Elaboration on Overloading

Marius Weidner

Chair of Programming Languages, University of Freiburg

Abstract. Most real world programming languages support 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 that supports overloading, while preserving full type inference. We also derive an alternative system with support for recursive instances, straight forward debruijn indices and give big step semantics.

1 Introduction

The term overloading refers to overloaded *identifiers*. We overload the meaning of an identifier o with multiple valid interpretations, in the form of functions, called *instances*. Each instance gives one specific meaning to the identifier. Each instance is unique in the type of the first argument T. We say identifier o is defined on T, if there exists an instance where T is the type of the first argument of that function. Hence, if a instance is invoked as a function we can decide the instance invoked based on the type of the first value applied to the identifier. In combination with polymorphism we can go one step further by permitting quantified type variables to be restricted. A type variable can be restricted to be only substitutable by a type T for which instances o_1, .., o_n are defined on T. Real world programming languages support overloading in a variety of ways, but they can usually be reduced to a more minimal system with only overloaded function identifiers.

1.1 Example

In Fig. 1 we define two instances for overloaded identifier `eq`. Using the *explicit* type annotation, we can reason that the first instance takes two `Nat` and performs pattern matching to determine if the are equal. The second instance is for lists with elements of type α. However, α is constrained to only be substituted by types T for which `eq` is defined on T. More precisely, the constraint `(eq :: a -> a -> Bool) => ..` expresses that we need to have an instance at hand for the type T, that is substituted for the type variable α, when using the second instance of `eq`. Inside the second instance we can safely call `eq` on elements of the list, because of the constraint, 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 is not satisfied.

```
inst eq :: Nat -> Nat -> Bool
  eq zero    zero    = true
  eq (suc x) (suc y) = eq x y
  eq _       _       = false in
inst eq :: (eq :: α -> α -> Bool) => [α] -> [α] -> 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

1.2 Overloading in Popular Languages

We briefly explore three examples of overloading in popular programming languages. Despite the fact that all the examples are high level language constructs, we could theoretically translate them to some minimal language similar to the one in in Fig. 1.

Python uses magic methods to support overloading of operators and standard library functions. A class can override the behavior for 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. Therefore we can instantiate a type class for a specific type by concretely defining the behavior for all functions defined by the type class concretely. 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 abstract function definitions. Traits are implemented for types. This reminds of Python's magic methods, since they are defined on classes. 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. Analogous 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.

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. Since the Hindley Milner system is widely known, we only discuss the extensions System O adds to the Hindley Milner System.

2.1 Syntax

Hindley Milner's syntax is extended by instance declarations and constraints on type variables.

Instance declarations **inst** $o : \sigma_T = e$ **in** p declare a new instance for overloaded identifier o inside the rest of the program p. The explicit type annotation σ_T must be a, possibly polymorphic, *function* type where T denotes the type of the first argument.

Polymorphic types σ_T can introduce type variables α using the $\forall$ quantifier. On top of the original Hindley Milner quantifier $\forall\alpha.\ \sigma$, System O allows to introduce constraints π_α on α. The list of constraints π_α contains constraints in the form of $o : \alpha \to \tau$, where o is some overloaded identifier and τ is a monomorphic type. We write $\forall\alpha.\ (o_1 : \alpha \to \tau_1, o_n : \alpha \to \tau_n) \Rightarrow \sigma$, to denote polymorphic types with constraints and $\forall\alpha.\ \sigma$, if there are no constraints on α.

2.2 Type System

System O's type system a simple an extension of Hindley Milner's typing rules [3].

The original (GEN) rule is renamed to ($\forall$-I) and extended by passing introduced constraints π_α from type scheme to context Γ. Hence constraints π_α are presumed to be valid inside the expression e that the $\forall$ was introduced to.

To eliminate the presumed constraints inside the ($\forall$-E) rule, we not only eliminate the $\forall$ by substituting a concrete type τ for the bound type variable α, but also substitute τ for α inside constraints π_α and check if they follow form the environment Γ.

Since π_α is a list of constraints $o_i : \alpha \to \tau_i$, $i \in \mathbb{N}$, an additional rule (SET) is necessary. The rule says that constraints π_α are valid, if each single constraint $o : \alpha \to \tau$ is valid.

Finally, there is an additional rule (INST) for instance declarations. Since `inst` declaration are similar to `let` statements their typing is also similar. Other than with `let`, we have an explicit type annotation σ_T at hand. We also require all other instances with the same name o, defined before in Γ with type annotations $\sigma_{T'}$, to differ in the type of the first argument T'. This requirement is necessary to *deterministically* find the correct instance when an overloaded identifier is invoked.

2.3 Semantics

Since all instances differ in the type of their first argument, it is straight forward to formulate *untyped* semantics. When an overloaded identifier is invoked $o\ v_1\ ..\ v_n$, we can determine the type of the first argument T uniquely by the value v_1. The authors give denotational semantics. Denotational semantics use a meaning function $[\![\cdot]\!]_\eta$ that gives a term or type $\cdot$ a mathematical meaning in environment η. In section 3.3 we give operational semantics for a slightly modified System O.

2.4 Type Inference Algorithm

The authors extended Hindley Milner's Algorithm W [2]. We only discuss changes to the original Algorithm W.

The algorithm is extended by a case for `inst` declarations. The *inferred* type σ'_T of the body e must be *less* general than the type annotation σ_T. Formally, a type $\forall\beta_1\ ..\ \beta_n.\ \tau'$ is less general than $\forall\alpha_1\ ..\ \alpha_n.\ \tau$, if there exists a substitution $\{\alpha_i \mapsto \tau_i\}$,

such that $\{\alpha_i \mapsto \tau_i\}\tau = \tau'$. For example, `Nat` $\to$ `Nat` is less general than $\forall\alpha.\ \alpha \to \alpha$ for substitution $\{\alpha \mapsto \text{Nat}\}$. This extension corresponds to the (INST) rule.

The algorithm uses unification to produce a most general substitution. When binding a variable α to τ while unifying, all constraints π_α on α in Γ are required to follow from Γ, when substituting τ in π_α for α. This extension embeds the ($\forall$-E) rule.

2.5 Dictionary Passing Transformation to Hindley Milner

