!\!+ \overline{\tau :\!=: [\overline{a \mapsto b}]\ v}, \\ \langle \overline{w} \mid \overline{a} \mid \Phi +\!\!+ \Xi \rangle \not\triangleright^* \langle \_ \mid \_ \mid \texttt{false} \mid \_ \rangle \end{array} \right\}$$

$$\begin{array}{l} improve(\overline{w} \mid \overline{a} \mid \Phi \mid C\ \overline{\tau}) = \\ \qquad \textbf{let}\ S = candidates(\overline{w} \mid \overline{a} \mid \Phi \mid C\ \overline{\tau}) \\ \qquad \textbf{in if}\ S = \emptyset\ \textbf{then}\ \texttt{false} \\ \qquad\quad \textbf{else let}\ \overline{(\upsilon')} = lcg(\{\overline{(\upsilon)} \mid (\_ \mid \overline{\upsilon}) \in S\}) \\ \qquad\qquad \textbf{in}\ \overline{\tau :\!=: \upsilon'} \end{array}$$

$$\begin{array}{l} leqinst(\Phi \mid w \mid w') \iff \\ \quad \langle \{w'\} \mid fv(\xi') \cup fv(\xi) \mid w : \xi, w' : \xi' \rangle \triangleright^* \langle \_ \mid \_ \mid w' : \xi' \mid \_ \rangle \\ \qquad\qquad \text{where } (w : \xi) \in \Phi \text{ and } (w' : \xi') \in \Phi \end{array}$$

$$mininst(\overline{w} \mid \overline{a} \mid \Phi \mid C\ \overline{\tau}) = \left\{ w \;\middle|\; \begin{array}{l} S = candidates(\overline{w} \mid \overline{a} \mid \Phi \mid C\ \overline{\tau}), \\ (w \mid \_) \in S, \\ \forall w' \in S\ .\ leqinst(\Phi \mid w' \mid w) \implies \\ \quad leqinst(\Phi \mid w \mid w') \end{array} \right\}$$

**Figure 18:** Ancillary functions for constraint simplification

of equality constraints to effect the third step given above. Rule S9 applies in the former case, and rule S10 in the latter.

Rule S11 eliminates a class constraint for a closed class when the set of candidate instance schemes has a least element. The ordering amongst instance schemes is decided by the function *leqinst* of Figure 18. The order relation induced by *leqinst* is a subset of that used by Rule OINST2 of Figure 11. Instead of invoking the entailment judgement, *leqinst* recursively invokes the simplifier to check if the smaller instance scheme may be simplified away using the larger instance scheme. (As in type inference rules ALET and AINST, we assume the simplier trys as hard as it can with these constraints, otherwise a least instance may never be found.) The function *mininst* uses *candidates* and *leqinst* to find all the minimal instance schemes applicable to the target class constraint. Rule S11 then only applies when *mininst* yields a singleton set.

One subtlety with rule S11 remains. To illustrate, consider the type declaration:

```
class closed C a where a
```

and the constraint context:

$$\Phi = w_1 : \texttt{forall a . C a}, w_2 : \texttt{C Int}, w_3 : \texttt{C b}$$

where $w_1$ and $w_2$ are in $\overline{w}$. Notice that $w_2$ : `C Int` is the

$$
\begin{array}{l}
lcg(\{\tau\}) = \tau \\
lcg(\{A, A\} \cup S) = lcg(\{A\} \cup S) \\
lcg(\{a, a\} \cup S) = lcg(\{a\} \cup S) \\
lcg(\{\texttt{(}\overline{\tau}\texttt{)}, \texttt{(}\overline{\upsilon}\texttt{)}\} \cup S) = lcg(\{\overline{(lcg(\{\tau, \upsilon\}))}\} \cup S) \\
lcg(\{(\tau \texttt{ -> } \upsilon), (\tau' \texttt{ -> } \upsilon')\} \cup S) = \\
\qquad\qquad lcg(\{lcg(\{\tau, \tau'\}) \texttt{ -> } lcg(\{\upsilon, \upsilon'\})\} \cup S) \\
lcg(\_) = a \qquad\qquad \text{where } a \text{ fresh}
\end{array}
$$

**Figure 19:** The least-common-generalisation function *lcg*

unique minimal instance scheme satisfying `C b`, suggesting `b` should be bound to `Int`, and $w_3$ bound to $w_2$. However, `b` may later be constrained to some type other than `Int`, in which case the instance scheme `forall a . C a` would be the only way to satisfy `C b`.

To prevent such a premature commitment to a least instance scheme, rule S11 is only applicable when the target class constraint is subsumed by the least matching instance scheme. This test is implemented using the function *mgus*, already defined in Figure 9. Rule S11 invokes *mgus* with all of the free type variables of the target class constraint as skolemised variables, which effectively prevents their substitution. In the above example, the rule would find

$$mgus(\{\texttt{b}\} \mid \texttt{b} :=: \texttt{Int}) = \emptyset$$

and thus the unsound simplification would not occur.

Conveniently, rule S11 subsumes the traditional Haskell rule for simplifying class constraints for non-closed and non-overlapping classes. In this case, the invocation of *candidates* within *mininst* yields either the empty set or a singleton set, which is trivially ordered. Furthermore, rule S11 is also appropriate for monomorphic class constraints of overlapping classes.

Rules S12 signals failure if there is no way of eliminating a monomorphic class constraint, either by projection from another constraint, or by using an instance scheme. Though this rule mimics Haskell's treatment of monomorphic class constraints, it is not entirely consistent with the view that non-closed classes may have their instance schemes scattered arbitrarily across modules. Nevertheless, this rule is vital to the success of our encoding of subtyping constraints within $\lambda^{\mathsf{O}}$.

