

COMP3161/COMP9164 Supplementary Lecture Notes

Overloading

Gabriele Keller

Liam O'Connor

Johannes Åman Pohjola

November 10, 2021

So far, all the operations we have in MinHS are either *monomorphic* or *polymorphic*. Monomorphic operations work on a specific type, for example addition $(+) : \mathbf{Int} \rightarrow \mathbf{Int} \rightarrow \mathbf{Int}$. Polymorphic operations work on *any type at all*, such as the identity function of type $\forall a. a \rightarrow a$.

In practice, this is not sufficient for a general purpose language. If we add, for example, floating point numbers to MinHS, we want at least all the operations we have on integers to be available on floats as well, so we need $(+_{\mathbf{Float}}) : \mathbf{Float} \rightarrow \mathbf{Float} \rightarrow \mathbf{Float}$. Note the operator name. The programmer must remember to use $+$ for adding **Ints**, and $+_{\mathbf{Float}}$ for adding **Floats**. This would make the language rather annoying to use. It gets even worse for a language with a more realistic set of types, including integers and floats of different size.

Even worse, consider an operation such as $==$. Ideally, $==$ should work not just on basic types, but also on compound types, like pairs and sum types, but not on function types.

We want a way to refer to multiple functions by the same name, and have the exact implementation determined by type of its arguments. This is called *overloading* or *ad-hoc polymorphism*.

1 Type Classes in Haskell

The idea behind type classes is to group a set of types together if they have several, conceptually similar operations in common. For example, in Haskell, the type class where the *methods* addition, multiplication, subtraction and such are defined is called **Num**, and contains the types **Int**, **Integer** (not fixed length), **Float** and **Double**.

For example, the type of addition is:

$$(+) : \forall a. \mathbf{Num} \ a \Rightarrow a \rightarrow a \rightarrow a$$

In words: for all types a in the type class **Num**, $(+)$ has the type $a \Rightarrow a \rightarrow a \rightarrow a$. The *constraint* **Num** restricts the types which addition can be applied to.

The **Eq** type class in Haskell contains all types whose elements can be tested for equality. When you type `:info Eq` into GHCi, the interpreter lists all its methods:

$$\begin{aligned} (==) &:: \mathbf{Eq} \ a \Rightarrow a \rightarrow a \rightarrow \mathbf{Bool} \\ (/=) &:: \mathbf{Eq} \ a \Rightarrow a \rightarrow a \rightarrow \mathbf{Bool} \end{aligned}$$

Also listed are the types which are in this type class. Basic integral types are included:

```
instance Eq Int
instance Eq Float
instance Eq Double
instance Eq Char
instance Eq Bool
```

Also, generative rules that specify instances that themselves depend on another instance:

```

instance Eq  $a \Rightarrow$  Eq [ $a$ ]
instance Eq  $a \Rightarrow$  Eq (Maybe  $a$ )
instance (Eq  $a$ , Eq  $b$ )  $\Rightarrow$  Eq ( $a$ ,  $b$ )

```

These are rules about type class membership: if a type a is in **Eq**, then lists of a are also in **Eq**, as well as **Maybe** a . If two types a and b are both in **Eq**, so are pairs of a and b . Operations on these compound types are implemented in terms of the operations of the argument types. So, two pairs of values are considered to be equal if both of their components are equal:

```

instance (Eq  $a$ , Eq  $b$ )  $\Rightarrow$  Eq ( $a$ ,  $b$ ) where
  ( $==$ ) ( $a_1$ ,  $b_1$ ) ( $a_2$ ,  $b_2$ ) = ( $a_1 == a_2$ ) && ( $b_1 == b_2$ )

```

And equality of lists can be defined, for example, as follows:

```

instance Eq  $a \Rightarrow$  Eq [ $a$ ] where
  ( $==$ ) [] [] = True
  ( $==$ ) ( $a : as$ ) ( $b : bs$ ) = ( $a == b$ ) && ( $as == bs$ )
  ( $==$ ) _ _ = False

```

Other examples of predefined type classes are **Show** and **Read**. A user can extend these type classes and define new classes.

