Type System for a Polymorphic Multi-Stage Programming Language

Atsushi Igarashi
(Kyoto Univ.)

Joint work with Megumi Kobayashi

MetaOCaml

[Calcagno, Taha, Huang, Leroy; GPCE03]

An extension of OCaml with features for multi-stage programming (MSP)

- (Hygienic) quasi-quotation
- Eval (a.k.a. run)
- Cross-stage persistence
- Strong type system (for a pure fragment)

Type-safe quasiquotation and eval

```
# let c1 = .< 3 * 3 >.;;
val c1 : int code = .< 3 * 3 >.
# let f c = .< float_of_int .~c >.;;
val f : int code → float code = <fun>
# let c2 = f c1;;
val c2 : float code =
            .< float_of_int (3 * 3)>.
# let x = !. c2;;
val x : float = 9.0
```

I will omit dots and use “eval” for !.

Cross-Stage Persistence (CSP)

- A value created outside of quotations can be referenced inside (namely, at a later stage)
  - CSP is limited for variable references in MetaOCaml

```
# let f c = < float_of_int .~c >.;;
val f : int code → float code = <fun>

# let r = ref 2;;
# let c = <fun x → r := !r + x> in
  (eval c) 4; !r;;
val - : int = 6
```

Specializing the Power function

```
# let rec pow' n c =
  if n = 0 then <1>
  else < ~c * ~(pow' (n-1) c) >;;

# let pow n =
    < fun x → ~(pow' n <x>)>;;

# let pow3 = pow 3;;
val - : (int → int) code
  = < fun x → x * x * x * 1 >
# (eval pow3) 5;;
val - : int = 125
```

Specializing a polymorphic function

```
# let rec iter' n f x =
    if n = 1 then <~f ~x>
    else < ~f ~x; ~(iter' (n-1) f x)>;;
val iter' :
  int→(α→unit) code→α code→unit code

# let iter n =
     < fun f x → ~(iter' n <f> <x>)>;;
val iter :
  int → ((α → unit) → α → unit) code
```

Polymorphism is lost by specialization

```
# let twice = iter 2;;
val twice : ((_α→unit) → _α→unit) code
 = <fun f x → f x; f x>
```

- Due to value restriction, polymorphism is lost
- _α can be instantiated only once

Polymorphism can be recovered

- By making RHS a syntactic value (i.e., quotation)

```
# let twice' =
     < fun f x → ~(iter 2) f x>;;
val twice' : ((α→unit) → α→unit) code
 = <fun f0 x0 →
       (fun f x → f x; f x) f0 x0>

# let twice'' =
     < fun f x → ~(iter' 2 <f> <x>)>;;
val twice'' : ((α→unit) → α→unit) code
 = <fun f x → f x; f x>
```

Value Restriction

- `e1` in `let x = e1 in e2` can be given a polymorphic type, only when `e1` is a *syntactic value* (e.g., variable, `fun`, quotation of fun)
  - In OCaml (and MetaOCaml) "relaxed" value restriction [Garrigue] is used
- However, in MetaOCaml, a syntactic value can involve computation as in `twice'`
- Is this really safe?

No, not really… [Shan&Kiselyov]

- By using cross-stage persistence (CSP), this "naive" value restriction can be shown to be unsound!
- True value restriction rejects the counter example (and probably sound)
  - But would make many useful examples monomorphic
  - No way to specialize polymorphic functions?

Counterexample by Shan & Kiselyov

```
# let c =
    <let f = fun () →
         ~(let r = ref [] in <r>) in
      f() := [1];
      "foo" :: !(f()) >
val c : string list code = …
# eval c;;
```

- `f` is given a polymorphic type `unit`→α `list`
  - RHS is a function "value", even though it involves allocation of a reference to an empty list

Our Work

Type system for MiniML$^{>\%}$

- MetaOCaml-like calculus $\lambda^{>\%}$ [Hanada&I.'14]

  + let-polymorphism

  + references

(N.B. The so-called "scope extrusion problem" is not addressed)

Our Approach

- Based on imperative type variables [Tofte]
    - To prevent "polymorphic references" from being allocated
- Enhancement to take staging into account

The Rest of The Talk

- Review of Tofte's type discipline
- Applying Tofte's to MetaOCaml
- Staged imperative type variables

