Overloading and Sub

Christine Rizkallah
CSE, UNSW
Term 3 2020

Motivation

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

Similarly, a numeric literal should take on whatever type is infer

$$(5 :: \texttt{Int})\ \mathrm{mo}$$

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

Without Overloading

We effectively have two functions:

$\texttt{Float}$ $\texttt{Float} \rightarrow \texttt{Float}$ $\texttt{Float}$

We would like to refer to both of these functions by the same name a 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 Class
- 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 (Num

In Haskell, the type `Num`, which has method

Example (Equality)

In Haskell, the `Eq` type class contains methods (e 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

Typically, P is a li

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:

Our typing judgement $\Gamma \vdash e : \pi$ now includes a set of ty

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

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

Typing Rules

The existing rules now just thread $\mathcal{A}$ through.
In order to use an overloaded type one must first show that the predicate is satisfied by the known axioms:

Right now, $\mathcal{A}$

If, adding a predicate to the known axioms, we can conclude a typi
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. $(+) :: \forall a.$
2. $\texttt{Num Float}$
3. Using $\textsc{Al}$
   $(+) :: (\texttt{Num Float}) \Rightarrow \texttt{Float} \rightarrow \texttt{Float} \rightarrow \texttt{Float}$
4. Using $\textsc{Inst}$ (on previous slide) and ❷, we can c
   $(+) :: \texttt{Float} \rightarrow \texttt{Float} \rightarrow \texttt{Float}$
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 co

$$(/=) : a \to a$$

becomes

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

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:


```
instance Eq Bool where
  True  ==  True  =  True
  _         _
```


becomes

$$\texttt{True} \ ==_{\texttt{Bool}} \ \texttt{True}$$
$$\texttt{False} \ ==_{\texttt{Bool}} \ \texttt{Fals}$$
$$\_ \ ==_{\texttt{Bool}} \ \_ \quad = \quad \texttt{False}$$
$$a \ /{=}_{\texttt{Bool}} \ b \quad = \quad \text{not}\ (a ==_{\texttt{Bool}} b)$$

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

Dictionaries and Resolution

Programs that rely on overloading now take dictionaries as parameters:

$$same :: \ a.\ (\mathtt{Eq}\ a)\ \ [a]\ \ \mathtt{Bool}$$

Becomes:

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

Generative Instances

We can make instances also predicated on some constraints:

$$\textbf{instance}\ (\texttt{Eq}\ a) \Rightarrow (\texttt{Eq}\ [a])\ \textbf{where}$$

Such instances are transformed into functions:

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

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 alo
- Defined alo

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 co ating
all run-time cost fo
This is only not possi
compile-time, such as with polymorphic recursion etc.

Subtyping

Subtyping

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

To aid in type infere
sometimes subsumptions (called *upcasts*) are made e

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

[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 com**
  `upcast` *v* ... judged to have type
- **Uncommon**: where upcasts cause a *coerci* ... g the value from τ to ρ at runtime.

Observation: By using an identity function as a coercion, the c ... general.

Desirable Properties

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

Example

Suppose `Int` ow two ways to coerce an `I`

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

Typically, we would enforce that the subtype coercions are co

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 τ ρ, then it should be reasonable to replace any valu rvable difference.

Liskov Substit

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

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

Product Types

Assuming a basic rule Int ≤ Float, how do we define subtyping for our compound data types?

What is the relatio

- (Int × In
- (Float ×
- (Float × Int)
- (Int × Float)

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

Sum Types

What is the relationship between these types?

- (Int + In
- (Float +
- (Float +
- (Int + Float)

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

Any other compound types?

Functions

What is the relationship between these types?

- (Int → Int)
- (Float →
- (Float →
- (Int → F

The relation is flipped on the left hand side!

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

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 $C\ \tau \leq$
  Example turn 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 = E $(a \rightarrow a)$

Stuffing it up

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

A few years later. . .

Java too

Java (and its Seatt riance in arrays.

We will demonstrate how this violates preservation, time permitting.