```
inst eq : Nat -> Nat -> Bool
  = λ.. in
inst eq : ∀α. (eq : α -> α -> Bool) => [α] -> [α] -> Bool
  = λ.. in
eq [0] [0]
---------------------- translates to ----------------------
let eq₀ :: Nat -> Nat -> Bool
  = λ.. in
let eq₁ :: ∀α. (α -> α -> Bool) -> [α] -> [α] -> Bool
  = λeq₀. λ.. in
eq₁ eq₀ [0] [0]
```

Fig. 2. Dictionary Passing Transform

The dictionary passing transform translates *typed* System O programs to typeable Hindley Milner programs. The transformation uses type information to translate `inst` declarations to `let` bindings and constraints π_α to higher order functions.

When substituting `inst` declaration with `let` bindings we omit the type annotation and introduce a new unique name for that instance. We use type information to replace later invocations on that specific instance with the new unique name.

Since constraints π_α can only appear in the explicit type of instances, they are transformed to *arguments* of the translated instance. For each constraint $o : \alpha \to \tau$ we add a new argument of type $\alpha \to \tau$. When invoking the unique name of the translated instance we pass the correct instances originally required by the constraints as arguments. Using the type information this is straight forward. While translating constraints to higher order functions it can happen that a type scheme is part of an function type. Since constraints π_α only use mono types τ and type variable α it is safe to define that $\tau \to \forall\alpha.\ \sigma \equiv \forall\alpha.\ \tau \to \sigma$ inside that process. Applying the equivalence rule until it cannot be used anymore, results results in typeable Hindley Milner programs.

An example translation can be found in Fig. 2.

2.6 Relationship with Record Typing

Extending System O with records and subtyping on records is straight forward. More surprisingly this extended language can be translated *back* to System O without the record extension.

```
let max :: ∀β. (gte : β -> β -> Bool)
            => ∀α. (α ≤ {key: β}) => α -> α -> α
  = λx. λy. if gte x.key y.key then x else y in
max {field: "a", key: 1} {field: "b", key: 2}
------------------ translates to ------------------
inst field : ∀α. ∀β. R₀ α β -> α = λR₀ x y. x in
inst key : ∀α. ∀β. R₀ α β -> β = λR₀ x y. y in
let max :: ∀β. (gte : β -> β -> Bool) =>
           ∀α. (key : α -> β) => α -> α -> α
  = λx. λy. if gte (key x) (key y) then x else y in
max (R₀ "a" 1) (R₀ "b" 2)
```

Fig. 3. Translating System O + Records + Subtyping to System O [1]

The record extension introduces record types $\{l_i : \tau_i\}_{i\in\mathbb{N}}$, record expressions $\{f_i = e_i\}_{i\in\mathbb{N}}$, selectors $e.f$ selecting field f on record e and subtyping constraints $a \leq b$ where a must have at least all the fields $l_i : \tau_i$ that b has.

The translation introduces data types $R_i\ \alpha_1\ ..\ \alpha_n$ for each unique record $r_i = \{f_1 : \alpha_1,\ ..\ f_n : \alpha_n\}$ given in the original program. For each field f_i an instance `inst f`$_i$ `: ∀`$\alpha_1\ ..\ \alpha_n$`. R`$_i$ $\alpha_1\ ..\ \alpha_n$ `-> `α_i `= λR`$_i$ `x`$_1$ `.. x`$_n$`. x`$_i$ `in ..` is declared. All selector expressions `e.f` are translated to instance invocations `f e`.

Subtyping constraints $\forall\alpha.\ \alpha \leq \{\mathtt{f}_1 : \beta_1,\ ..\ , \mathtt{f}_n : \beta_n\} \Rightarrow \tau$ are translated to constraints $\forall\alpha.(\mathtt{f}_1 : \alpha \to \beta_1,\ ..\ , \mathtt{f}_n : \alpha \to \beta_n) \Rightarrow \tau$.

The original paper gave a vivid example. A slightly modified version can be found in Fig. 3.

3 Extending System O

We extend System O by recursive instances and give big step semantics. The system is designed to straight forwardly use debruijn representation.

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.

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

Fig. 4. Syntax

3.2 Typing

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

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

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

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

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

Fig. 5. 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.

3.3 Big Step Semantics

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 in rule (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

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

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

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

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

where $v \sqsubseteq T$:

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

Fig. 6. Big Step Semantics ($\mathcal{E} \vdash p \downarrow v$)

instance declaration we extend the list of types for each overloaded identifier in Γ and list of expressions in $\mathcal{E}$ 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. With these changes transforming a given program to debruijn representation is straight forward.

4 Conclusion

We have studied System O, a minimal system that is foundation of many popular programming languages features like Haskell's type classes and Rust's traits. Because of the close relation to Hindley Milner's system, full type inference is preserved by a simple extension of Algorithm W. We extended System O by recursive instance declarations and gave operational semantics.

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
3. Damas, L. & Milner, R. Principal Type-Schemes for Functional Programs. *Proceedings Of The 9th ACM SIGPLAN-SIGACT Symposium On Principles Of Programming Languages*. pp. 207-212 (1982), https://doi.org/10.1145/582153.582176