An extension of HM(X) with bounded existential and universal data-types
(To appear at ICFP’03)

Vincent Simonet
INRIA Rocquencourt — Cristal project

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

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 (e.g. human beings !alice, !bob, ...). A partial order between these levels specifies legal information flow, hence the type system has subtyping.

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

Problem: the types !alice client_info and !bob client_info are not comparable.
Odersky and Läufer proposed an extension of ML where data-types declarations may introduce existentially quantified variables:

\[
\text{type } t = K \text{ of } \exists \ 'a \ . \ 'a \ list \ * \ (\ 'a \ -> \ \text{unit})
\]

This extension preserves type inference: the annotation provided by the introduction and the matching of the constructor \( K \) are sufficient to guide the type synthetizer.

```ml
let v1 = K ([3; 42; 111], print_int)
let v2 = K (st "Hello"; "World"], print_string)
let iter = function K (x, f) -> List.iter f x
```

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

```ml
let open = function K (x, _) -> x
```
ML with First-Class Polymorphic Types

Symmetrically, universally quantified type variables can be introduced in data-types declarations [Rémy, 1994]:

```
  type t = L of ForAll 'a . ('a list -> 'a)
```

They are in particular useful in presence of abstract data-types:

```
  let apply g = function K (x, f) -> f (g x)
```

is ill typed, but one can write:

```
  let apply (L g) = function K (x, f) -> f (g x)
```

(Poor man’s first class polymorphism)
Our work

HM(\(X\)) is a generic constraint-based type inference system with let-polymorphism. It generalizes Hindley-Milner type system. It is parametrized by the first-order logic \(X\), which is used to express types and constraints relating them. The type inference problem is reduced to solving constraints in the logic.

- We define a conservative extension of \(HM(\mathcal{X})\) with bounded existential and universal data-types.

- We propose a realistic algorithm for solving constraints in the case of structural subtyping.
The type system
Types and constraints

We assume two distinct sets of existential $\varepsilon$ and universal $\pi$ type constructors.

$$
\tau ::= \alpha, \beta, \ldots \mid \tau \to \tau \mid \varepsilon(\vec{\tau}) \mid \pi(\vec{\tau}) \quad \text{(type)}
$$

$$
C, D ::= \tau \leq \tau \mid C \land C \mid \exists \alpha.C \quad \text{(constraint)}
$$

$$
\sigma ::= \forall \bar{\alpha}[C].\tau \quad \text{(scheme)}
$$

Every data-type must be introduced by a declaration:

$$
\varepsilon(\bar{\alpha}) \triangleq \exists \bar{\beta}[D].\tau \quad \pi(\bar{\alpha}) \triangleq \forall \bar{\beta}[D].\tau
$$
Subtyping

The interpretation of the subtyping order between types is left open. However, \( \rightarrow \), \( \varepsilon \) and \( \pi \) types must be incomparable and the variances of the existential and universal type constructors must fit their logical interpretation:

\[
\varepsilon(\bar{\alpha}_1) \triangleq \exists \bar{\beta}_1[D_1].\tau_1 \quad \varepsilon(\bar{\alpha}_2) \triangleq \exists \bar{\beta}_2[D_2].\tau_2
\]

with \( \bar{\beta}_2 \not\approx \text{fv}(\tau_1) \) imply

\[
D_1 \land \varepsilon(\bar{\alpha}_1) \leq \varepsilon(\bar{\alpha}_2) \vdash \exists \bar{\beta}_2.(D_2 \land \tau_1 \leq \tau_2)
\]

\[
\pi(\bar{\alpha}_1) \triangleq \forall \bar{\beta}_1[D_1].\tau_1 \quad \pi(\bar{\alpha}_2) \triangleq \forall \bar{\beta}_2[D_2].\tau_2
\]

with \( \bar{\beta}_1 \not\approx \text{fv}(\tau_2) \) imply

\[
D_2 \land \pi(\bar{\alpha}_1) \leq \pi(\bar{\alpha}_2) \vdash \exists \bar{\beta}_1.(D_1 \land \tau_1 \leq \tau_2)
\]

Several instances: unification, non-structural subtyping, structural subtyping.
The type system

The language

We extend the \(\lambda\)-calculus with explicit constructs for packing and opening existential and universal values:

\[
e ::= x \mid \lambda x. e \mid e \ e \mid \text{let } x = e \text{ in } e \quad \text{(expression)}
\]
\[
\mid \langle e \rangle_\epsilon \mid \text{open}_\epsilon e \text{ with } e
\]
\[
\mid \langle e \rangle_\pi \mid \text{open}_\pi e
\]

