# Elaboration on Overloaded Functions

Marius Weidner

Chair of Programming Languages, University of Freiburg

**Abstract.** Most popular programming lanuages support function overloading. Prominent use cases include operator overloading or showing a arbitrary value as a string. We study a minimal extension of the Hindley Milner system that supports function overloading while preserving full type inference [1]. We also derive an alternative system with support for recursive implementations, straight forward debruijn indices and give big step semantics.

## 1 Introduction

When we talk about shared or overloaded functions we usually mean overloaded identifiers. If we invoke overloaded identifiers as functions we expect the language to choose the correct implementation of multiple given implementations for us based on the type of the arguments.

### 1.1 Example

In Fig. 1 we give an example in pseudocode with only three top level language constructs. Using `data` definitions we can define inductive data types. A `delc` statement defines an overloaded identifier. With similar syntax to let statements, `impl` statements provide a concrete implementation for calls to the overloaded identifier for some explicit type.

The code example first defines 3 well known data types `Bool`, `Nat` and Lists of type `a` (`[a]`). Type variables are implicitly quantified. Next we declare that `eq` should be an overloaded identifier. We follow with two implementations for `eq`. From the explicit type annotation we can deduce that the first implementation takes two `Nat` and performs pattern matching to determine if the are equal. The second implementation is for lists of any type that has an implementation for `eq` itself. The constraint `(eq :: a -> a -> Bool) => ..` expresses that we need to have an implementation at hand for the type that is substituted for the type variable `a` when using the second implementation of `eq`. Inside the second implementation we can safely call `eq` on elements of the list (because of the constraint) and on sublists (given the language supports recursive implementations). The call `eq [zero] [zero]` would type check while `eq [true] [true]` would fail to type check because the constraint for `eq` of lists requires an implementation of `eq` for `Bool -> Bool -> Bool`.

### 1.2 Examples of Overloading in Popular Languages

*Python* uses magic methods to support overloading of operators and standard library functions. A class can override the behaviour of any of the predefined magic methods.

```
data Bool
  true  : Bool
  false : Bool
data Nat
  zero : Nat
  suc  : Nat -> Nat
data List a
  nil  : List a
  cons : a -> List a -> List a

decl eq

impl eq :: Nat -> Nat -> Bool
  eq zero    zero    = true
  eq (suc x) (suc y) = eq x y
  eq _       _       = false
impl eq :: (eq :: a -> a -> Bool) => List a -> List a -> Bool
  eq nil         nil         = true
  eq (cons x xs) (cons y ys) = eq x y && eq xs ys
  eq _           _           = false
```

**Fig. 1.** Overloading Example in Preudocode

Commonly used magic methods are for example `__init__(self)` to provide logic when an object of the class is initialized and `__eq__(self, other)`to give custom equality logic for objects of the class 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 abstact functions that can be overloaded. A type can be an instance of a type class by concretely defining the behaviour for all functions required by the type class when called on that type. Type variables can have constraints to be an instance of one or more type classes. The type checker then searches for a suitable instance by instance resultion when given a concrete type for the type variable.

*Rust* has a language feature called traits. Similar to Haskells type classes, a trait defines shared functionality in the form of abstract function definitions that can be implemented by 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. In constrast to Haskells type classes, traits can also act as a special kind of types using the `dyn` and `impl` keywords.

## 2 System O

System O is a minimal system similar to the Hindley Milner system [2] for overloaded functions by Odersky, Wadler and Wehr [1] with similarities to the pseudo code example above.

### 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 discuss the following three topics not covered by Odersky, Wadler and Wehr in their original paper [1]:

- Recursive Implementations
- Big Step Semantics
- Debruijn Indices

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

**Fig. 2.** Syntax

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

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

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

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

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

where $\pi(\forall \alpha_1.\pi_1.\ \ .. \ \forall \alpha_n.\pi_n.\ \tau) = \pi_1,\ ..,\ \pi_n$

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

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

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

$$\text{(R-Decl)}\quad \frac{\mathcal{E},\ x : \cdot \vdash p \downarrow v}{\mathcal{E} \vdash \text{decl } x \text{ in } p \downarrow v} \qquad \frac{\mathcal{E}(x) \uplus (e,\ T) \vdash p \downarrow v}{\mathcal{E} \vdash \text{impl } x : \sigma_T = \ e \text{ in } p \downarrow v}\quad \text{(R-Impl)}$$

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

where $v \sqsubseteq T$ :

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

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

### 3.1 Debruijn Indices

### 3.2 Recursive Implementations

## 4 Conclusion

## 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