Overloading, searching for alternatives

Nicolas Pouillard

`Nicolas.Pouillard@inria.fr`

June 6, 2007

Outline

Introduction

Overloading or not?

- Crucial design choice
- Hard to avoid for comparison, arithmetic and printing

Some languages already have overloading

- **C++**, well used but too complex and not formalized
- **Haskell**, less used but well understood

While working for `Intel`...

At `Intel` they develop their own language, *reFLect*

- Functional and lazy
- Circuit modeling and BDDs are deeply integrated
- A fully reflective language
- An advanced overloading system
- An interpreter for that language

At `INRIA`, we develop for them a compiler

- Based on the **OCaml** environment
- Sharing the back-end compilers (starting at lambda-code)
- Lifting the design a bit

The *reFLect* overloading system

- Has evolved from version to version

The *reFLect* overloading system

- Has evolved from version to version
- Firstly, features ”closed overloading”

The *reFLect* overloading system

- Has evolved from version to version
- Firstly, features ”closed overloading”
- Provides direct and delegated forms

The *reFLect* overloading system

- Has evolved from version to version
- Firstly, features ”closed overloading”
- Provides direct and delegated forms
- Then provides ”open overloading”

The *reFLect* overloading system

- Has evolved from version to version
- Firstly, features ”closed overloading”
- Provides direct and delegated forms
- Then provides ”open overloading”
- Finally, recursion trough ”open overloading”

Today's overloading presentation

Extends the ML we know

Closer to **OCaml** than *reFLect* : Call-by-value, modules, signatures...

Takes another path, not chronological

Starting from "open overloading" and then close it

Syntax adopted, closer to OCaml but:

- Operators are escaped like _+_ instead of `( + )`
- `Int.t`, `List.t`... instead of `int`, `list`...
- Types and constructors application are like functions: `List.t Int.t`, `Cons 1 Nil`...

Outline

Declaring overloaded symbols

One declares it by giving a name and a type:

```
overload print : IO.output → α → unit
overload _+_ : α → α → α
```

The given type is of any form

- Can be a value (no arrow)
- As many type variables as you want
- Type variables are implicitly universally quantified
- This type scheme is called the anti-unifier

Declaring instances

One gives the overloaded name and a list of new alternatives.

```
instance print Int.print Float.print
instance _+_ Int._+_ Float._+_
```

Each alternative must be

- Unifiable with the anti-unifier
- Non-unifiable with all other alternatives

Overloading and type inference

Type inference in this system is a two step process:

Overloading and type inference

Type inference in this system is a two step process:

Traditional Hindley-Milner type inference

- Fast, linear (in practice) in the size of the abstract syntax tree
- Starts with the anti-unifier for occurrences of overloaded symbols
- Gathers constraints on the type of occurrences

Overloading and type inference

Type inference in this system is a two step process:

Traditional Hindley-Milner type inference

- Fast, linear (in practice) in the size of the abstract syntax tree
- Starts with the anti-unifier for occurrences of overloaded symbols
- Gathers constraints on the type of occurrences

Overload resolution

- Searches for a unique most-general satisfying assignment to the type variables in those constraints
- With all the complexity that entails

The outcome of overloading (basic)

Candidates:

- A candidate is valid when it is unifiable with the constrained occurrence
- Finally on each occurrence of an overloaded symbol, we have a list of valid candidates

The outcome of overloading (basic)

Candidates:

- A candidate is valid when it is unifiable with the constrained occurrence
- Finally on each occurrence of an overloaded symbol, we have a list of valid candidates

The simplified outcome:

- When only one candidate: keep it
- Else: raise an overloading error

Simple examples

```
# let isucc x = 1 + x
- val isucc : Int.t → Int.t

# let fsucc x = 1.0 + x
- val fsucc : Float.t → Float.t

# let foo x y = if x = y then x + 12 else y + y
- val foo : Int.t → Int.t → Int.t
```

Simple errors

```
# false + true
! Unresolved symbol: _+_ : Bool.t → Bool.t → Bool.t
! The 2 possible alternatives are:
!   Int._+_ : Int.t → Int.t → Int.t
!   Float._+_ : Float.t → Float.t → Float.t

# fun x → x + x
! Unresolved symbol: _+_ : α → α → α
! The 2 possible alternatives are:
!   Int._+_ : Int.t → Int.t → Int.t
!   Float._+_ : Float.t → Float.t → Float.t
```