Rule S13 exploits any functional dependency annotations to merge two class constraints which agree on their functional argument types. The rule simply unifies their functional result types.

Finally, rule S14 simplifies a higher-order constraint `forall` $\overline{b}$ `.` $\overline{\xi}$ `=>` $C\ \overline{\upsilon}$ by extending the current constraint context by $\overline{\xi}$ and $C\ \overline{\upsilon}$ (suitably named), and attempting to (recursively) simplify away $C\ \overline{\upsilon}$. If this succeeds, the appropriate witness function may be constructed by abstracting over the witness names assigned to $\overline{\xi}$. Care must be taken to ensure that $C\ \overline{\upsilon}$ is the only "simplifiable" constraint, *i.e.*, that all other constraints in the recursive invocation of the simplifier are marked as user-supplied constraints.

Of course, in a practical implementation of the simplifier, it is impossible to predict whether the recursive invocation of the simplifier needed by rule S14 would succeed. Hence the implementation must support backtracking. Since this is potentially costly, rule S14 should only be applied when absolutely necessary, *i.e.*, when the type inference rules ALET and AINST would otherwise fail.

Two more judgements are necessary to complete the machinery for constraint simplification. The judgement $\langle \overline{w} \mid \overline{a} \mid \Phi \rangle \triangleright^* \langle \overline{w'} \mid \theta \mid \Phi' \mid B \rangle$, is true when $\Phi$ may be simplified to $\Phi'$ by any sequence of single-step simplifications. Its two rules appear in Figure 17. The judgement $\langle \Phi_0 \mid \overline{a} \mid \Phi_1 \rangle \triangleright^c \langle \theta \mid \Phi_2 \mid B \rangle$ is true when $\Phi_1$ may be simplified to $\Phi_2$, assuming the user-supplied constraints in $\Phi_0$. Its single rule, also in Figure 17, simply collapses $\Phi_0$ and $\Phi_1$, invokes the above judgement, and extracts from the result all constraints which are not a consequence of any user-supplied constraints.

## A.7 Incompleteness of Simplification

The simplifier is *incomplete* with respect to the notion of constraint entailment in a number of ways.

Firstly, to keep the complexity of constraint simplification manageable, the simplifier only considers closed class constraints in isolation. To see how this effects type inference, consider the program:

```
class closed C a where a
class closed D a where a
instance C (Int -> Int) = ... in
instance C (Bool -> Int) = ... in
instance D (Int -> Int) = ... in   -- (*)
instance D (Char -> Int) = ... in
let f x = C x + D x in
f 1
```

Even though `f` is well-typed with type `Int -> Int`, type inference only manages to assign `f` the type scheme:

```
forall a . (C (a -> Int), D (a -> Int)) => a -> Int
```

The simplifier could only discover that

$$w_1 : \texttt{C (a -> Int)}, w_2 : \texttt{D (a -> Int)}$$

entails that `a` must be `Int` if it were to systematically enumerate all combinations of instance schemes for `C` and `D`.

As a slight variation, consider the above program with the line marked `(*)` replaced by:

```
instance D (String -> Int) = ... in
```

Now `f` is ill-typed, but type inference continues to assign `f` the above type scheme. The program shall be rejected as ill-typed only when the application of `f` to `1` is considered.

The second source of incompleteness arises as a consequence of the first. When constructing a set of candidate instances, those instances which would lead to an unsatisfiable constraint context should be ignored. However, since we use the simplifier itself to test for unsatisfiability, and the simplifier is incomplete, the set of candidate instance schemes may be larger than necessary. This, in turn, may prevent the improvement or elimination of a class constraint.

## A.8 Finiteness and Termination

For this workshop paper we won't present any complexity results for deciding entailment or simplification. However, we shall at least briefly consider their decidability.

As presented so far, $\lambda^{\mathsf{O}}$ has adopted a somewhat *laisser faire* attitude to instance schemes. We have said nothing to reject instance schemes such as `forall a . C a => C a`, or pairs such as `forall a . D a => C a` and `forall a . C a => D a`. However, as mentioned in Section 5.5, it's easy to see such schemes could lead to infinite derivations within the $\vdash^c$ judgement, or infinite loops within the simplifier.

To avoid this we require that in an instance scheme `forall` $\overline{a}$ `.` $\overline{\xi}$ `=>` $C\ \overline{\tau}$, at least one $\tau_i$ must be of the form $F\ \overline{v}$ for some type constructor $F$, and that each $\xi_i$ is of the form $D\ \overline{b}$. Though draconian, these restrictions are easy to enforce and easy to motivate.

Alas, for closed classes even this restriction is not strong enough to prevent the simplifier from looping. For example, assume the type declaration:

```
class closed C a where a -> a
```

and the term:

```
instance a . C a => C [a] = map C
in \x . C x
```

Type inference shall construct a constraint context containing

$$w : \texttt{forall a . C a => C [a]}$$

(arising from the instance declaration), and

$$w' : \texttt{C b}$$

(arising from the application of `C`). Notice that the instance scheme is legal by the rules given above.

Now consider the action of the simplifier on this constraint context. Since `C` is closed, the constraint `C b` may be improved to `C [b']` for fresh `b'` using rule S10. This constraint may then be eliminated using rule S11 to yield the new constraint context:

$$w : \texttt{forall a . C a => C [a]}, w'' : \texttt{C b'}$$

Hence the simplifier loops.