The (call-by-value) semantics is extended as follows:

\[
\text{open}_\epsilon \langle v \rangle_\epsilon \text{ with } (\lambda x. e) \rightarrow (\lambda x. e) v \quad (\epsilon)
\]
\[
\text{open}_\pi \langle v \rangle_\pi \rightarrow v \quad (\pi)
\]
Standard HM(X) typing rules

\[
\begin{align*}
\text{VAR} & \quad \Gamma(x) = \forall \bar{\alpha}[D].\tau \quad C \vdash D \\
& \quad \quad \quad \quad \quad \quad C, \Gamma \vdash x : \tau
\end{align*}
\]

\[
\begin{align*}
\text{ABS} & \quad C, \Gamma[x \mapsto \tau'] \vdash e : \tau \\
& \quad \quad \quad \quad \quad \quad C, \Gamma \vdash \lambda x.e : \tau' \rightarrow \tau
\end{align*}
\]

\[
\begin{align*}
\text{APP} & \quad C, \Gamma \vdash e_1 : \tau' \rightarrow \tau \quad C, \Gamma \vdash e_2 : \tau' \\
& \quad \quad \quad \quad \quad \quad C, \Gamma \vdash e_1 \ e_2 : \tau
\end{align*}
\]

\[
\begin{align*}
\text{LET} & \quad C, \Gamma \vdash e_1 : \sigma \quad C, \Gamma[x \mapsto \sigma] \vdash e_2 : \tau \\
& \quad \quad \quad \quad \quad \quad C, \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau
\end{align*}
\]

\[
\begin{align*}
\text{GENERALIZE} & \quad C \land D, \Gamma \vdash e : \tau \quad \bar{\alpha} \not\in \text{fv}(C, \Gamma) \\
& \quad \quad \quad \quad \quad \quad C \land \exists \bar{\alpha}.D, \Gamma \vdash e : \forall \bar{\alpha}[D].\tau
\end{align*}
\]

\[
\begin{align*}
\text{SUB} & \quad C, \Gamma \vdash e : \tau' \quad C \vdash \tau' \leq \tau \\
& \quad \quad \quad \quad \quad \quad C, \Gamma \vdash e : \tau
\end{align*}
\]

\[
\begin{align*}
\text{HIDE} & \quad C, \Gamma \vdash e : \tau \quad \bar{\alpha} \not\in \text{fv}(\Gamma, \tau) \\
& \quad \quad \quad \quad \quad \quad \exists \bar{\alpha}.C, \Gamma \vdash e : \tau
\end{align*}
\]
Typing rules for the new constructs

**Exist**

\[
\begin{align*}
\text{EXIST} & \\
C, \Gamma \vdash e : \tau & \quad \varepsilon(\bar{\alpha}) \triangleq \exists \bar{\beta}[D].\tau & \quad C \models D \\
\hline \hline
C, \Gamma \vdash \langle e \rangle_\varepsilon : \varepsilon(\bar{\alpha})
\end{align*}
\]

**OpenExist**

\[
\begin{align*}
\text{OPENEXIST} & \\
C, \Gamma \vdash e_1 : \varepsilon(\bar{\alpha}) & \quad \varepsilon(\bar{\alpha}) \triangleq \exists \bar{\beta}[D].\tau' & \quad C, \Gamma \vdash e_2 : \forall \bar{\beta}[D].\tau' \to \tau & \quad \bar{\beta} \not\equiv \text{fv}(\tau)
\hline \hline
C, \Gamma \vdash \text{open}_\varepsilon e_1 \text{ with } e_2 : \tau
\end{align*}
\]

**Poly**

\[
\begin{align*}
\text{POLY} & \\
C, \Gamma \vdash e : \forall \bar{\beta}[D].\tau & \quad \pi(\bar{\alpha}) \triangleq \forall \bar{\beta}[D].\tau \\
\hline \hline
C, \Gamma \vdash \langle e \rangle_\pi : \pi(\bar{\alpha})
\end{align*}
\]

**OpenPoly**

\[
\begin{align*}
\text{OPENPOLY} & \\
C, \Gamma \vdash e : \pi(\bar{\alpha}) & \quad \pi(\bar{\alpha}) \triangleq \forall \bar{\beta}[D].\tau & \quad C \models D \\
\hline \hline
C, \Gamma \vdash e : \tau
\end{align*}
\]
Type safety

An expression $e$ is well-typed if $C, \emptyset \vdash e : \tau$ holds for some satisfiable constraint $C$.

