COMP3161/COMP9164 Supplementary Lecture Notes

Overloading

Gabriele Keller, Liam O'Connor

November 11, 2019

So far, all the operations we have in MinHS are either *monomorphic* in that they work on a specific type, as for example addition $(+)$: `Int` $\rightarrow$ `Int` $\rightarrow$ `Int` or they are *polymorphic* in that they 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 $(+_{Float})$: `Float` $\rightarrow$ `Float` $\rightarrow$ `Float`. However, this would make the language pretty annoying to use, and get even worse for a language with a more realistic set of types, including integers and floats of different size.

The situation is even worse when we consider an operation like checking two values for equality. This should, ideally, not only work on basic types, but on also on compound types, like pairs and sum types (but not on function types).

What we desire is a way to refer to multiple functions by the same name, and have the exact implementation determined based on the 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 within which 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:

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

which reads as: for all types a in 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 pairwise equality. When you type `:info Eq` into GHCi, the interpreter lists all its methods:

$$(\texttt{==})\ ::\ \texttt{Eq}\, a\ \Rightarrow\ a\ \rightarrow\ a\ \rightarrow\ \texttt{Bool}$$
$$(\texttt{/=})\ ::\ \texttt{Eq}\, a\ \Rightarrow\ a\ \rightarrow\ a\ \rightarrow\ \texttt{Bool}$$

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:

$$
\begin{array}{l}
\textbf{instance}\ \texttt{Eq}\ a\ \Rightarrow\ \texttt{Eq}\ [a] \\
\textbf{instance}\ \texttt{Eq}\ a\ \Rightarrow\ \texttt{Eq}\ (\texttt{Maybe}\ a) \\
\textbf{instance}\ (\texttt{Eq}\ a,\ \texttt{Eq}\ b)\ \Rightarrow\ \texttt{Eq}\ (a,\ b)
\end{array}
$$

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:

$$
\begin{array}{l}
\textbf{instance}\ (\texttt{Eq}\ a, \texttt{Eq}\ b)\ \Rightarrow\ \texttt{Eq}\ (a, b)\ \textbf{where} \\
\quad (\texttt{==})\ (a_1, b_1)\ (a_2, b_2) = (a_1\ \texttt{==}\ a_2)\ \texttt{\&\&}\ (b_1\ \texttt{==}\ b_2)
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{instance}\ \texttt{Eq}\ a\ \Rightarrow\ \texttt{Eq}\ [a]\ \textbf{where} \\
\quad (\texttt{==})\ [\,]\ [\,] = \mathsf{True} \\
\quad (\texttt{==})\ (a : as)\ (b : bs) = (a\ \texttt{==}\ b)\ \texttt{\&\&}\ (as\ \texttt{==}\ bs) \\
\quad (\texttt{==})\ \_\ \_ = \mathsf{False}
\end{array}
$$

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	$::=$	$\texttt{C}\ \tau$
Polytypes	π	$::=$	$\tau \mid \forall a.\ \pi \mid P \Rightarrow \pi$
Monotypes	τ	$::=$	$\texttt{Int} \mid \texttt{Bool} \mid \tau + \tau \mid \cdots$
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 add typing rules for this, we leave the existing rules unchanged, save that they thread our axiom set $\mathcal{A}$ through.

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

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

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

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

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

3 Overloading Resolved

Up to this point, we only discussed the effect overloading has on the static semantics. But how 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 language, 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:

$$\begin{array}{l}\textbf{class}\ \texttt{Eq}\ a\ \textbf{where}\\ \quad (\texttt{==}) : a \rightarrow a \rightarrow \texttt{Bool}\\ \quad (\texttt{/=}) : a \rightarrow a \rightarrow \texttt{Bool}\end{array}$$

becomes

$$\textbf{type}\ \texttt{EqDict}\ a = (a \rightarrow a \rightarrow \texttt{Bool} \times a \rightarrow a \rightarrow \texttt{Bool})$$

Instances become *values* of the dictionary type:

$$\begin{array}{l}\textbf{instance}\ \texttt{Eq Bool}\ \textbf{where}\\ \quad \begin{array}{llll}\texttt{True} & \texttt{==} & \texttt{True} & = \texttt{True}\\ \texttt{False} & \texttt{==} & \texttt{False} & = \texttt{True}\\ \_ & \texttt{==} & \_ & = \texttt{False}\\ a & \texttt{/=} & b & = \texttt{not}\ (a\ \texttt{==}\ b)\end{array}\end{array}$$

becomes

$$\begin{array}{llll}\texttt{True} & \texttt{==}_{\texttt{Bool}} & \texttt{True} & = \texttt{True}\\ \texttt{False} & \texttt{==}_{\texttt{Bool}} & \texttt{False} & = \texttt{True}\\ \_ & \texttt{==}_{\texttt{Bool}} & \_ & = \texttt{False}\\ a & \texttt{/=}_{\texttt{Bool}} & b & = \texttt{not}\ (a\ \texttt{==}_{\texttt{Bool}}\ b)\end{array}$$

$$\mathit{eqBoolDict} = ((\texttt{==}_{\texttt{Bool}}), (\texttt{/=}_{\texttt{Bool}}))$$

Programs that rely on overloading now take dictionaries as parameters:

$$\begin{array}{l}\mathit{same} :: \forall a.\ (\texttt{Eq}\ a) \Rightarrow [a] \rightarrow \texttt{Bool}\\ \mathit{same}\ [] = \texttt{True}\\ \mathit{same}\ (x : []) = \texttt{True}\\ \mathit{same}\ (x : y : \mathit{xs}) = x\ \texttt{==}\ y \wedge \mathit{same}\ (y : \mathit{xs})\end{array}$$

Becomes:

$$
\begin{aligned}
&\mathit{same} :: \forall a.\ (\texttt{EqDict}\ a) \rightarrow [a] \rightarrow \texttt{Bool}\\
&\mathit{same}\ \mathit{eq}\ [\,] = \texttt{True}\\
&\mathit{same}\ \mathit{eq}\ (x : [\,]) = \texttt{True}\\
&\mathit{same}\ \mathit{eq}\ (x : y : \mathit{xs}) = (\mathsf{fst}\ \mathit{eq})\ x\ y \land \mathit{same}\ \mathit{eq}\ (y : \mathit{xs})
\end{aligned}
$$

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

$$
\begin{array}{l}
\textbf{instance}\ (\texttt{Eq}\ a) \Rightarrow (\texttt{Eq}\ [a])\ \textbf{where}\\
\quad \begin{array}{llcll}
[\,] & \texttt{==} & [\,] & = & \texttt{True}\\
(x : \mathit{xs}) & \texttt{==} & (y : \mathit{ys}) & = & x\ \texttt{==}\ y \land (\mathit{xs}\ \texttt{==}\ \mathit{ys})\\
\_ & \texttt{==} & \_ & = & \texttt{False}
\end{array}\\
\quad a\ \ \texttt{/=}\ \ b\ \ =\ \ \mathsf{not}\ (a\ \texttt{==}\ b)
\end{array}
$$

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

$$
\mathit{eqList} :: \texttt{EqDict}\ a \rightarrow \texttt{EqDict}\ [a]
$$