Problem of naive let-polymorphism

- Unsound in the presence of imperative features

```
# let r = ref [];;
val r : α list ref
# r := [1];; (* use as int list ref *)
val - : unit = ()
# "foo" :: !r;;
???        (* use as string list ref *)
```

Tofte's idea

- Allocation of a reference involving implicitly bound type variables leads to unsoundness

```
let r = Λα.ref ([] : α list) in …
```

- If RHS is a value, type variables are instantiated by the time refs are allocated

```
let r = Λα.fun () → ref ([]:α list) in
r() := [1]; "foo" :: r();
```

→ When RHS is not a value, don't abstract type variables that occurs under `ref`

Distinguishing applicative and imperative type variables

- Applicative type variables
  - cannot appear under `ref`
  - can be bound/abstracted at any `let`
- Imperative type variables
  - can appear under `ref`
  - can be bound/abstracted only at `let` with a value as RHS
  - can be instantiated only by types w/o applicative
- Value restriction = no applicative type vars

Examples revisited

```
# let r = Λα::app.ref ([]:α list) in
  r := [1]; “foo” :: !r
```

- Ill typed, because applicative var. appears under ref

```
# let r = Λα::imp.ref ([]:α list) in
  r := [1]; “foo” :: !r
```

- Ill-typed, because RHS is not a value

The Rest of The Talk

- Review of Tofte's type discipline
- Applying Tofte's to MetaOCaml
- Staged imperative type variables

Applying Tofte to MetaOCaml: Specialization of polymorphic code

- If no reference types are involved, all lets can be safely polymorphic

```
# let twice = Λα::app. iter 2;;
val twice : ((α→unit) → α → unit) code
 = <fun f x → f x; f x>
# let twice'' = Λα::app.
    < fun f x -> ~(iter' 2 <f> <x>)>;;
val twice'' : ((α→unit) → α→unit) code
 = <fun f x → f x; f x>
```

Applying Tofte to MetaOCaml: Rejecting the Counterexample

```
# let c =
    <let f = Λα::imp.fun () →
        ~(let r = ref ([]:α list)
          in <r>) in
     f() := [1];
     "foo" :: !(f()) >.
```

- Rejected under true value restriction
  - RHS of `let f =` is an abstraction but *not a value!*

Slight Variant

```
# let c =
    <let f =
      Λα::imp.fun()→ref ([]:α list) in
     f() := [1];
     “foo” :: !(f()) >;;
```

- Accepted because RHS is now a proper value (abstraction *w/o unquote*) and imperative α can be abstracted

How About This One?

```
# let twice'n'return = Λα::???.
    <fun f x → ~(iter' 2 <f> <x>); !x>;;
val twice'n'return :
  ((α ref → unit) → α ref → α ) code
 = <fun f x → f x; f x; !x>
```

Unfortunately, it is rejected:

- α cannot be `app`, because it appears under `ref`
- α cannot be `imp`, because the RHS isn't a value

The Rest of The Talk

- Review of Tofte's type discipline
- Applying Tofte's to MetaOCaml
- Staged imperative type variables

Observations

```
# let twice'n'return =
    <fun f x → ~(iter' 2 <f> <x>); !x>;;
```

Should be safely used polymorphically because code generation is pure

- It seems safe to use α under `ref` as long as it is inside quotation

Staged Imperative Type Variables

- Imperative type var at stage 1 (`imp1`)
  - Cannot appear under `ref` outside quotation or code type
  - Can be bound/abstracted at stage-1 func def *and any stage-0 let*
  - Demoted to `imp0` if code is evaluated
- Imperative type var at stage 0 (`imp0`)
  - Can be bound/abstracted at stage-0 value def
- Applicative type var (`app`)
  - Can be bound/abstracted at any `let` (but cannot appear under `ref`)

twice'n'return revisited

```
# let twice'n'return = Λα::impl.
    <fun f (x:α ref) →
         ~(iter' 2 <f> <x>); !x>;;
val twice'n'return :
  ((α ref → unit) → α ref → α ) code
 = <fun f x → f x; f x; !x>
```

- α appears under `ref` but it's inside quotation

Counterexample revisited

- α cannot be **imp1**, because it is used outside quotation (that is, in the type of **r**)

```
# let c =
    <let f = Λα::imp1.fun () →
        ~(let r = ref [] in <r>) in
     f() := [1];
     "foo" :: !(f()) >
```

Flavor of Formal Bits (1/3)

