Elaboration on Overloaded Functions

Marius Weidner

Chair of Programming Languages, University of Freiburg

Abstract. Most popular programming languages support function overloading. Prominent use cases include overloading of arithmetic operators for different types or showing a arbitrary value as a string. We study a minimal extension of the Hindley Milner system [2] that supports function overloading [1]. We also derive an alternative system with support for recursive instances, straight forward debruijn indices and give big step semantics.

1 Introduction

When we say "overloaded functions" we usually mean overloaded identifiers. If we invoke overloaded identifiers as functions we expect the type checker to choose the correct instance of multiple given instances for us based on the type of the arguments. In combination with polymorphism we can go one step further by allowing quantified type variables to be restricted on instances.

1.1 Examples of Overloading in Popular Languages

Python uses magic methods to support overloading of operators and standard library functions. A class can override the behavior of any of the predefined magic methods. Commonly used magic methods are for example `__init__(self)` to provide logic when an object is initialized and `__eq__(self, other)` to give custom equality logic for objects when using the `==` operator. In Python it is not possible to define custom magic methods or any other form of custom overloading.

Haskell makes use of type classes. Type classes define abstract polymorphic functions that can be overloaded for specific types. Therefore we can instantiate a type class for specific types by concretely defining the behavior for all functions required by the type class when the type variables are substituted for those specific types. A function can have type class constraints to force substituted types for type variables to be a member of some instances.

Rust has a language feature called traits. Similar to Haskell's type classes, a trait defines shared functionality in the form of abstract function definitions. Traits are then implemented for one or more types. Type variables can be annotated with a trait bound forcing a concrete type, when substituted for the type variable, to have implemented a specific trait. Similar to Python some traits are predefined to overload operators, but custom traits can be defined. In contrast to Haskell's type classes, traits can also act as a special kind of types using the `dyn` and `impl` keywords.

```
inst eq :: Nat -> Nat -> Bool
  eq zero    zero    = true
  eq (suc x) (suc y) = eq x y
  eq _       _       = false in
inst eq :: (eq :: a -> a -> Bool) => [a] -> [a] -> Bool
  eq nil         nil         = true
  eq (cons x xs) (cons y ys) = eq x y && eq xs ys
  eq _           _           = false in
let is_eq = eq [0] [0] in unit
```

Fig. 1. Overloading Example in Preudocode

1.2 Example

In Fig. 1 we begin by defining two instances for `eq`. From the explicit type annotation, we can see that the first instance takes two `Nat` and performs pattern matching to determine if the are equal. The second instance is for lists of any type that has an instance for `eq`. More precisely, the constraint `(eq :: a -> a -> Bool) => ..` expresses that we need to have an instance at hand for the type that is substituted for the type variable `a` when using the second instance of `eq`. Inside the second instance we can safely call `eq` on elements of the list and on sub lists, given the language supports recursive instances. While `eq [zero] [zero]` would type check, `eq [true] [true]` would fail to type check, because the constraint for `eq` of lists requires an instance of `eq` for `Bool -> Bool -> Bool`.

2 System O

System O is a minimal extension to the Hindley Milner system [2] by Odersky, Wadler and Wehr [1] and supports overloaded identifiers for functions. To determine the correct instance for a call to an overloaded function System O restricts instances for the same identifier to differ in the type of their first argument. It is therefore straight forward to formulate untyped semantics, since we can determine the type of the first argument uniquely by the value given.

2.1 Example

2.2 Type Inference Algorithm

2.3 Dictionary Passing Transformation to Hindley Milner

2.4 Record Extension

3 Extending System O

We extend System O by recursive instances and give big step semantics. The system is designed to be easily used with debruijn indices. First we extend syntax and type system, then we give big steps semantics and finally study the use of debruijn indices.

$$
\begin{array}{rrcl}
\text{Constructors} & k & \in & \mathcal{K} = \bigcup\{\mathcal{K}_D \mid D \in \mathcal{D}\} \\
\text{Unique Variables} & u & \in & \mathcal{U} \\
\text{Overloaded Variables} & o & \in & \mathcal{O} \\
\text{Variables} & x & := & u \mid o \mid k \\
\text{Expressions} & e & := & x \mid \lambda x.\ e \mid e\ e \mid \textbf{let}\ x = e\ \textbf{in}\ e \\
\text{Programs} & p & := & \textbf{decl}\ o\ \textbf{in}\ p \mid \textbf{inst}\ o : \sigma_T = e\ \textbf{in}\ p \\
 & & & \\
\text{Datatype constructors} & D & \in & \mathcal{D} \\
\text{Type constructors} & T & \in & \mathcal{T} = \mathcal{D} \cup \{\rightarrow\} \\
\text{Type variables} & \alpha & \in & \mathcal{A} \\
\text{Mono types} & \tau & := & \alpha \mid \tau \rightarrow \tau \mid D\ \tau_1\ ..\ \tau_n \\
\text{Poly types} & \sigma & := & \tau \mid \forall \alpha.\ \pi_\alpha \Rightarrow \sigma \\
\text{Instance types} & \sigma_T & := & T\ \alpha_1\ ..\ \alpha_n \rightarrow \tau \mid \forall \alpha.\ \pi_\alpha \Rightarrow \sigma_T \\
\text{Constraints} & \pi_\alpha & := & x_1 : \alpha \rightarrow \tau_1\ ..\ x_n : \alpha \rightarrow \tau_n \\
 & & & \\
\text{Instance Type Contexts} & \Sigma & := & \cdot \mid \Sigma \uplus \sigma_T \\
\text{Type Contexts} & \Gamma & := & \cdot \mid \Gamma,\ x : \sigma \mid \Gamma,\ o : \Sigma \mid \Gamma(o) \uplus \sigma_T \\
 & & & \\
\text{Values} & v & := & \lambda(\mathcal{E};\ x).\ e \mid k\ v_1\ ..\ v_n \mid \mathcal{S} \\
\text{Instance Eval Contexts} & \mathcal{S} & := & \cdot \mid \mathcal{S} \uplus (e,\ T) \\
\text{Evaluation Contexts} & \mathcal{E} & := & \cdot \mid \mathcal{E},\ x : v \mid \mathcal{E}(o) \uplus (e,\ \sigma_T)
\end{array}
$$