The type system has standard subject-reduction and progress theorems.

“Well-typed expressions do not go wrong”
Generating constraints
Outline

We define an algorithm for computing principal typing judgments:

\[(\Gamma \vdash e : \tau) \rightsquigarrow C\]

The algorithm must be **correct**: for all \(\Gamma\), \(e\) and \(\tau\),

\[(\Gamma \vdash e : \tau), \Gamma \vdash e : \tau\]

and **complete**: for all \(C\), \(\Gamma\), \(e\) and \(\tau\),

if \(C, \Gamma \vdash e : \tau\) then \(C \models (\Gamma \vdash e : \tau)\).
Generating constraints

Core language

\[
(\Gamma \vdash x : \tau) = \exists \alpha. (C \land \tau' \leq \tau)
\]
where \( \Gamma(x) = \forall \alpha[C].\tau' \)

\[
(\Gamma \vdash \lambda x.e : \tau) = \exists \alpha_1 \alpha_2. ((\Gamma[x \mapsto \alpha_1] \vdash e : \alpha_2) \land \alpha_1 \rightarrow \alpha_2 \leq \tau)
\]

\[
(\Gamma \vdash e_1 e_2 : \tau) = \exists \alpha. ((\Gamma \vdash e_1 : \alpha \rightarrow \tau) \land (\Gamma \vdash e_2 : \alpha))
\]

\[
(\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau) = (\Gamma[x \mapsto \forall \alpha[C].\alpha] \vdash e_2 : \tau) \land \exists \alpha.C
\]
where \( C = (\Gamma \vdash e_1 : \alpha) \)
Existential and universal data-types

We introduce a non-standard construct in constraints:

\[ \forall \bar{\beta}. D \triangleright C \] interpreted as \[ \exists \bar{\beta}. D \land \forall \bar{\beta} D \Rightarrow C \]

\[
\begin{align*}
(\Gamma \vdash \langle e \rangle_\epsilon : \tau) &= \exists \bar{\alpha}. (\exists \bar{\beta}. (\langle \Gamma \vdash e : \tau' \rangle \land D) \land \epsilon(\bar{\alpha}) \leq \tau) \\
(\Gamma \vdash \text{open}_\epsilon e_1 \text{ with } e_2 : \tau) &= \exists \bar{\alpha}. ((\langle \Gamma \vdash e_1 : \epsilon(\bar{\alpha}) \rangle) \land \forall \bar{\beta}. D \triangleright (\langle \Gamma \vdash e_2 : \tau' \rightarrow \tau \rangle)) \\
&\quad \text{where } \epsilon(\bar{\alpha}) \triangleq \exists \bar{\beta}[D]_\tau' \\
(\Gamma \vdash \text{open}_\pi e : \tau) &= \exists \bar{\alpha}. ((\langle \Gamma \vdash e : \pi(\bar{\alpha}) \rangle) \land \exists \bar{\beta}. (D \land \tau' \leq \tau)) \\
(\Gamma \vdash \langle e \rangle_\pi : \tau) &= \exists \bar{\alpha}. (\forall \bar{\beta}. D \triangleright (\langle \Gamma \vdash e : \tau' \rangle) \land \pi(\bar{\alpha}) \leq \tau) \\
&\quad \text{where } \pi(\bar{\alpha}) \triangleq \forall \bar{\beta}[D]_\tau'
\end{align*}
\]
Summary

An expression $e$ is well-typed in $\text{HM}_{\exists\forall}(X)$ in and only if the constraint $\exists \alpha. (\emptyset \vdash e : \alpha)$ is satisfiable in the logic $X$. This constraint belongs to the following language:

$$C, D ::= \tau \leq \tau \mid C \land C \mid \exists \alpha. C \mid \forall \beta.D \triangleright C$$

where every bound $\beta.D$ of a universal quantification comes from a data-type declaration.

It remains to provide algorithms that solve these constraints.
Solving constraints: The case of structural subtyping
Overview

We need an algorithm for solving constraints which include a restricted form of universal quantification and implication.

On the one-hand, efficient (polynomial) algorithms that decide top-level implication of constraints ($C_1 \vdash C_2$, where all free variables are implicitly universally quantified) are known.

On the other hand, Kuncak and Rinard recently showed [LICS 2003] that the first order theory of structural subtyping is decidable, but their algorithm has a non-elementary complexity.