A more complex example

```
# let id_int x = x + 0
- val id_int : Int.t → Int.t
# let id_bool x = x && true
- val id_bool : Bool.t → Bool.t
# let id_float x = x + 0.0
- val id_float : Float.t → Float.t
# overload id1 : α → α
# instance id1 id_int id_bool
# overload id2 : α → α
# instance id2 id_float id_int
# fun x → id1 (id2 x)
```

A more complex example

```
# let id_int x = x + 0
- val id_int : Int.t → Int.t
# let id_bool x = x && true
- val id_bool : Bool.t → Bool.t
# let id_float x = x + 0.0
- val id_float : Float.t → Float.t
# overload id1 : α → α
# instance id1 id_int id_bool
# overload id2 : α → α
# instance id2 id_float id_int
# fun x → id1 (id2 x)
- <fun> : Int.t → Int.t
```

Why alternatives must be pairwise not unifiable?

```
# let inc_fst p = fst p + 1
- val inc_fst : (Int.t * α) → Int.t
# let inc_snd p = snd p + 1
- val inc_snd : (α * Int.t) → Int.t
# overload inc_one : (α * β) → Int.t
# instance inc_one inc_fst inc_snd
! Attempt to overload "inc_one" with alternatives...
!   inc_fst : (Int.t * α) → Int.t
!   inc_snd :: (β * Int.t) → Int.t
```

Indeed `inc_one(42,true)` and `inc_one("foo",64)` make sense, but what about `inc_one(16,64)`?

Overloading and modularity

- Better distinction since we must give to the candidate a name
- And then, add the candidates to an overloaded symbol
- Modules like `Int`, `Float` are concrete
- A module like `Num` can declare overloaded symbols
- Then the user can choose and is not a prisoner of overloading

Outline

Delegated overloading

One wants to compile such a generic definition

```
let double x = x + x
```

The solution adopted

An overloading error (too much candidates) becomes an implicit argument when the overloading can be resolved later

In a nutshell

While classic overloading is just some type directed sugar, delegated overloading allows generic programming

The outcome of overloading (complete)

Context

- Are we defining a value? (called **def**)
- Is this choice "future proof"? (called **future_proof**)

Outcome

```
> if no candidates and future_proof then
>   raise an overloading error
> else if only one candidate
>     and (future_proof or not def) then keep it
> else if def then abstract over its implementation
> else raise an overloading error
```

How delegated overloading works

Principle

When an overloaded symbol cannot be resolved but could be in the future, we abstract the current definition from the implementation of unresolved symbols.
These implicit arguments are then re-introduced at each call site as overloaded occurrences.

Note

In this system, the resolution strategy is fixed

Delegated overloading in action

```
# let double x = x + x
- val double : [_+_ : α → α → α] ⇒ α → α
```

In fact the definition of `double` implicitly becomes:

```
let double' _+_ x = x + x
```

And a call to the function `double` is treated like that:

```
double 42
⟶ double' (_+_ : α → α → α) 42
⟶ double' (_+_ : Int.t → Int.t → Int.t) 42
⟶ double' Int._+_ 42
```

Implicit arguments

For each occurrence

Given an overloaded symbol `f`:

```
f : [a1 : t1, ..., aN : tN] ⇒ t
```

Add all implicit arguments

```
f x ⟶ (f' : t1 → ... → tN → t) a1 ... aN x
```

Implicit arguments are:

- lexically ordered
- distinct by name and type (others are merged)

Closing the overloading

Prevent a symbol from being extended:

- Is useful from development policy point of view
- Can help the typing algorithm to pick a candidate

Syntax of closing

```
close_overload _+_
```

`future_proof` extension

We extend the `future_proof` predicate to return true when the overloaded symbol is closed

Guessing the anti-unifier

One can provide short-cut for "closed overloading", by just taking candidates and computing the least general anti-unifier

```
t ::= X | F T1...TN

t          ⊗ t          = t
F t1...tN  ⊗ F u1...uN  = F (t1 ⊗ u1)...(tN ⊗ uN)
t          ⊗ u          = fresh_var t u
```

Outline

Recursive functions

A compilation choice

- Choose for once the set of implicit arguments
- Pass different implicit arguments trough recursive calls

The second choice will require an annotation, and will be useful only with polymorphic recursion (which also often requires an annotation).

Example

```
# let rec print_list = ... print ... print_list ...
- val print_list :
-  [print : IO.output → α → unit] ⇒
-    IO.output → List.t α → unit
```

Recursion Through Open Overloading

Can a candidate use its own overloaded symbol?

Seems evident that that printing a list also rely on printing an element, that adding pairs rely on adding it's elements

Example

```
# overload size : α → Int.t
# let int_size (_:Int.t) = 1
- val int_size : Int.t → Int.t
# let pair_size (x, y) = size x + size y
- val pair_size :
-   [size : α → Int.t, size : β → Int.t]
-   ⇒ (α * β) → Int.t
# instance size int_size pair_size
```

Recursion Through Open Overloading

Can a candidate use its own overloaded symbol?