- MiniML$^{>\%}$
  - based on $\lambda^{>\%}$ [Hanada&I.'14]
    - Classifiers to represent *how thick a quotation is*
    - Quotation indexed by classifiers: $\langle\gamma\, M\rangle$
    - CSP for any terms $\%\gamma\, M$
    - Classifier abstraction: $\Lambda\gamma.M$
    - Classifier application: $M\,(\gamma_1 \ldots \gamma_n)$
    - Eval as derived form: $(\Lambda\gamma.\langle\gamma\, M\rangle)\ \varepsilon \ \to M$
  - Type/classifier abstraction restricted at let
  - References

Empty sequence (thickness is zero)

Flavor of Formal Bits (2/3)

- Imperative type vars are classified (kinded) by a set of classifiers
  - $\alpha :: \mathrm{imp}\{\gamma_1,\ldots,\gamma_n\}$ means α gets instantiated by the time $\gamma_1,\ldots,\gamma_n$ are instantiated by ε
- Judgments:
  - $\Gamma \vdash M : T \mathbin{@} \gamma_1\ldots\gamma_n$ *M has type T at stage $\gamma_1\ldots\gamma_n$*
  - $\Gamma \vdash T :: \mathrm{imp}\{\gamma_1,\ldots,\gamma_n\}$ *T* ref can be used at stage containing only γ_i
    - $\alpha::\mathrm{app} \nvdash \alpha \text{ list} :: \mathrm{imp}\{\varepsilon\}$
    - $\alpha::\mathrm{imp}\{\varepsilon\} \vdash \alpha \text{ list} :: \mathrm{imp}\{\gamma\}$
    - $\alpha::\mathrm{imp}\{\gamma\} \nvdash \alpha \text{ list} :: \mathrm{imp}\{\varepsilon\}$

Flavor of Formal Bits (3/3)

$$\frac{\Gamma \vdash M : T \ @\ \gamma_1 \ldots \gamma_n \qquad \Gamma \vdash T :: \mathrm{imp}\{\gamma_1, \ldots, \gamma_n\}}{\Gamma \vdash \mathrm{ref}\ M : T\ \mathrm{ref}\ @\ \gamma_1 \ldots \gamma_n}$$

$$\frac{\begin{array}{c}\Gamma, \gamma, \alpha{::}K \vdash \lambda y.M : T' \ @\ \gamma_1 \ldots \gamma_n \\ \Gamma, x{:}\ \forall\gamma.\forall\alpha{::}K.T' \vdash N : \mathrm{T}\ @\ \gamma_1 \ldots \gamma_n \\ K = \mathrm{app}\ \mathrm{or}\ \mathrm{imp}\{\gamma_1, \ldots, \gamma_n\}\end{array}}{\Gamma \vdash \mathrm{let}\ x = \Lambda\gamma.\Lambda\alpha{::}K.\lambda y.M\ \mathrm{in}\ \mathrm{N}{:}\ T\ @\ \gamma_1 \ldots \gamma_n}$$

imp*i* can be abstracted at stage-*i* fun def

$$\frac{\begin{array}{c}\Gamma, \gamma, \alpha{::}K \vdash M : T' \ @\ \gamma_1 \ldots \gamma_n \\ \Gamma, x{:}\ \forall\gamma.\forall\alpha{::}K.T' \vdash N : \mathrm{T}\ @\ \gamma_1 \ldots \gamma_n \\ K = \mathrm{app}\ \mathrm{or}\ \mathrm{imp}\{\gamma, \gamma_1, \ldots, \gamma_n\}\end{array}}{\Gamma \vdash \mathrm{let}\ x = \Lambda\gamma.\Lambda\alpha{::}K.M\ \mathrm{in}\ \mathrm{N}{:}\ T\ @\ \gamma_1 \ldots \gamma_n}$$

imp1 can be abstracted at any stage-0 let

Technical Results So Far

- Operational semantics
  - Scope extrusion raises a run-time exception, which this type system doesn't care about
- Type system
- Soundness proof

Summary

- Naive value restriction in MetaOCaml is unsafe
- Tofte's type discipline for MSP can be adapted
  - Staged imperative type variables
  - c.f. Weak polymorphism in SML/NJ

Future work:

- Type inference
- Investigation of naive value restriction for a sublanguage where the use of CSP is restricted
- Scope extrusion problem