We strike a compromise between expressiveness and efficiency:

• thanks to the “weak” interpretation of $\forall\bar{\beta}.D \triangleright C$ which implies $\exists\bar{\beta}.D$,

• by restricting the form of the quantification bounds in every construct $\forall\bar{\beta}.D \triangleright C$. 
A model of structural subtyping

Let a variance $\nu$ be one of $\oplus$ (covariant), $\ominus$ (contravariant) and $\odot$ (invariant).

We assume given a set of symbols $\varphi$. Every symbol has a fixed arity $a(\varphi)$ and a signature $\text{sig}(\varphi) = [\nu_1, \ldots, \nu_{a(\varphi)}]$. Then ground types are defined by:

$$t ::= \varphi(t_1, \ldots, t_{a(\varphi)}) \quad \text{(ground type)}$$

Symbols of arity 0 are ground atoms: we suppose they are partially ordered by the lattice order $\leq_0$. Then, subtyping is defined by:

$$\frac{\varphi \leq_0 \varphi'}{\varphi \leq \varphi'} \quad \text{sig}(\varphi) = [\nu_1, \ldots, \nu_n] \quad \forall i \quad t_i \leq_{\nu_i} t'_i \Rightarrow \varphi(t_1, \ldots, t_n) \leq \varphi(t'_1, \ldots, t'_n)$$
Solving constraints: The case of structural subtyping

Shapes

In structural subtyping, two comparable types must have the same shape. We define the relation $t \approx t'$ (read: $t$ has the same shape as $t'$) by:

$$\text{sig}(\varphi) = [\nu_1, \ldots, \nu_n] \quad \forall i \quad t_i \approx \nu_i t'_i$$

$$\varphi(t_1, \ldots, t_n) \approx \varphi(t'_1, \ldots, t'_n)$$

$\approx$ is the reflexive, symmetric, transitive closure of $\leq$. Its equivalence classes are lattices.
Solving constraints: The case of structural subtyping

Expansion and decomposition

In structural subtyping, the two following equivalence rules hold:

Expansion:
\[
\varphi(\bar{\tau}) \leq \alpha \equiv \exists \bar{\alpha}.(\varphi(\bar{\alpha}) = \alpha \land \varphi(\bar{\tau}) \leq \varphi(\bar{\alpha}))
\]
\[
\equiv \exists \langle \varphi(\bar{\alpha}) = \alpha \rangle. (\varphi(\bar{\tau}) \leq \varphi(\bar{\alpha}))
\]

Decomposition:
\[
\varphi(\tau_1, \ldots, \tau_n) \leq \varphi(\tau'_1, \ldots, \tau'_n) \equiv \tau_1 \leq^{\nu_1} \tau'_1 \land \cdots \land \tau_n \leq^{\nu_n} \tau'_n
\]

Our algorithm consists in rewriting the input constraint into a solved form:

\[
\begin{align*}
\eta & ::= \varphi \mid \alpha & \text{(atom)} \\
R & ::= \emptyset \mid \eta \leq \eta \land R \mid \eta \approx \eta \land R & \text{(multiset of atomic constraints)} \\
S & ::= R \mid \exists \langle \varphi(\bar{\alpha}) = \alpha \rangle. S & \text{(solved form)}
\end{align*}
\]

(In \(\exists \langle \varphi(\bar{\alpha}) = \alpha \rangle. S\), we require \(\alpha \notin \text{fv}(S)\)).

By orienting the two above rules from left to right, we obtain an algorithm which rewrites any conjunction of inequalities into a solved form. It remains to eliminate quantifiers.
Eliminating existential quantifiers

Goal: $\exists \beta. S \leadsto S'$

$\exists \beta. []$ commutes with $\exists \langle \phi(\bar{\alpha}) = \alpha \rangle. []$

- $\exists \beta. \exists \langle \phi(\bar{\alpha}) = \alpha \rangle. S \leadsto \exists \langle \phi(\bar{\alpha}) = \alpha \rangle. \exists \beta. S$ if $\alpha \neq \beta$ (and $\beta \notin \bar{\alpha}$)
- $\exists \alpha. \exists \langle \phi(\bar{\alpha}) = \alpha \rangle. S \leadsto \exists \bar{\alpha}. S$

$\exists \beta. []$ can be eliminated when it reaches the multiset of atomic inequalities

$\exists \beta. R \leadsto \{ \eta_1 \diamond \eta_2 \mid \eta_1 \diamond \eta_2 \in R \text{ and } \eta_1, \eta_2 \neq \beta \}$

$\cup \{ \eta_1 \diamond_1 \diamond_2 \eta_2 \mid \eta_1 \diamond_1 \beta \in R \text{ and } \beta \diamond_2 \eta_2 \in R \}$

where $\diamond$ ranges over $\approx$, $\leq$ and $\geq$.

