Overloading and Subtyping

Johannes Åman Pohjola
UNSW
Term 3 2022

myExperience

The myExperience survey is out.

I would be very grateful if you could take 5 minutes out of your day to go on Moodle and fill out the survey. Even if you don't have much to say.

Motivation

Suppose we added `Float` to MinHS.
Ideally, the arithmetic operations should be able to work on both `Int` and `Float`.

$$4 + 6 :: \texttt{Int}$$

$$4.3 + 5.1 :: \texttt{Float}$$

Similarly, a numeric literal should take on whatever type is inferred from context.

$$(5 :: \texttt{Int}) \bmod 3$$

$$\sin(5 :: \texttt{Float})$$

Without Overloading

We effectively have two functions:

$$(+_{\texttt{Int}}) :: \texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int}$$

$$(+_{\texttt{Float}}) :: \texttt{Float} \rightarrow \texttt{Float} \rightarrow \texttt{Float}$$

We would like to refer to both of these functions by the same name and have the specific implementation chosen based on the type.

Such type-directed name resolution is called *ad-hoc polymorphism* or *overloading*.

Type Classes

Type classes are a common approach to ad-hoc polymorphism, and exist in various languages under different names:

- Type Classes in Haskell
- Traits in Rust
- Implicits in Scala
- Protocols in Swift
- Contracts in Go 2
- Concepts in C++
- Other languages approximate with *subtype polymorphism* (coming)

Type Classes

A *type class* is a set of types for which implementations (*instances*) have been provided for various functions, called *methods*[1].

Example (Numeric Types)

In Haskell, the types `Int`, `Float`, `Double` etc. are all instances of the type class `Num`, which has methods such as $(+)$, *negate*, etc.

Example (Equality)

In Haskell, the `Eq` type class contains methods $(==)$ and $(/=)$ for computable equality. What types cannot be an instance of `Eq`?

[1] Nothing to do with OO methods.

Notation

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 $\texttt{Num}\ a$ or $\texttt{Eq}\ b$.

Example

- $(+) :: \forall a.\ (\texttt{Num}\ a) \Rightarrow a \rightarrow a \rightarrow a$
- $(==) :: \forall a.\ (\texttt{Eq}\ a) \Rightarrow a \rightarrow a \rightarrow \texttt{Bool}$

Is $(1 :: \texttt{Int}) + 4.4$ a well-typed expression?
No. The type of $(+)$ requires its arguments to have the same type.

Extending MinHS

Extending implicitly typed MinHS with type classes:

Predicates	P	$::=$	$\mathtt{C}\ \tau$
Polytypes	π	$::=$	$\tau \mid \forall a.\ \pi \mid P \Rightarrow\ \pi$
Monotypes	τ	$::=$	$\mathtt{Int} \mid \mathtt{Bool} \mid \tau + \tau \mid \cdots$
Class names	C		

Our typing judgement $\Gamma \vdash e : \pi$ now includes a set of type class axiom schema:

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

This set contains predicates for all type class instances known to the compiler.

Typing Rules

The existing rules now just thread $\mathcal{A}$ through.
To use an overloaded type, we must show that the predicate is satisfied by the known axioms:

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

Right now, $\mathcal{A} \Vdash P$ iff $P \in \mathcal{A}$, but we will complicate this situation later.

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

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

Example

Suppose we wanted to show that $3.2 + 4.4 :: \texttt{Float}$.

1. $(+) :: \forall a.\ (\texttt{Num}\ a) \Rightarrow a \rightarrow a \rightarrow a \in \Gamma$.
2. $\texttt{Num Float} \in \mathcal{A}$.
3. Using $\textsc{AllE}$ (from previous lecture), we can conclude $(+) :: (\texttt{Num Float}) \Rightarrow \texttt{Float} \rightarrow \texttt{Float} \rightarrow \texttt{Float}$.
4. Using $\textsc{Inst}$ (on previous slide) and 2, 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.

Dictionaries and Resolution

This is called *ad-hoc* polymorphism because the type checker removes it — it is not a fundamental language feature, but merely a naming convenience.