The culprit is improvement: even though applying any instance scheme would lead to a constraint context containing smaller types, improvement may causes some types to grow again. In order to prevent such loops, we must impose a much stronger restriction on instance schemes for closed classes: we disallow any instance scheme which mentions a closed class in its context.

Thankfully, this restriction does not effect the encoding of member constraints.

## A.9 Adding Subtype Constraints

In this section we'll augment $\lambda^{\mathsf{O}}$ with subtyping constraints. Though not strictly necessary for encoding the methods of .NET classes, subtyping constraints make it possible to encode covariant methods and some aspects of generic classes.

| | | | |
|---|---|---|---|
| Atoms | $r, s$ | $::= a \mid A$ | |
| Type decls | $decl$ | $::= \texttt{newtype}\ A\ \texttt{:<:}\ \overline{B} \mid \ldots$ | |
| Prim constraints | $\phi, \xi$ | $::= \tau\ \texttt{:=:}\ \upsilon \mid \tau\ \texttt{:<:}\ \upsilon \mid \ldots$ | |
| Witness terms | $W$ | $::= \mathsf{True}$ | Trivial witness |
| | | $\mid \ldots$ | |

**Figure 20:** Changes to $\lambda^{\mathsf{O}}$ source and run-time syntax for subtyping constraints

Figure 20 presents the augmented syntax. Atoms, ranged over by $r$ and $s$, are type variables or newtype names. Though they do not appear within the source syntax, they shall be necessary when we consider entailment for subtype constraints.

Newtype declarations now take the form `newtype` $A$ `:<:` $\overline{B}$. This declares $A$ to be a newtype which is a subtype of each of $B_i$, and is intended to capture the subtyping hierarchy between classes $A$ and each $B_i$. We shall assume all the newtype declarations of a program form a partially ordered set. This rules out cyclic definitions such as:

```
newtype A :<: B
newtype B :<: A
```

However, we *do not* require this poset to be a lattice, nor even an upper semi-lattice. In particular, we do not assume any two pairs of newtypes have a lub or glb, nor assume there is a least or greatest newtype. This is a weaker assumption than typically adopted in the literature (for example [10]), but is necessary to support class hierarchies.

Primitive constraints are augmented by equality constraints ($\tau$ `:=:` $\upsilon$) and subtype constraints ($\tau$ `:<:` $\upsilon$). Subtype constraints have the usual meaning, and equality constraints are convenient shorthand for pairs of subtype constraints.

Equality and subtype constraints are witnessed by the trivial witness `True`. Of course, no such witness is required at runtime, and we shall elide these witnesses wherever possible.

Figure 21 refines the definitions of *inheritable* and *generalizable* to include the new primitive constraint forms. This figure also defines helper functions for encoding and querying the subtype hierarchy as a poset.

The largest addition to $\lambda^{\mathsf{O}}$ is two new entailment relations: $\Phi \vdash^s \tau$ `:<:` $\upsilon$ for deciding subtype entailment, and $\Phi \vdash^q \tau$ `:=:` $\upsilon$ for deciding equality entailment. As is standard [22], these entailment relations assume $\Phi$ is a conjuction of atomic subtype constraints. Thus, deciding subtype entailment for an arbitrary constraint $\Phi$ requires all the equality and non-atomic constraints in $\Phi$ to be "normalized." Figure 23 defines the function *normeqsubs* which, given an arbitrary constraint $\Phi$, calculates the most general *matching substitution* for $\Phi$ (see [22] for details), applies this substitution, and extracts the remaining atomic subtype constraints. Most of the work is done by the functions *match* and *mgus* (already defined in Figure 9).

The entailment relation itself is fairly standard [10]. We do not test for unsatisfiability of $\Phi$, hence given the type declaration:

$$
\begin{aligned}
eqs(\Phi) &= \{w : \tau \coloneqq \upsilon \mid (w : \tau \coloneqq \upsilon) \in \Phi\} \\
subs(\Phi) &= \{w : \tau \coloneqq \upsilon \mid (w : \tau \coloneqq \upsilon) \in \Phi\} \\
eqsubs(\Phi) &= eqs(\Phi) \cup subs(\Phi)
\end{aligned}
$$

$$
\begin{aligned}
&inheritable(\tau \coloneqq \upsilon) \iff \mathbf{tt} \\
&inheritable(\tau \coloneqq \upsilon) \iff \mathbf{tt}
\end{aligned}
$$

$$
\begin{aligned}
&generalizable(\tau \coloneqq \upsilon) \iff \mathbf{tt} \\
&generalizable(\tau \coloneqq \upsilon) \iff \mathbf{tt}
\end{aligned}
$$

$$
newtypes = \{A \mid (\texttt{newtype}\ A \coloneqq \overline{B}) \in decls\}
$$

$$
A \leq B \Leftrightarrow (A, B) \in tranrefclosure(G)
$$
$$
\text{where } G = \left\{ (A, B') \;\middle|\; \begin{array}{l} (\texttt{newtype}\ A \coloneqq \overline{B}) \in decls, \\ B' \in \overline{B} \end{array} \right\}
$$

$$
lub(\overline{A}) = B
$$
$$
\text{where } \forall i \,.\, A_i \leq B
$$
$$
\text{and } \forall B' \in newtypes \,.\, (\forall i \,.\, A_i \leq B') \Longrightarrow B \leq B'
$$

$$
glb(\overline{A}) = B
$$
$$
\text{where } \forall i \,.\, B \leq A_i
$$
$$
\text{and } \forall B' \in newtypes \,.\, (\forall i \,.\, B' \leq A_i) \Longrightarrow B' \leq B
$$

$$
maximals = \left\{ A \in newtypes \;\middle|\; \begin{array}{l} \forall B \in newtypes \,. \\ \quad A \leq B \Longrightarrow B \leq A \end{array} \right\}
$$

$$
minimals = \left\{ A \in newtypes \;\middle|\; \begin{array}{l} \forall B \in newtypes \,. \\ \quad B \leq A \Longrightarrow A \leq B \end{array} \right\}
$$

