An Extension of HM(X) with Bounded Abstract and Polymorphic Data-Types

Vincent Simonet

Vincent.Simonet@inria.fr
http://cristal.inria.fr/~simonet/

INRIA ROCQUENCOURT

Reminder: HM(X)

[Odersky Sulzmann Wehr, 1999]

HM(X) is a generic and constraint based presentation of type systems of the Hindley–Milner family, with let polymorphism and full type inference.

- May feature subtyping and custom constraints forms.
- Allows a modular approach of type inference, which is reduced to constraint solving.

Reminder: abstract data-types in ML

[Odersky Läufer, 1992]

Existential types are introduced as an extension of ML data-types:

```
type t = K of Exists β . β list * (β -> unit)
```

▶ This extension preserves type inference

No type annotation is required in the source code: data constructors introduction and elimination are sufficient to guide the type checker.

Reminder: abstract data-types in ML

[Odersky Läufer, 1992]

Existential types are introduced as an extension of ML data-types:

```
type t = K of Exists β . β list * (β -> unit)
```

▶ Values are explicitly packed into existential types by data constructors:

```
K ([3; 42; 111], print_int)
K (["Hello"; "World"], print_string)
```

Reminder: abstract data-types in ML

[Odersky Läufer, 1992]

Existential types are introduced as an extension of ML data-types:

```
type t = K of Exists β . β list * (β -> unit)
```

▶ Existential values are opened by pattern matching:

```
let iter =
  function K (x, f) -> List.iter f x
```

Reminder: abstract data-types in ML

[Odersky Läufer, 1992]

Existential types are introduced as an extension of ML data-types:

```
type t = K of Exists β . β list * (β -> unit)
```

▶ Existential type variables must not escape their scope. The following piece of code is ill-typed:

```
let open = function K (x, _) -> x
```

This work

Goal:

- Extending Odersky and Laüfer system with subtyping,
- Allowing bounded quantifications,
- Preserving type inference *à la ML*.

Proposal:

- A conservative extension of HM(X) with bounded existential and universal data-types,
- A realistic algorithm for solving constraints in the case of structural subtyping.

A concrete example

Example's purpose

Before giving a formal description of our contributions, we introduce them through a concrete example:

- This summarizes the requirement the system must fulfill,
- This gives an informal overview of our proposal.

The background is the Flow Caml language, but no particular knowledge of its type system is required to understand them.

Flow Caml in one slide

Flow Caml is an extension of the Objective Caml language with a type system tracing information flow.

- Usual ML types are annotated by security levels, which represent principals:

$$!alice\ \mathsf{int} \qquad !bob\ \mathsf{int} \qquad !clients\ \mathsf{int} \qquad \alpha\ \mathsf{int}$$

- A partial order between these levels specifies legal information flow, hence the type system has subtyping.

$$!alice \leq !clients \qquad !alice\ \mathsf{int} \leq !clients\ \mathsf{int}$$

The initial problem

Current Flow Caml data-type declarations look like Caml ones:

```
type (β:level) client_info =
  { cash: β int;
    send_msg: β int -> unit;
    ...
  }
```

▶ Problem:

Types of two distinct clients

$!alice$ client_info $!bob$ client_info

do not have an upper bound. As a result, they cannot be stored in the same data structure, *e.g.* a list.

Bounded existential data-type

type client_info = Exists β with $\beta \leq$ $!clients$.
  { cash: β int;
    send_msg: β int -> unit;
    ...
  }

▶ All records about clients now have the common type

client_info

Hence, they can be stored in a list of type client_info list.

Iterating over a clients list

The function `send_balances` iterates over a list of clients and sends to each of them a message indicating its current balance:

$\exists\beta[\beta \leq\ !clients]$

β int β int $\longrightarrow$ unit

```
let rec send_balances = function
    [] -> []
  | { cash = x; send_msg = f } :: tl ->
      f x; send_balances tl
```

β int $\leq$ β int

▶ Typing the second clause of the pattern matching yields the constraint:

$$\forall\beta.(\beta \leq\ !clients) \Rightarrow (\beta\ \mathsf{int} \leq \beta\ \mathsf{int})$$

Summing a clients list

The function `total` computes the total balance of the bank from the clients file. It's principal type is client_info list $\longrightarrow$ $!clients$ int.