Seems evident that that printing a list also rely on printing an element, that adding pairs rely on adding it's elements

Impact on the resolution

Recursion makes recursive the overloading resolution step, since resolution can introduce new implicit arguments

Recursions Must Be Well Founded

Example

```
# let list_size = function
# | []      → 0
# | x::xs → size x + size xs
- val list_size :
-    [size : α → Int.t, size : List.t α → Int.t]
-    ⇒ List.t α → Int.t
# instance size list_size
# size
#  [(1,2); (3,4)]
```

Recursions Must Be Well Founded

Example

```
# let list_size = function
# | []     → 0
# | x::xs → size x + size xs
- val list_size :
-   [size : α → Int.t, size : List.t α → Int.t]
-   ⇒ List.t α → Int.t
# instance size list_size
# list_size size size
#  [(1,2); (3,4)]
```

Recursions Must Be Well Founded

Example

```
# let list_size = function
# | []     → 0
# | x::xs → size x + size xs
- val list_size :
-   [size : α → Int.t, size : List.t α → Int.t]
-   ⇒ List.t α → Int.t
# instance size list_size
# list_size size (list_size size size)
#  [(1,2); (3,4)]
```

Recursions Must Be Well Founded

Example

```
# let list_size = function
# | []      → 0
# | x::xs → size x + size xs
- val list_size :
-   [size : α → Int.t, size : List.t α → Int.t]
-   ⇒ List.t α → Int.t
# instance size list_size
# list_size size (list_size size (list_size ...))
#  [(1,2); (3,4)]
```

Recursions Must Be Well Founded

Example

```
# let rec list_size = function
# | []     → 0
# | x::xs → size x + list_size xs
- val list_size :
-    [size : α → Int.t]
-    ⇒ List.t α → Int.t
# instance size list_size
# size
#  [(1,2); (3,4)]
```

Recursions Must Be Well Founded

Example

```
# let rec list_size = function
# | []     → 0
# | x::xs → size x + list_size xs
- val list_size :
-    [size : α → Int.t]
-    ⇒ List.t α → Int.t
# instance size list_size
# list_size size
#  [(1,2); (3,4)]
```

Recursions Must Be Well Founded

Example

```
# let rec list_size = function
# | []     → 0
# | x::xs → size x + list_size xs
- val list_size :
-   [size : α → Int.t]
-   ⇒ List.t α → Int.t
# instance size list_size
# list_size (pair_size size size)
#  [(1,2); (3,4)]
```

Recursions Must Be Well Founded

Example

```
# let rec list_size = function
# | []     → 0
# | x::xs → size x + list_size xs
- val list_size :
-    [size : α → Int.t]
-    ⇒ List.t α → Int.t
# instance size list_size
# list_size (pair_size int_size int_size)
#  [(1,2); (3,4)]
```

Polymorphic Recursion

Example

```
# type sequence α = Unit | Seq α (sequence (α * α))
# val seq_size :
#   [size : α → Int.t] ⇒ sequence α → Int.t
# instance size seq_size
# let seq_size = function
# | Unit → 0
# | Seq x xs → size x + size xs
- val seq_size : [size : α → Int.t]
-                ⇒ sequence α → Int.t
```

Outline

Explicit overloading

Minor feature but makes the implicit argument system more complete

Example

```
- val implicit : [size : α → Int.t]
-                  ⇒ List.t α → Int.t
# explicit_overloading explicit implicit
- val explicit : (α → Int.t) → List.t α → Int.t
```

Default Values

Can be really useful too

Example

```
# let print_default oc _ =
#   IO.print_string oc "?"
- val print_default : IO.output → α → unit
# default_overloading print print_default
```

Higher order kinds

Extending the type system with these kinds makes the system more fine grained

Type algebra becomes

- Type constructors are implicitly sorted, at definition time
- Type variables are explicitly sorted, just checked

-
  ```
  T ::= (X, S) | (A, S) | T T

  S ::= * | S → S
  ```

No it's not higher order unification since the application is not reduced

Higher order kinds, examples

Example

```
# overload map :
#   (α → β) → **container α → **container β

# overload bind :
#   **monad α → (α → **monad β) → **monad β

# type view **container α =
#    Nil | Cons α (**container α)

# type stateM σ **monad α =
#    State (σ → **monad (α * σ))
```

Outline

Conclusion and questions

So, ready for overloading?