**Figure 21:** Additional ancillary definitions for subtype constraints

```
newtype A :<: B
```

the entailment:

$$\texttt{a :<: A, B :<: a} \vdash^s \texttt{() :<: ((), ())}$$

is not provable. Furthermore, since we do not assume that the primitive subtype relation $\leq$ is a lattice, some rules must be careful to check that lubs and glbs of types actually exist.

We may now augment the class entailment judgement $\Phi \vdash^c \phi \hookrightarrow W$ with two additional rules for when $\phi$ is a subtype or equality constraint. Rules CSUB and CEQ, defined in Figure 24, first collect all equality and subtype constraints available either directly or by projection from $\Phi$. They then normalize these constraints, and invoke the appropriate primitive entailment judgement.

The witness order must also be augmented to include the new witness True.

One particularly troublesome consequence of introducing general subtype constraints is that testing for equality of types, which so far has been syntactic, must now be done by testing for entailment of an equality constraint. Thus, we must now take care to replace all of the implicit tests for equality in $\lambda^{\mathsf{O}}$ with explicit tests. Such a test occurs in the entailment rule CINST: Figure 24 contains the refined rule. The type checking rules APP and INST must be similarly refined: these are given in Figure 25.

All that remains is to refine the simplifier to deal with subtype constraints. The additional rule for single-step simplifications are given in Figures 26 and 27.

Rules S15–S21 implement simplifications for structural subtyping and matching, and eliminate trivial subtype constraints.

Rule S22 eliminates subtype constraints which are a consequence of the transitive closure of other subtype constraints.

Rule S23 detects cycles, replacing the cyclic subtype constraints with equality constraints. To illustrate, assume the type definition:

```
newtype A :<: B
```

$$\boxed{\Phi \vdash^s \tau \coloneqq \upsilon}$$

$$\frac{A \leq B}{\Phi \vdash^s A \coloneqq B}\ \text{STAUT}$$

$$\frac{}{\Phi \vdash^s \tau \coloneqq \tau}\ \text{SREF}$$

$$\frac{\Phi \vdash^s r \coloneqq r' \quad \Phi \vdash^s r' \coloneqq s}{\Phi \vdash^s r \coloneqq s}\ \text{STRAN}$$

$$\frac{(r \coloneqq s) \in \Phi}{\Phi \vdash^s r \coloneqq s}\ \text{SID}$$

$$\frac{\Phi \vdash^s \overline{\tau \coloneqq \upsilon}}{\Phi \vdash^s (\overline{\tau}) \coloneqq (\overline{\upsilon})}\ \text{STUPLE}$$

$$\frac{\Phi \vdash^s \tau' \coloneqq \tau \quad \Phi \vdash^s \upsilon \coloneqq \upsilon'}{\Phi \vdash^s (\tau \texttt{ -> } \upsilon) \coloneqq (\tau' \texttt{ -> } \upsilon')}\ \text{SFUNC}$$

$$\frac{\Phi \vdash^s A_1 \coloneqq a \quad \Phi \vdash^s A_2 \coloneqq a \quad B = lub(\{A_1, A_2\})\ \text{well-defined}}{\Phi \vdash^s B \coloneqq a}\ \text{SLUB}$$

$$\frac{\Phi \vdash^s a \coloneqq A_1 \quad \Phi \vdash^s a \coloneqq A_2 \quad B = glb(\{A_1, A_2\})\ \text{well-defined}}{\Phi \vdash^s a \coloneqq B}\ \text{SGLB}$$

$$\boxed{\Phi \vdash^q \tau \coloneqq \upsilon}$$

$$\frac{\Phi \vdash^s A \coloneqq a \quad A \in maximals}{\Phi \vdash^q a \coloneqq A}\ \text{EMAX}$$

$$\frac{\Phi \vdash^s a \coloneqq A \quad A \in minimals}{\Phi \vdash^q a \coloneqq A}\ \text{EMIN}$$

$$\frac{\Phi \vdash^s \tau \coloneqq \upsilon \quad \Phi \vdash^s \upsilon \coloneqq \tau}{\Phi \vdash^q \tau \coloneqq \upsilon}\ \text{EEQ}$$