The type checker will convert ad-hoc polymorphism to parametric polymorphism.

Type classes are converted to types:

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

becomes

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

A *dictionary* contains all the method implementations of a type class for a specific type.

Dictionaries and Resolution

Instances become values of the dictionary type:

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

becomes

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

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

Dictionaries and Resolution

Programs that rely on overloading now take dictionaries as parameters:

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

Becomes:

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

Generative Instances

We can make instances also predicated on some constraints:

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

Such instances are transformed into functions:

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

Our set of axiom schema $\mathcal{A}$ now includes implications, like $(\texttt{Eq}\ a) \Rightarrow (\texttt{Eq}\ [a])$. This makes the relation $\mathcal{A} \Vdash P$ much more complex to solve.

Coherence

Some languages (such as Haskell and Rust) insist that there is only one instance per class per type in the entire program. It achieves this by requiring that all instances are either:

- Defined along with the definition of the type class, or
- Defined along with the definition of the type.

This rules out so-called *orphan* instances.

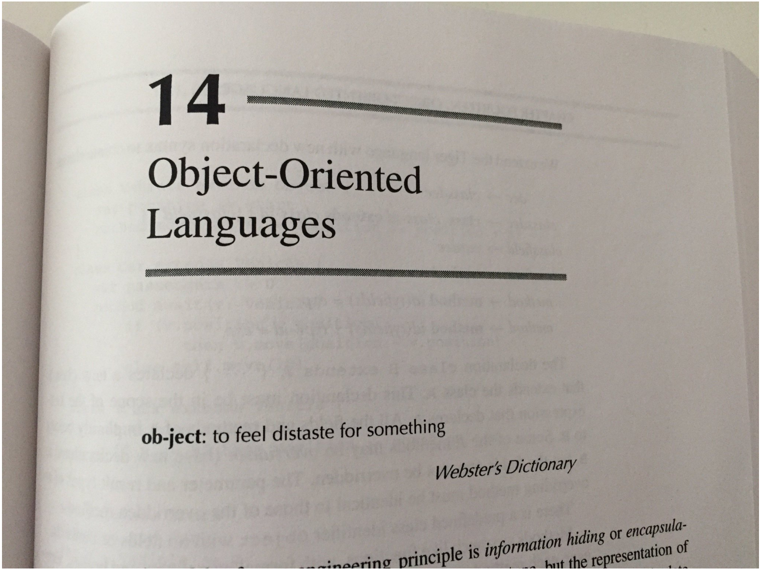
There are a number of trade-offs with this decision:

- Modularity has been compromised but,
- Types like `Data.Set` can exploit this coherence to enforce invariants.

Static Dispatch

Typically, the compiler can *inline* all dictionaries to their usage sites, thus eliminating all run-time cost for using type classes. This is only not possible if the exact type being used cannot be determined at compile-time, such as with polymorphic recursion etc.

Subtyping



14
Object-Oriented Languages
ob-ject: to feel distaste for something
Webster's Dictionary

Subtyping

To add subtyping to a language, we define a *partial order*[2] on types $\tau \leq \rho$ and a *rule of subsumption*:

$$\frac{\Gamma \vdash e : \tau \qquad \tau \leq \rho}{\Gamma \vdash e : \rho}$$

Type inference with subtyping is undecidable in general. Therefore, subsumptions (called *upcasts*) are sometimes made explicit (e.g. in OCaml):

$$\frac{\Gamma \vdash e : \tau \qquad \tau \leq \rho}{\Gamma \vdash \texttt{upcast}\ \rho\ e : \rho}$$

[2] Remember discrete maths, or check the glossary.

What is Subtyping?

What this partial order $\tau \leq \rho$ actually means is up to the language. There are two main approaches:

- **Most common**: where upcasts do not have dynamic behaviour, i.e. `upcast` $v \mapsto v$. This requires that any value of type τ could also be judged to have type ρ. If types are viewed as sets, this could be viewed as a subset relation.
- **Uncommon**: where upcasts cause a *coercion* to occur, actually converting the value from τ to ρ at runtime.