2 Static Semantics

We write

$$f :: \forall a. P \Rightarrow \tau$$

to indicate that f has the type τ where a can be instantiated to any type *under the condition* that the constraint P is satisfied. Typically, P is a list of *instance constraints*, such as **Num** a or **Eq** b .

Extending implicitly typed MinHS with type classes, we allow constraints to occur on quantified type variables:

Predicates	$P ::=$	$\mathbf{C} \ \tau$
Polytypes	$\pi ::=$	$\tau \mid \forall a. \pi \mid \mathbf{P} \Rightarrow \pi$
Monotypes	$\tau ::=$	Int \mid Bool $\mid \tau + \tau \mid \dots$
Class names	C	

Our typing judgement $\Gamma \vdash e : \pi$ now includes a set of type class *axiom schema* \mathcal{A} :

$$\mathcal{A} \mid \Gamma \vdash e : \pi$$

This set contains predicates for all type class instances known to the compiler, including generative instances, which take the form of implications like **Eq** $a \Rightarrow$ **Eq** [a].

To adapt our existing typing rules to this new feature, we thread our axiom set \mathcal{A} through them but otherwise leave them unchanged. For example, here is the typing rule for **fst** before and after:

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{fst} \ e : \tau_1} \qquad \frac{\mathcal{A} \mid \Gamma \vdash e : \tau_1 \times \tau_2}{\mathcal{A} \mid \Gamma \vdash \mathbf{fst} \ e : \tau_1}$$

To use an overloaded type, we must first show that the predicate is *satisfied* by the known axioms, written $\mathcal{A} \Vdash P$:

$$\frac{\mathcal{A} \mid \Gamma \vdash e : P \Rightarrow \pi \quad \mathcal{A} \Vdash P}{e : \pi} \text{INST}$$

If, adding a predicate to the known axioms, we can conclude a typing judgement, then we can overload the expression with that predicate:

$$\frac{P, \mathcal{A} \mid \Gamma \vdash e : \pi}{\mathcal{A} \mid \Gamma \vdash e : P \Rightarrow \pi} \text{GEN}$$

Putting these rules to use, we could show that $3.2 + 4.4$, which uses the overloaded operator $(+)$, is of type `Float`:

1. We know that $(+) :: \forall a. (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a \in \Gamma$.
2. We know that $\text{Num Float} \in \mathcal{A}$.
3. Instantiating the type variable a , we can conclude that $(+) :: (\text{Num Float}) \Rightarrow \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}$.
4. Using the INST rule above and (3), we can conclude $(+) :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}$.
5. By the function application rule, we can conclude $3.2 + 4.4 :: \text{Float}$ as required.

Here's a more formal derivation, where we'll abbreviate `F` as `Float`.

$$\begin{array}{c}
\frac{(+): \forall a. (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a \in \Gamma}{\mathcal{A} \mid \Gamma \vdash (+): \forall a. (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a} \quad \frac{\text{Num F} \in \mathcal{A}}{\mathcal{A} \Vdash \text{Num F}} \\
\hline
\mathcal{A} \mid \Gamma \vdash (+): (\text{Num F}) \Rightarrow \text{F} \rightarrow \text{F} \rightarrow \text{F} \quad \mathcal{A} \Vdash \text{Num F} \\
\hline
\mathcal{A} \mid \Gamma \vdash (+): \text{F} \rightarrow \text{F} \rightarrow \text{F} \quad \mathcal{A} \mid \Gamma \vdash 3.2 : \text{F} \\
\hline
\mathcal{A} \mid \Gamma \vdash (+) 3.2 : \text{F} \rightarrow \text{F} \quad \mathcal{A} \mid \Gamma \vdash 4.4 : \text{F} \\
\hline
\mathcal{A} \mid \Gamma \vdash 3.2 + 4.4 : \text{F}
\end{array}$$