**Figure 22:** Atomic subtype and equality entailment. $\Phi$ contains only atomic subtype constraints.

$$
\begin{array}{l}
match(\texttt{true}) = \texttt{true} \\
match(a \mathrel{:<:} (\overline{\tau}), \Phi) = a \mathrel{:=:} (\overline{b}), match(\overline{b \mathrel{:<:} \tau} +\!\!+ \Phi) \\
\qquad\qquad \text{where } a \notin fv((\overline{\tau})) \text{ and } \overline{b} \text{ fresh} \\
match((\overline{\tau}) \mathrel{:<:} a, \Phi) = a \mathrel{:=:} (\overline{b}), match(\overline{\tau \mathrel{:<:} b} +\!\!+ \Phi) \\
\qquad\qquad \text{where } a \notin fv((\overline{\tau})) \text{ and } \overline{b} \text{ fresh} \\
match(a \mathrel{:<:} (\tau \texttt{ -> } \upsilon), \Phi) = \\
\qquad a \mathrel{:=:} (b \texttt{ -> } b'), match(\tau \mathrel{:<:} b, b' \mathrel{:<:} \upsilon, \Phi) \\
\qquad\quad \text{where } a \notin fv(\tau \texttt{ -> } \upsilon) \text{ and } b,\ b' \text{ fresh} \\
match((\tau \texttt{ -> } \upsilon) \mathrel{:<:} a, \Phi) = \\
\qquad a \mathrel{:=:} (b \texttt{ -> } b'), match(b \mathrel{:<:} \tau, \upsilon \mathrel{:<:} b', \Phi) \\
\qquad\quad \text{where } a \notin fv(\tau \texttt{ -> } \upsilon) \text{ and } b,\ b' \text{ fresh} \\
match(\tau \mathrel{:<:} \tau, \Phi) = match(\Phi) \\
match((\overline{\tau}) \mathrel{:<:} (\overline{\upsilon}), \Phi) = match(\overline{\tau \mathrel{:<:} \upsilon} +\!\!+ \Phi) \\
match((\tau \texttt{ -> } \upsilon) \mathrel{:<:} (\tau' \texttt{ -> } \upsilon'), \Phi) = \\
\qquad\qquad match(\tau' \mathrel{:<:} \tau, \upsilon \mathrel{:<:} \upsilon', \Phi) \\
match(r \mathrel{:<:} s, \Phi) = r \mathrel{:<:} s, match(\Phi)
\end{array}
$$

$$
\begin{array}{l}
normeqsubs(\theta \mid \Phi) = \\
\quad \textbf{case } mgus(\emptyset \mid eqs(\Phi)) \textbf{ of} \\
\qquad \emptyset \quad \to (\textbf{Id} \mid \texttt{false}) \\
\qquad \{\theta'\} \to \textbf{let } \Phi' = match(\theta' \; subs(\Phi)) \\
\qquad\qquad \textbf{in if } \Phi' = \Phi \textbf{ then } (\theta \mid \Phi) \\
\qquad\qquad\quad \textbf{else } normeqsubs(\theta' \circ \theta \mid \theta' \; \Phi')
\end{array}
$$

**Figure 23:** The function *normeqsubs*

Then the unsatisfiable constraint:

$$\texttt{a :<: A}, \texttt{B :<: a}$$

which appears to the simplifier (because of the revised definition of $\triangleright^c$, see below) as:

$$\texttt{a :<: A}, \texttt{A :<: B}, \texttt{B :<: a}$$

may be rewritten by this rule to:

$$\texttt{a :=: A}, \texttt{A :=: B}$$

This constraint may in turn be rewritten by the rules for equality constraints to `false`.

Rule s24 detects subtype constraints which cannot be supported by the primitive subtype relation. For example, assuming the declarations:

```
newtype A
newtype B
```

then the constraint:

$$w_1 : \texttt{A :<: a}, w_2 : \texttt{a :<: B}$$

cannot be supported, and is simplified to `false`.

Rule s25 and s26 exploit the lub and maximality properties of the primitive subtype poset. Notice that there are no analogous rules for glb and minimality, since the primitive subtype relation may always be extended downwards by additional newtype declarations.

Rules s27 and s28 eliminate subtype constraints involving a unique type variable. Similarly, rule s29 takes the transitive closure of a set of subtype constraints whenever this would eliminate a type variable which otherwise cannot be subject to further constraints.

Finally, rule s30 adds to the current constraint context any equality and subtype constraints available by projection. Clearly this rule must be used with care, as it grows the size of the current constraint context arbitrarily.

$$\boxed{\Phi \vdash W \leq W'}$$

$$\frac{}{\Phi \vdash \mathsf{True} \leq \mathsf{True}}$$

$$\boxed{\Phi \vdash^c \phi \hookrightarrow W}$$

$$\frac{\begin{array}{c}\Phi' = \{\xi \mid (\Phi \vdash^p \xi \hookrightarrow W)\} \\ (\theta \mid \Xi) = normeqsubs(eqsubs(\Phi')) \\ \Xi \vdash^s \theta\ \tau \mathrel{:<:} \theta\ \upsilon\end{array}}{\Phi \vdash^c \tau \mathrel{:<:} \upsilon \hookrightarrow \mathsf{True}}\ \text{CSUB}$$

$$\frac{\begin{array}{c}\Phi' = \{\xi \mid (\Phi \vdash^p \xi \hookrightarrow W)\} \\ (\theta \mid \Xi) = normeqsubs(eqsubs(\Phi')) \\ \Xi \vdash^q \theta\ \tau \mathrel{:=:} \theta\ \upsilon\end{array}}{\Phi \vdash^c \tau \mathrel{:=:} \upsilon \hookrightarrow \mathsf{True}}\ \text{CEQ}$$

$$\frac{\begin{array}{c}\Phi \vdash^p \texttt{forall } \overline{a} \texttt{ . } \overline{\phi} \texttt{ => } C\ \overline{\upsilon} \hookrightarrow W \\ \Phi \vdash^c \overline{[a \mapsto \upsilon']}\ \phi \hookrightarrow W' \\ \Phi \vdash^c \overline{[a \mapsto \upsilon']}\ \upsilon \mathrel{:=:} \tau\end{array}}{\Phi \vdash^c C\ \overline{\tau} \hookrightarrow W\ \overline{\upsilon'}\ \overline{W'}}\ \text{CINST}$$