Observation: By using an identity function as a coercion, the coercion view is more general.

Desirable Properties

The coercion approach is the most general, but we might have some confusing results.

Example

Suppose `Int` $\leq$ `Float`, `Float` $\leq$ `String` and `Int` $\leq$ `String`.
There are now two ways to coerce an `Int` to a `String`:

1. Directly: `"3"`
2. via `Float`: `"3.0"`

Typically, we would enforce that the subtype coercions are coherent, such that no matter which coercion is chosen, the same result is produced.

Behavioural Subtyping

Another constraint is that the syntactic notion of subtyping should correspond to something semantically. In other words, if we know $\tau \leq \rho$, then it should be reasonable to replace any value of type ρ with an value of type τ without any observable difference.

Liskov Substitution Principle

Let $\varphi(x)$ be a property provable about objects x of type ρ. Then $\varphi(y)$ should be true for objects y of type τ where $\tau \leq \rho$.

Languages such as Java and C++, which allow for user-defined subtyping relationships (*inheritance*), put the onus on the user to ensure this condition is met.

Product Types

Assuming a basic rule $\texttt{Int} \leq \texttt{Float}$, how do we define subtyping for our compound data types?
What is the relationship between these types?

- $(\texttt{Int} \times \texttt{Int})$
- $(\texttt{Float} \times \texttt{Float})$
- $(\texttt{Float} \times \texttt{Int})$
- $(\texttt{Int} \times \texttt{Float})$

$$\frac{\tau_1 \leq \rho_1 \qquad \tau_2 \leq \rho_2}{(\tau_1 \times \tau_2) \leq (\rho_1 \times \rho_2)}$$

Sum Types

What is the relationship between these types?

- $(\texttt{Int} + \texttt{Int})$
- $(\texttt{Float} + \texttt{Float})$
- $(\texttt{Float} + \texttt{Int})$
- $(\texttt{Int} + \texttt{Float})$

$$\frac{\tau_1 \leq \rho_1 \qquad \tau_2 \leq \rho_2}{(\tau_1 + \tau_2) \leq (\rho_1 + \rho_2)}$$

Any other compound types?

Functions

What is the relationship between these types?

- $(\texttt{Int} \to \texttt{Int})$
- $(\texttt{Float} \to \texttt{Float})$
- $(\texttt{Float} \to \texttt{Int})$
- $(\texttt{Int} \to \texttt{Float})$

The relation is flipped on the left hand side!

$$\frac{\rho_1 \le \tau_1 \qquad \tau_2 \le \rho_2}{(\tau_1 \to \tau_2) \le (\rho_1 \to \rho_2)}$$

Variance

The way a *type constructor* (such as $+$, $\times$, `Maybe` or $\rightarrow$) interacts with subtyping is called its variance. For a type constructor C, and $\tau \leq \rho$:

- If $C\ \tau \leq C\ \rho$, then C is *covariant*.
  Examples: Products (both arguments), Sums (both arguments), Function return type, ...
- If $C\ \rho \leq C\ \tau$, then C is *contravariant*.
  Examples: Function argument type, ...
- If it is neither covariant nor contravariant then it is (confusingly) called *invariant*.
  Examples: **data** `Endo` $a = \mathsf{E}\ (a \rightarrow a)$

Stuffing it up

Many languages have famously stuffed this up, at the expense of type safety.

19 Types

Dart supports optional typing based on interface types.

The type system is unsound, due to the covariance of generic types. This is a deliberate choice (and undoubtedly controversial). Experience has shown that sound type rules for generics fly in the face of programmer intuition. It is easy for tools to provide a sound type analysis if they choose, which may be useful for tasks like refactoring.

A few years later...

Language and libraries

- Dart's type system is now sound.
  - Fixing common type problems

Java too

Java (and its Seattle-based cousin, C$^\sharp$) also broke type safety with incorrect variance in arrays.

We will demonstrate how this violates preservation, time permitting.

(Java redeemed itself by introducing invariant collections along with parametric polymorphism in 2004. These were believed sound until 2016.)