client_info list $\longrightarrow$ α int

```
let rec total = function
  [] -> 0
| { cash = x } :: tl ->
    x + total tl
```

$\exists\beta[\beta \leq\ !clients]$

β int

β int $\leq$ α int

▶ Typing the second clause of the pattern matching yields the constraint:

$$\forall\beta.(\beta \leq\ !clients) \Rightarrow (\beta \leq \alpha)$$

which is equivalent to

$$!clients \leq \alpha$$

An illegal information flow

The function `illegal_flow` tries to send information about one client to another client:

$\exists\beta_1[\beta_1 \leq\, !clients]$ β_1 int

$\exists\beta_2[\beta_2 \leq\, !clients]$ β_2 int $\longrightarrow$ unit

```
let illegal_flow = function
  { cash = x1 } :: { send_msg = f2 } :: _ ->
    f2 x1
| _ -> ()
```

β_1 int $\leq \beta_2$ int

▶ Typing the first clause of the pattern matching yields the unsatisfiable constraint:

$$\forall\beta_1\beta_2.(\beta_1 \leq\, !clients \wedge \beta_2 \leq\, !clients) \Rightarrow (\beta_1 \leq \beta_2)$$

The type system

The language

Types: $\tau ::= \alpha, \beta, \ldots \mid \tau \rightarrow \tau \mid \varepsilon(\bar{\tau})$

Constraints: $C, D ::= \tau \leq \tau \mid C \wedge C \mid \exists \alpha . C$

Every existential type constructor has a declaration of the form:

$$\text{type } \varepsilon(\bar{\alpha}) = \exists \bar{\beta}[D].\tau$$

```
type client_info = Exists β with β ≤ !clients .
  { cash: β int;
    send_msg: β int -> unit;
    ...
  }
```

The language

Types: $\tau ::= \alpha, \beta, \ldots \mid \tau \rightarrow \tau \mid \varepsilon(\bar{\tau})$

Constraints: $C, D ::= \tau \leq \tau \mid C \wedge C \mid \exists \alpha. C$

Every existential type constructor has a declaration of the form:

$$\text{type } \varepsilon(\bar{\alpha}) = \exists \bar{\beta}[D].\tau$$

Expressions: $e ::= \ldots \mid \langle e \rangle_\varepsilon \mid \text{open}_\varepsilon\, e_1 \text{ with } e_2$

Semantics: $\text{open}_\varepsilon\, \langle v \rangle_\varepsilon \text{ with } (\lambda x.e) \rightarrow (\lambda x.e)\, v$

The key typing rule

Typing judgments have the same form as in HM(X).

$$\frac{C, \Gamma \vdash e_1 : \varepsilon(\bar{\alpha}) \qquad \varepsilon(\bar{\alpha}) \triangleq \exists \bar{\beta}[D].\tau' \qquad C, \Gamma \vdash e_2 : \forall \bar{\beta}[D].\tau' \to \tau \qquad \bar{\beta} \# \mathrm{fv}(\tau)}{C, \Gamma \vdash \mathsf{open}_{\varepsilon}\, e_1\, \mathsf{with}\, e_2 : \tau}$$

▶ First contribution of the paper:

I proved the type system is safe

Type inference

As usual in constraint-based type systems, type inference is reduced to constraint solving: we let $(\!|\Gamma \vdash e : \tau|\!)$ be the minimal constraint required for e to have type τ in the environment Γ.

$$(\!|\Gamma \vdash \mathsf{open}_\varepsilon\, e_1 \,\mathsf{with}\, e_2 : \tau|\!) = \exists \bar{\alpha}. \begin{cases} (\!|\Gamma \vdash e_1 : \varepsilon(\bar{\alpha})|\!) \wedge \\ \exists \bar{\beta}.D \wedge \forall \bar{\beta}.D \Rightarrow (\!|\Gamma \vdash e_2 : \tau' \rightarrow \tau|\!) \end{cases}$$

where $\varepsilon(\bar{\alpha}) \triangleq \exists \bar{\beta}[D].\tau'$

▶ **Reminder:**

$$\frac{C, \Gamma \vdash e_1 : \varepsilon(\bar{\alpha}) \qquad \varepsilon(\bar{\alpha}) \triangleq \exists \bar{\beta}[D].\tau' \qquad C, \Gamma \vdash e_2 : \forall \bar{\beta}[D].\tau' \rightarrow \tau \qquad \bar{\beta} \,\#\, \mathrm{fv}(\tau)}{C, \Gamma \vdash \mathsf{open}_\varepsilon\, e_1 \,\mathsf{with}\, e_2 : \tau}$$