**Figure 24:** Changes to witness ordering and class entailment for subtype constraints

$$\boxed{\Delta \mid \Phi \mid \Gamma \vdash t : \tau \hookrightarrow T}$$

$$\frac{\begin{array}{c}\Delta \mid \Phi \mid \Gamma \vdash t : \upsilon \hookrightarrow T \\ \Delta \mid \Phi \mid \Gamma \vdash u : \upsilon' \hookrightarrow U \\ \Phi \vdash^e \upsilon \mathrel{:=:} (\upsilon' \texttt{ -> } \tau)\end{array}}{\Delta \mid \Phi \mid \Gamma \vdash t\ u : \tau \hookrightarrow T\ U}\ \text{APP}$$

$$\frac{\begin{array}{c}(\texttt{class } \overline{m}\ \overline{\xi'} \texttt{ => } C\ \overline{c} \texttt{ where } \upsilon') \in decls \\ C \in fcv(t) \quad \overline{a} \cap \overline{b} \cap \overline{c} = \emptyset \quad w \text{ fresh} \\ \Xi = named(\overline{\xi}) \quad \overline{a} +\!\!+ \overline{b} \vdash \Xi\ \mathsf{constraint} \quad \overline{a} +\!\!+ \overline{b} \vdash \overline{\upsilon\ \mathsf{type}} \\ \phi = \texttt{forall } \overline{b} \texttt{ . } \overline{\xi} \texttt{ => } C\ \overline{\upsilon} \quad gens(\Xi) = \Xi \\ (\Gamma \neq \cdot) \vee (inhs(\Xi) \neq \Xi) \Longrightarrow \texttt{local} \in \overline{m} \\ inhs(\Phi) +\!\!+ \Xi \vdash^e \overline{[c \mapsto \upsilon]}\ \xi' \hookrightarrow W \\ \overline{a} +\!\!+ \overline{b} \mid inhs(\Phi) +\!\!+ \Xi \mid \Gamma \vdash u : \tau' \hookrightarrow U \\ \Phi \vdash^e \tau' \mathrel{:=:} [\overline{c \mapsto \upsilon}]\ \upsilon' \\ \Phi' = extend(\Phi \mid w : \phi) \text{ well-defined} \\ \overline{a} \mid \Phi' \mid \Gamma \vdash t : \tau \hookrightarrow T\end{array}}{\begin{array}{l}\overline{a} \mid \Phi \mid \Gamma \vdash \\ \quad \texttt{instance } \overline{b} \texttt{ . } \overline{\xi} \texttt{ => } C\ \overline{\upsilon} \texttt{ = } u \texttt{ in } t : \tau \hookrightarrow \\ \qquad \mathsf{letw}\ w = \Lambda \overline{b}\ .\ \lambda names(\Xi)\ .\ C\ \overline{W}\ U\ \mathsf{in}\ T\end{array}}\ \text{INST}$$

**Figure 25:** Changes to well-typing rules for subtype constraints

**Simple Subtype Constraints**

$$\frac{}{\langle \overline{w} \mid \overline{a} \mid w : r \mathrel{:<:} r, \Phi \rangle \triangleright \langle \emptyset \mid \mathbf{Id} \mid \Phi \mid \cdot \rangle} \text{ s15}$$

$$\frac{\begin{array}{c} b' \notin fv((\overline{\tau})) \quad \overline{b} \text{ fresh} \quad w' \text{ fresh} \\ S = \mathbf{if}\ w \in \overline{w}\ \mathbf{then}\ \{w'\}\ \mathbf{else}\ \emptyset \end{array}}{\begin{array}{l} \langle \overline{w} \mid \overline{a} \mid w : b' \mathrel{:<:} (\overline{\tau}), \Phi \rangle \triangleright \\ \quad \langle S \mid \mathbf{Id} \mid w' : b' \mathrel{:=:} (\overline{b}), w : b' \mathrel{:<:} (\overline{\tau}), \Phi \mid \cdot \rangle \end{array}} \text{ s16}$$

$$\frac{\begin{array}{c} b' \notin fv((\overline{\tau})) \quad \overline{b} \text{ fresh} \quad w' \text{ fresh} \\ S = \mathbf{if}\ w \in \overline{w}\ \mathbf{then}\ \{w'\}\ \mathbf{else}\ \emptyset \end{array}}{\begin{array}{l} \langle \overline{w} \mid \overline{a} \mid w : (\overline{\tau}) \mathrel{:<:} b', \Phi \rangle \triangleright \\ \quad \langle S \mid \mathbf{Id} \mid w' : b' \mathrel{:=:} (\overline{b}), w : (\overline{\tau}) \mathrel{:<:} b', \Phi \mid \cdot \rangle \end{array}} \text{ s17}$$

$$\frac{\begin{array}{c} b' \notin fv(\tau_1 \texttt{ -> } \tau_2) \quad \overline{b} \text{ fresh} \quad w' \text{ fresh} \\ S = \mathbf{if}\ w \in \overline{w}\ \mathbf{then}\ \{w'\}\ \mathbf{else}\ \emptyset \end{array}}{\begin{array}{l} \langle \overline{w} \mid \overline{a} \mid w : b' \mathrel{:<:} (\tau_1 \texttt{ -> } \tau_n), \Phi \rangle \triangleright \\ \quad \langle S \mid \mathbf{Id} \mid w' : b' \mathrel{:=:} b_1 \texttt{ -> } b_2, w : b' \mathrel{:<:} (\tau_1 \texttt{ -> } \tau_n), \Phi \mid \cdot \rangle \end{array}} \text{ s18}$$