The following example derivation illustrates the use of the GEN rule. We will assume that $x : a \in \Gamma$ and $y : a \in \Gamma$. We can then derive a type for the expression $x + y$ as follows:

$$\begin{array}{c}
\frac{(+): \forall b. (\text{Num } b) \Rightarrow b \rightarrow b \rightarrow b \in \Gamma}{\text{Num } a, \mathcal{A} \mid \Gamma \vdash (+): \forall b. \text{Num } b \Rightarrow b \rightarrow b \rightarrow b} \\
\hline
\frac{\text{Num } a, \mathcal{A} \mid \Gamma \vdash (+): \text{Num } a \Rightarrow a \rightarrow a \rightarrow a}{\text{Num } a, \mathcal{A} \mid \Gamma \vdash (+): a \rightarrow a \rightarrow a} \quad \frac{x : a \in \Gamma}{\text{Num } a, \mathcal{A} \mid \Gamma \vdash x : a} \quad \frac{y : a \in \Gamma}{\text{Num } a, \mathcal{A} \mid \Gamma \vdash y : a} \\
\hline
\text{Num } a, \mathcal{A} \mid \Gamma \vdash (+) x : a \rightarrow a \quad \text{Num } a, \mathcal{A} \mid \Gamma \vdash x : a \quad \text{Num } a, \mathcal{A} \mid \Gamma \vdash y : a \\
\hline
\text{Num } a, \mathcal{A} \mid \Gamma \vdash x + y : a \\
\hline
\mathcal{A} \mid \Gamma \vdash x + y : \text{Num } a \Rightarrow a
\end{array}$$

This type has a free type variable, which is appropriate since x, y occur in the typing context. This derivation also uses the INST rule to massage the type of $(+)$; note that we elided the side condition $\text{Num } a, \mathcal{A} \Vdash \text{Num } a$.

3 Overloading Resolved

Up to this point, we only discussed the effect overloading has on the static semantics. What about the dynamic semantics? At some point, the overloaded function has to be instantiated to the correct concrete operation. This could happen at run time, but that doesn't seem to be the best way.

In object oriented languages, objects typically know what to do — that is, an object is associated with a so-called virtual method or dispatch table, which contains all the methods of the object's class. This approach isn't applicable to a language like MinHS, where arguments of an overloaded function aren't objects and can be of basic type, but we can use a similar idea: we pass the table of all the methods of a class to an overloaded function, and replace the overloaded function with one that simply picks the appropriate method from the table. We call this table a *dictionary*, and in MinHS, we represent a table with n functions as n -tuple.

Type classes are converted to type declarations for their dictionary:

```
class Eq a where
  (==) : a → a → Bool
  (/=) : a → a → Bool
```

becomes

```
type EqDict a = (a → a → Bool × a → a → Bool)
```

Instances become *values* of the dictionary type:

```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
  a  /=  b  = not (a == b)
```

becomes

```
True  ==Bool True  = True
False ==Bool False = True
_     ==Bool _     = False
a  /=Bool b  = not (a ==Bool b)
```

```
eqBoolDict = ((==Bool), (/=Bool))
```

Programs that rely on overloading now take dictionaries as parameters:

```
same :: ∀a. (Eq a) ⇒ [a] → Bool
same [] = True
same (x : []) = True
same (x : y : xs) = x == y ∧ same (y : xs)
```

Becomes:

```
same :: ∀a. (EqDict a) → [a] → Bool
same eq [] = True
same eq (x : []) = True
same eq (x : y : xs) = (fst eq) x y ∧ same eq (y : xs)
```

In some cases, we can make instances also predicated on some constraints:

```
instance (Eq a) ⇒ (Eq [a]) where
  []      == []      = True
  (x : xs) == (y : ys) = x == y ∧ (xs == ys)
  _       == _       = False
  a  /=  b  = not (a == b)
```

Such instances are transformed into *functions* to produce new dictionaries:

```
eqList :: EqDict a → EqDict [a]
```