$\exists \beta. (\beta \leq \alpha_1 \land \beta \leq \alpha_2) \leadsto \alpha_1 \approx \alpha_2$
Restricting universal quantification bounds

We consider a constraint $\forall \vec{\beta}. D \triangleright C$.

- Existential quantifiers in $D$ can be fused with the universal one:
  $$\forall \vec{\beta}. (\exists \vec{\alpha}. D) \triangleright C \equiv \forall \vec{\beta} \vec{\alpha}. D \triangleright C$$

- Type constructors in $D$ can be eliminated by expansion and decomposition, e.g.
  $$\forall \beta. (\beta \leq \alpha_1 \rightarrow \alpha_2) \triangleright C \equiv \forall \beta_1 \beta_2. (\alpha_1 \leq \beta_1 \land \beta_2 \leq \alpha_2) \triangleright C[\beta_1 \rightarrow \beta_2/\beta]$$

Thus, we may assume that $D$ is a conjunction of inequalities involving atoms.
Restricting universal quantification bounds

Consider a constraint \( \forall \vec{\beta}.D \triangleright C \) and a variable \( \beta \in \vec{\beta} \). Three situations may arise:

- \( \beta \) has no external bound in \( D \), i.e. is only related to variables of \( \vec{\beta} \). In this case, \( C \) cannot constrain its shape. For instance \( \forall \beta.\text{true} \triangleright \beta \leq \alpha_1 \rightarrow \alpha_2 \) is not satifiable.

- \( \beta \) has one lower and/or upper bound(s) in \( D \).

\[
\forall \beta.(\beta \leq \alpha) \triangleright (\beta \leq \alpha'_1 \rightarrow \alpha'_2)
\equiv \exists \langle \alpha_1 \rightarrow \alpha_2 = \alpha \rangle. (\forall \beta.(\beta \leq \alpha_1 \rightarrow \alpha_2) \triangleright (\beta \leq \alpha'_1 \rightarrow \alpha'_2))
\equiv \exists \langle \alpha_1 \rightarrow \alpha_2 = \alpha \rangle. (\forall \beta_1/\beta_2.(\alpha_1 \leq \beta_1 \land \beta_2 \leq \alpha_2) \triangleright (\alpha'_1 \leq \beta_1 \land \beta_2 \leq \alpha'_2))
\equiv \exists \langle \alpha_1 \rightarrow \alpha_2 = \alpha \rangle. (\alpha'_1 \leq \alpha_1 \land \alpha_2 \leq \alpha'_2)
\]

[...]

Restricting universal quantification bounds

[...]  

- $\beta$ has several lower or upper bounds in $D$.

\[
\forall \beta. (\beta \leq \alpha_1 \land \beta \leq \alpha_2) \triangleright (\beta \leq \alpha) \\
\equiv \forall \beta. (\beta \leq \alpha_1 \sqcap \alpha_2) \triangleright (\beta \leq \alpha) \\
\equiv \alpha_1 \sqcap \alpha_2 \leq \alpha
\]

We exclude this third case.

Some examples of allowed quantification bounds:

1. $\forall \beta_1 \beta_2 \beta_3. (\beta_1 \leq \beta_2 \leq \beta_3) \triangleright \cdots$
2. $\forall \beta_1 \beta_2. (\alpha_1 \leq \beta_1 \leq \alpha_2 \land \alpha_1 \leq \beta_2 \leq \alpha_2) \triangleright \cdots$
3. $\forall \beta_1 \beta_2. (\phi_1 \leq \beta_1 \leq \beta_2 \leq \phi_2) \triangleright \cdots$
Eliminating universal quantifiers

Goal: $\forall \bar{\beta}.D \triangleright S \rightsquigarrow S'$

$\forall \bar{\beta}.D \triangleright \emptyset$ commutes with $\exists \langle \phi(\bar{\alpha}) = \alpha \rangle.\emptyset$