$$\frac{\begin{array}{c} b' \notin fv(\tau_1 \texttt{ -> } \tau_2) \quad \overline{b} \text{ fresh} \quad w' \text{ fresh} \\ S = \mathbf{if}\ w \in \overline{w}\ \mathbf{then}\ \{w'\}\ \mathbf{else}\ \emptyset \end{array}}{\begin{array}{l} \langle \overline{w} \mid \overline{a} \mid w : (\tau_1 \texttt{ -> } \tau_2) \mathrel{:<:} b', \Phi \rangle \triangleright \\ \quad \langle S \mid \mathbf{Id} \mid w' : b' \mathrel{:=:} b_1 \texttt{ -> } b_2, w : (\tau_1 \texttt{ -> } \tau_2) \mathrel{:<:} b', \Phi \mid \cdot \rangle \end{array}} \text{ s19}$$

$$\frac{\overline{w'} \text{ fresh} \quad S = \mathbf{if}\ w \in \overline{w}\ \mathbf{then}\ \overline{w'}\ \mathbf{else}\ \emptyset}{\begin{array}{l} \langle \overline{w} \mid \overline{a} \mid w : (\overline{\tau}) \mathrel{:<:} (\overline{\upsilon}), \Phi \rangle \triangleright \\ \quad \langle S \mid \mathbf{Id} \mid \overline{w' : \tau \mathrel{:<:} \upsilon} +\!\!+ \Phi \mid \cdot \rangle \end{array}} \text{ s20}$$

$$\frac{\overline{w'} \text{ fresh} \quad S = \mathbf{if}\ w \in \overline{w}\ \mathbf{then}\ \overline{w'}\ \mathbf{else}\ \emptyset}{\begin{array}{l} \langle \overline{w} \mid \overline{a} \mid w : (\tau_1 \texttt{ -> } \tau_2) \mathrel{:<:} (\upsilon_1 \texttt{ -> } \upsilon_2), \Phi \rangle \triangleright \\ \quad \langle S \mid \mathbf{Id} \mid \overline{w' : \upsilon \mathrel{:<:} \tau} +\!\!+ \Phi \mid \cdot \rangle \end{array}} \text{ s21}$$

**Figure 26:** Additional simplifier rules for subtype constraints (part 1 of 2)

Figures 26 and 27 are also interesting for the rules they *exclude*. Applicative languages with subtyping typically allow a type coercion at *every* function application. For example, if `B` $\leq$ `A` and `f :: A -> B` then `f` may be applied to an argument of type `B` by default. As a result, a function of type:

```
forall a :<: A, B :<: b => a -> b
```

may be applied exactly where a function of type:

```
A -> B
```

may be applied, so the former type is simply more verbose than the later. Simplifiers for "subtyping everywhere" languages exploit this situation by including rules to *minimize* covariant type variables, and *maximize* contravariant variables. Indeed, these rules form a cornerstone of the simplifier.

However, $\lambda^{\mathsf{O}}$ is not a "subtyping everywhere" language: the type checking rule APP requires argument types to match

**Advanced Subtype Constraints**

$$\frac{1 \leq n \quad w_n \notin \overline{w}}{\begin{array}{l} \langle \overline{w} \mid \overline{a} \mid w_1 : r_1 \mathrel{:<:} r_2, \ldots, w_{n-1} : r_{n-1} \mathrel{:<:} r_n, \\ \qquad w_n : r_1 \mathrel{:<:} r_n, \Phi \rangle \triangleright \\ \quad \langle \emptyset \mid \mathbf{Id} \mid w_1 : r_1 \mathrel{:<:} r_2, \ldots, \\ \qquad w_{n-1} : r_{n-1} \mathrel{:<:} r_n, \Phi \mid \cdot \rangle \end{array}} \text{ s22}$$

$$\frac{2 \leq n}{\begin{array}{l} \langle \overline{w} \mid \overline{a} \mid w_1 : r_1 \mathrel{:<:} r_2, \ldots, w_n : r_n \mathrel{:<:} r_1, \Phi \rangle \triangleright \\ \quad \langle \emptyset \mid \mathbf{Id} \mid \ w_1 : r_1 \mathrel{:=:} r_2, \ldots, \\ \qquad w_{n-1} : r_{n-1} \mathrel{:=:} r_n, \Phi \mid \cdot \rangle \end{array}} \text{ s23}$$

$$\frac{0 \leq n \quad A \not\leq B}{\begin{array}{l} \langle \overline{w} \mid \overline{a} \mid w_1 : A \mathrel{:<:} r_1, \ldots, w_n : r_n \mathrel{:<:} B, \Phi \rangle \triangleright \\ \quad \langle \emptyset \mid \mathbf{Id} \mid \texttt{false} \mid \cdot \rangle \end{array}} \text{ s24}$$

$$\frac{\begin{array}{c} n > 1 \quad B = lub(\overline{A}) \text{ exists} \quad w'' \text{ fresh} \\ S = \mathbf{if}\ \overline{w'} \subseteq \overline{w}\ \mathbf{then}\ \{w''\}\ \mathbf{else}\ \emptyset \end{array}}{\begin{array}{l} \langle \overline{w} \mid \overline{a} \mid w'_1 : A_1 \mathrel{:<:} b, \ldots, w'_n : A_n \mathrel{:<:} b, \Phi \rangle \triangleright \\ \quad \langle S \mid \mathbf{Id} \mid w'' : B \mathrel{:<:} b, \Phi \mid \cdot \rangle \end{array}} \text{ s25}$$