Fig. 2. Syntax

3.1 Syntax

We only discuss changes to the original System O syntax.

The **decl** statement declares an identifier o to be overloaded in p. Identifiers can only have instances, if declared as overloaded.

Typing context Γ can hold one or more types per identifier. Normal identifiers x have exactly one type σ while overloaded identifiers have a list of types Σ with length equal to the amount of instance definitions. We write $\Gamma(o) \uplus \sigma_T$ to append a type σ_T to the list of types Σ of identifier o.

A value v can be a closure $\lambda(\mathcal{E};\ x).\ e$, constructor k applied to values v_1 to v_n or a list $\mathcal{S}$ of type annotated expressions (e, T). The latter occurs when an overloaded identifier is treated as value.

The evaluation context $\mathcal{E}$ is analogous to the typing context Γ. $\mathcal{E}$ can hold exactly one value for normal identifiers x and multiple typed expressions for overloaded identifiers o. We write $\mathcal{E}(o) \uplus (e, \sigma_T)$ to append a type (e, σ_T) to the list of typed expressions $\mathcal{E}$ of identifier o.

3.2 Typing

$$\text{(T-Var)}\quad \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad\qquad \frac{o : \Sigma \in \Gamma \qquad \sigma_T \in \Sigma}{\Gamma \vdash o : \sigma_T}\quad \text{(T-OVar)}$$

$$\text{(T-Abs)}\quad \frac{\Gamma,\ x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.\ e : \tau \to \tau'} \qquad\qquad \frac{\Gamma \vdash e : \tau \to \tau' \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \tau'}\quad \text{(T-App)}$$

$$\text{(T-Gen)}\quad \frac{\Gamma,\ \pi_\alpha \vdash e : \sigma \qquad \text{fresh } \alpha}{\Gamma \vdash e : \forall \alpha. \pi_\alpha \Rightarrow\ \sigma} \qquad\qquad \frac{\Gamma \vdash e : \forall \alpha.\ \pi_\alpha \Rightarrow \sigma \qquad \Gamma \vdash [\tau/\alpha]\pi_\alpha}{\Gamma \vdash e : [\tau/\alpha]\sigma}\quad \text{(T-Inst)}$$

$$\text{(T-Set)}\quad \frac{\Gamma \vdash x_1 : \sigma_1 \quad ... \quad \Gamma \vdash x_n : \sigma_n}{\Gamma \vdash x_1 : \sigma_1 \quad ... \quad x_n : \sigma_n} \qquad\qquad \frac{\Gamma \vdash e' : \sigma \qquad \Gamma,\ x : \sigma \vdash e : \tau}{\Gamma \vdash \textbf{let}\ x = e'\ \textbf{in}\ e : \tau}\quad \text{(T-Let)}$$

$$\text{(T-Decl)}\quad \frac{\Gamma,\ o : \cdot \vdash p : \sigma \qquad \text{fresh } o}{\Gamma \vdash \textbf{decl}\ o\ \textbf{in}\ p : \sigma} \qquad \frac{\begin{array}{c}\Gamma \vdash o : \Sigma \qquad \forall \sigma_{T'} \in \Sigma \Rightarrow T \neq T' \\ \Gamma(o) \uplus \sigma_T \vdash e : \sigma_T \qquad \Gamma(o) \uplus \sigma_T \vdash p : \sigma\end{array}}{\Gamma \vdash \textbf{inst}\ o : \sigma_T = e\ \textbf{in}\ p : \sigma}\quad \text{(T-Inst)}$$

Fig. 3. Typing ($\Gamma \vdash p : \sigma$)

Again, we only discuss changes to the original type system.

Rule (T-OVar) says that an overloaded identifier o has type σ_T if it occurs in the list of function types Σ that the variable is overloaded with.

Rule (T-Decl) introduces an new overloaded variable o to p by appending Γ in p with the empty list, for future **inst**'s to append their explicit type.