Type inference

As usual in constraint-based type systems, type inference is reduced to constraint solving: we let $(\!|\Gamma \vdash e : \tau|\!)$ be the minimal constraint required for e to have type τ in the environment Γ.

$$(\!|\Gamma \vdash \mathsf{open}_\varepsilon\, e_1 \,\mathsf{with}\, e_2 : \tau|\!) = \exists\bar{\alpha}. \begin{cases} (\!|\Gamma \vdash e_1 : \varepsilon(\bar{\alpha})|\!) \wedge \\ \exists\bar{\beta}.D \wedge \forall\bar{\beta}.D \Rightarrow (\!|\Gamma \vdash e_2 : \tau' \rightarrow \tau|\!) \end{cases}$$

where $\varepsilon(\bar{\alpha}) \triangleq \exists\bar{\beta}[D].\tau'$

▶ Problem:

The generated constraints include (restricted) forms of universal quantification and implication, which are generally not handled by constraint solvers for subtyping.

Solving constraints: the case of structural subtyping

Reminder: structural subtyping

- Comparable types must have the same shape,
- They can only differ by their atomic leaves,
- In particular, there is no lowest ($\bot$) or greatest ($\top$) type,
- Naturally arises when extending ML type system with atomic annotations to perform static analyses.

The problem

Type inference requires solving constraints that include universal quantifiers and implications.

- Efficient (polynomial) algorithms that decide top-level implication of constraints are known ($C_1 \Rightarrow C_2$, where all free variables are implicitly universally quantified).

  But our constraints $\forall\bar{\beta}.D \Rightarrow C$ may have free type variables.

- The first order theory of structural subtyping is decidable [Kuncak Rinard, LICS 2003]

  But the given algorithm has a non-elementary complexity.

Our approach

We strike a compromise between expressiveness and efficiency, thanks to the particular form of constraints produced by type inference:

$$\exists\bar{\beta}.D \wedge \forall\bar{\beta}.D \Rightarrow C$$

- Every universal quantifier $\forall\bar{\beta}.D \Rightarrow \cdots$ comes with the constraint $\exists\bar{\beta}.D$.
- The quantification bound $\forall\bar{\beta}.D \Rightarrow \cdots$ comes from a type declaration type $\varepsilon(\bar{\alpha}) \triangleq \exists\bar{\beta}[D].\tau$. As a result, some restrictions can be imposed about its form.

Restrictions on quantification bounds

Universally quantified variables must have at most one external lower bound and one external upper bound:

$$
\begin{array}{ll}
(1) & \forall \beta_1 \beta_2 \beta_3 . (\beta_1 \leq \beta_2 \leq \beta_3) \Rightarrow \cdots \\
(2) & \forall \beta_1 \beta_2 . (\alpha_1 \leq \beta_1 \leq \alpha_2 \wedge \alpha_1 \leq \beta_2 \leq \alpha_2) \Rightarrow \cdots \\
(3) & \forall \beta_1 \beta_2 . (\alpha_1 \leq \beta_1 \leq \beta_2 \leq \alpha_2) \Rightarrow \cdots
\end{array}
$$

Multiple bounds yield constraints whose resolution is expensive, thus thay are disallowed in data-type declarations:

$$\forall \beta . (\beta \leq \alpha_1 \wedge \beta \leq \alpha_2) \Rightarrow (\beta \leq \alpha) \quad \equiv \quad \alpha_1 \sqcap \alpha_2 \leq \alpha$$

▶ Second contribution of the paper:

I designed a realistic algorithm for solving constraints under the above restriction, and proved its correctness.

Current and future work

- Implementing this framework in the `Flow Caml` system.
- François Pottier and I designed an extension of HM(X) with guarded algebraic data-types [Xi Chen Chen, 2003]: they may be described as a combination of bounded existential types and sum types.