$$\frac{A \in maximals}{\langle \overline{w} \mid \overline{a} \mid w : A \mathrel{:<:} b, \Phi \rangle \triangleright \langle \emptyset \mid \mathbf{Id} \mid w : A \mathrel{:=:} b, \Phi \mid \cdot \rangle} \text{ s26}$$

$$\frac{b \notin \overline{a} \cup fv(\Phi)}{\langle \overline{w} \mid \overline{a} \mid w : r \mathrel{:<:} b, \Phi \rangle \triangleright \langle \emptyset \mid \mathbf{Id} \mid \Phi \mid \cdot \rangle} \text{ s27}$$

$$\frac{b \notin \overline{a} \cup fv(\Phi)}{\langle \overline{w} \mid \overline{a} \mid w : b \mathrel{:<:} r, \Phi \rangle \triangleright \langle \emptyset \mid \mathbf{Id} \mid \Phi \mid \cdot \rangle} \text{ s28}$$

$$\frac{\begin{array}{c} b \notin \overline{a} \cup fv(\Phi) \quad 0 < m, n \quad \overline{w''} \text{ fresh} \\ S = \mathbf{if}\ \overline{w'} \subseteq \overline{w}\ \mathbf{then}\ \overline{w''}\ \mathbf{else}\ \emptyset \end{array}}{\begin{array}{l} \langle \overline{w} \mid \overline{a} \mid \ w'_1 : r_1 \mathrel{:<:} b, \ldots, w'_m : r_m \mathrel{:<:} b, \\ \qquad w'_{m+1} : b \mathrel{:<:} s_1, \ldots, w'_{m+n} : b \mathrel{:<:} s_n, \Phi \rangle \triangleright \\ \quad \langle S \mid \mathbf{Id} \mid (+\!\!+_{i,j}(w''_{i,j} : r_i \mathrel{:<:} s_j)) +\!\!+ \Phi \mid \cdot \rangle \end{array}} \text{ s29}$$

$$\frac{\Xi = named(anon(eqsubs(closesup(w : C\ \overline{\tau}))))}{\begin{array}{l} \langle \overline{w} \mid \overline{a} \mid w : C\ \overline{\tau}, \Phi \rangle \triangleright \\ \quad \langle names(\Xi) \mid \mathbf{Id} \mid w : C\ \overline{\tau}, (\Phi +\!\!+ \Xi) \mid \cdot \rangle \end{array}} \text{ s30}$$

**Figure 27:** Additional simplifier rules for subtype constraints (part 2 of 2)

$$\frac{\begin{array}{c} \Phi_2 = named(\left\{ A \mathrel{:<:} B' \;\middle|\; \begin{array}{l} (\texttt{newtype}\ A \mathrel{:<:} \overline{B}) \in decls, \\ B' \in \overline{B} \end{array} \right\}) \\ \overline{w} = names(\Phi_0 +\!\!+ \Phi_2) \\ \langle \overline{w} \mid \overline{a} \mid \Phi_0 +\!\!+ \Phi_2 +\!\!+ \Phi_1 \rangle \triangleright^* \langle \overline{w'} \mid \theta \mid \Phi_3 \mid B \rangle \\ \Phi_4 = \{ w : \xi \mid (w : \xi) \in \Phi_3, w \notin \overline{w} \cup \overline{w'} \} \end{array}}{\langle \Phi_0 \mid \overline{a} \mid \Phi_1 \rangle \triangleright^c \langle \theta \mid \Phi_4 \mid B \rangle} \text{ SIMP}$$

**Figure 28:** Changes to simplifer rules for subtype constraints

function types, and Figures 7 and 25 don't include any non-syntax directed rule for coercion. As a result, the above two function types are not equivalent, and our simplifier should not, in general, minimize or maximize type variables.

These single-step simplifier rules assume the primitive subtype relation has been internalized within the constraint context being simplified. Figure 28 refines the definition of the top-level simplifier judgement to include these constraints upon each invocation of the simplifier.

### A.10 Weak Completeness

We're still pondering how to formulate a suitably weak notion of completeness of type inference with respect to type checking for $\lambda^{\mathsf{O}}$. Particularly troublesome is capturing the notion of closed classes.

For example, consider the program:

```
class closed C a where a
instance C Int = 1 in
let f x = C + 1 in
f 1
```

This program has two well-typing derivations, each assigning a different type scheme to `f`:

```
f :: forall a . (C a, Num a) => a -> a  -- (1)
f :: Int -> Int                         -- (2)
```

The program also has two type inference derivations, assigning the same two type schemes to `f`. At first glance this seems wrong: surely type (1) is a principal type for `f`, and thus inferring type (2) shows inference is incomplete?

However, what's at fault here are not the rules of type inference, but rather the notion of principal type: The instantiation ordering amongst type schemes must be relative to the available instances of closed classes. That is to say, closed classes impose the invariant:

> "The set of instances of closed class `C` available at the point of definition of `f` is equal to the set available at each occurrence of `f`."

Hence, the instantiation ordering must be chosen so as to quotient the type schemes of `f` appropriately. In our example, we should find that, relative to the constraint `C Int`, the two type schemes for `f` are equivalent under the instantiation ordering.

Using this machinery, we hope to be able to show that type inference is *weakly complete*: If inference succeeds, the inferred type is principal under the refined instantiation ordering.

Of course, we have already seen in Section 5.7 that type inference is not *strongly complete*: Type inference may reject programs which have valid well-typing derivations. However, we feel this deficiency, which is already present in Haskell, is easier to live with. Given sufficiently informative type errors, the programmer always has recourse to type annotations in order to encourage a recalcitrant complier to accept her program.