Finally, (T-Inst) checks that for every σ'_T in Σ of o the constructor of the first argument T is unique. To support recursive instances we append the type annotation of the instance σ_T to Γ when checking the body e. We also can to assume that all constraints $\pi(\sigma_T)$ are met inside e.

3.3 Big Step Semantics

$$\text{(R-Var)}\quad \frac{x : v \in \mathcal{E}}{\mathcal{E} \vdash x \downarrow v} \qquad\qquad \frac{}{\mathcal{E} \vdash \lambda x.\ e \downarrow \lambda(\mathcal{E};\ x).\ e} \quad\text{(R-Abs)}$$

$$\text{(R-App)}\quad \frac{\mathcal{E} \vdash e_1 \downarrow \lambda(\mathcal{E}';\ x).\ e \qquad \mathcal{E} \vdash e_2 \downarrow v_2 \qquad \mathcal{E}',\ x : v_2 \vdash e \downarrow v}{\mathcal{E} \vdash e_1\ e_2 \downarrow v} \qquad \frac{\mathcal{E} \vdash e_1 \downarrow S \qquad \mathcal{E} \vdash e_2 \downarrow v_2 \qquad \exists (e',\ \sigma_T) \in S \Rightarrow v_2 \sqsubseteq T \qquad \mathcal{E} \vdash e' \downarrow \lambda(\mathcal{E}';\ x).\ e \qquad \mathcal{E}',\ x : v_2 \vdash e \downarrow v}{\mathcal{E} \vdash e_1\ e_2 \downarrow v} \quad\text{(R-IApp)}$$

$$\text{(R-Decl)}\quad \frac{\mathcal{E},\ o : \cdot \vdash p \downarrow v}{\mathcal{E} \vdash \textbf{decl}\ o\ \textbf{in}\ p \downarrow v} \qquad\qquad \frac{\mathcal{E}(o) \uplus (e,\ \sigma_T) \vdash p \downarrow v}{\mathcal{E} \vdash \textbf{inst}\ o : \sigma_T =\ e\ \textbf{in}\ p \downarrow v} \quad\text{(R-Inst)}$$

$$\text{(R-CApp)}\quad \frac{\mathcal{E} \vdash e_1 \downarrow v_1\ ..\ \mathcal{E} \vdash e_1 \downarrow v_1}{\mathcal{E} \vdash k\ e_1\ ..\ e_n \downarrow k\ v_1\ ..\ v_n} \qquad\qquad \frac{\mathcal{E} \vdash e' \downarrow v' \qquad \mathcal{E},\ x : v' \vdash e \downarrow v}{\mathcal{E} \vdash \textbf{let}\ x = e'\ \textbf{in}\ e \downarrow v} \quad\text{(R-Let)}$$

where $v \sqsubseteq T$:

$$\text{(C-Abs)}\ \frac{}{\lambda(\mathcal{E};\ x).\ e \sqsubseteq \rightarrow} \quad \text{(C-Cstr)}\ \frac{k \in \mathcal{K}_D}{k\ v_1\ ..\ v_n \sqsubseteq D} \quad \text{(C-Inst)}\ \frac{}{S \sqsubseteq \rightarrow}$$

Fig. 4. Big Step Semantics ($\mathcal{E} \vdash p \downarrow v$)

Rules (R-Var), (R-App), (R-Abs), (R-Let) are standard.

(R-CApp) evaluates n-ary predefined constructors, threating k as a function applied to n arguments.

Analogous to (T-Decl), (R-Decl) adds the overloaded identifier to the evaluation context with zero instances and evaluates the continuation.

When an expression e_1, that evaluates to a list of type annotated expressions S, is applied to some e_2, the (R-IApp) rule is invoked. If there exists an instance $(e',\ \sigma_T) \in S$ which's type T matches the type of the argument e_2, we take e' and apply e_2 to it.

The binary relation $v \sqsubseteq T$ relates constructor values to their corresponding type and is used inside (R-IApp).

3.4 Debruijn Indices

In contrast to the original paper our system has the advantage of having only exactly one entry in environments per overloaded identifier. Instead of a new entry for each instance declaration we extend the list of types for each overloaded identifier in Γ and list of expressions in $mathcalE$ respectively. The reason for introducing the `decl` expression to the language is to have exactly one specific expression to define the new variable, all instance definitions then refer to this one variable.

4 Conclusion

We have studied System O, a minimal system that is foundation of many popular programming languages features like type classes and traits. Because of the close relation to Hindley Milner's system, full type inference is preserved by a simple extension of Algorithm W. System O itself is foundation to many more advanced systems and can be extend by a lot of interesting features. Exemplary, we studied the extension of System O by recursive instance declarations.

References

1. Odersky, M., Wadler, P. & Wehr, M. A Second Look at Overloading. *Proceedings Of The Seventh International Conference On Functional Programming Languages And Computer Architecture*. pp. 135-146 (1995), https://doi.org/10.1145/224164.224195
2. Milner, R. A theory of type polymorphism in programming. *Journal Of Computer And System Sciences*. **17**, 348-375 (1978), https://www.sciencedirect.com/science/article/pii/0022000078900144