$\forall \bar{\beta}.D \triangleright (\exists \langle \phi(\bar{\alpha}) = \alpha \rangle.S) \rightsquigarrow \exists \langle \phi(\bar{\alpha}) = \alpha \rangle.(\forall \bar{\beta}.D[\phi(\bar{\alpha})/\alpha] \triangleright S') \quad \alpha \notin \bar{\beta}$

$\forall \alpha \bar{\beta}.D \triangleright (\exists \langle \phi(\bar{\alpha}) = \alpha \rangle.S) \rightsquigarrow \forall \alpha \bar{\beta}.D[\phi(\bar{\alpha})/\alpha] \triangleright S \quad \alpha$ bounded

$\forall \alpha \bar{\beta}.D \triangleright (\exists \langle \phi(\bar{\alpha}) = \alpha \rangle.S) \rightsquigarrow \text{failure} \quad \alpha$ unbounded

$\forall \bar{\beta}.D \triangleright \emptyset$ can be eliminated when it reaches the multiset

$\forall \bar{\beta}.D \triangleright R \rightarrow (\exists \bar{\beta}.D)$

$\cup \{\text{ub}_{\bar{\beta}.D}(\eta_1) \leq \text{lb}_{\bar{\beta}.D}(\eta_2) \mid \eta_1 \leq \eta_2 \in R \setminus D^*\}$

$\cup \{\text{sh}_{\bar{\beta}.D}(\eta_1) \approx \text{sh}_{\bar{\beta}.D}(\eta_2) \mid \eta_1 \approx \eta_2 \in R \setminus D^*\}$

$\text{ub}_{\bar{\beta}.D}(\eta)$ is the upper bound of $\eta$ under $\forall \bar{\beta}.D \triangleright \cdots$

$\text{lb}_{\bar{\beta}.D}(\eta)$ is the lower bound of $\eta$ under $\forall \bar{\beta}.D \triangleright \cdots$

$\text{sh}_{\bar{\beta}.D}(\eta)$ is the shape of $\eta$ under $\forall \bar{\beta}.D \triangleright \cdots$
Summary

Our algorithm rewrites an arbitrary constraint into a solved form.

\[ C \rightsquigarrow S \]

A solved form is satisfiable if and only if its multiset is satisfiable.

\[
\begin{align*}
\eta & ::= \varphi | \alpha \quad \text{(atom)} \\
R & ::= \emptyset | \eta \leq \eta \land R | \eta \approx \eta \land R \quad \text{(multiset of atomic constraints)} \\
S & ::= R | \exists \langle \phi(\bar{\alpha}) = \alpha \rangle.S \quad \text{(solved form)}
\end{align*}
\]
Examples
The bank example

In the lattice of security levels, we have one security level for every client (\texttt{alice}, \texttt{bob}, ...). We let \texttt{clients} be their least upper bound.

\begin{verbatim}
  type client_info = Exists 'a with 'a < clients .
  { cash: 'a int;
    send_msg: 'a int -> unit;
    ... }
\end{verbatim}
The bank example (2)

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

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

De-sugaring this example in the syntax of the current talk, we realize that the function which corresponds to the second case of the pattern matching

```
\lambda x, send, tl. (send x; send_balances tl)
```

must have the type scheme

\[
\forall \alpha \left[ \alpha \leq \text{client info list} \right]. \\
\alpha \text{ int} \rightarrow (\alpha \text{ int} \rightarrow \text{unit}) \rightarrow \text{client_info list} \rightarrow \text{unit}
\]
The function `illegal_flow` tries to send information about one client to another client:

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

Typing this piece of code yields the constraint

\[ \forall \beta_1 \beta_2. (\beta_1 \sqcup \beta_2 \leq \text{clients}) \uparrow (\beta_1 \leq \beta_2) \]

which is not satisfiable.
The bank example (4)

The function `total` computes the total balance of the bank from the clients file:

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

It receives the type scheme

```
client_info list → !clients int
```
Future work

- We intend to extend our generic type inference engine for structural subtyping, *Dalton*, in order to handle the new construct.

- Then, it will be possible to extend the *Flow Caml* system with existential and universal data-types.

- We study the possibility to make security levels also *values* of the Flow Caml language: this would allow to perform some *dynamic tests* (whose correctness must be verified statically) on existentially quantified variables when opening data-structures.
Possible work

- Giving a faithful description of the solving algorithm which describes the simplification techniques used in the implementation.
- Studying constraints resolution for other forms of subtyping.
- Introducing subtyping in more powerful extensions of ML with first order polymorphism (PolyML, ML$_F$, ...).