Generalised recursion and type inference for intersection types

Seminar for Comete– May 4th, 2004

Pascal Zimmer

INRIA Sophia Antipolis

General motivation

To design a base language with:

- functional core
- objects
- well-defined semantics, that can be realistically implemented
- ML-like inference of principal types

in the goal of adding other paradigms (migration, reactive)...

INRIA

Outline

First part:

- semantics of object languages
- a type system with degrees
- implementation, abstract machine
- mixins

Outline

First part:

- semantics of object languages
- a type system with degrees
- implementation, abstract machine
- mixins

Second part:

- intersection types
- Klop calculus
- type inference
- extensions

First part

Generalised recursion

Semantics of objects

Auto-application semantics

- model initiated by Kamin, 1988;
  reference: Abadi and Cardelli, 1996
- object = collection of pre-methods:

$$o = [\ldots, l = \varsigma(\text{self})\ b, \ldots]$$

- method call:

$$o.l \ \Rightarrow \ b\,\{\text{self} \leftarrow o\}$$

- specific typing
- inference of principal types impossible

Semantics of objects 2

Recursive record semantics

- Cardelli 1988, Wand 1994, Cook 1994
- class:

$$C = \lambda x_1 \ldots \lambda x_n \, \lambda \mathrm{self} \, \{l_1 = M_1, \ldots, l_p = M_p\}$$

- object: $o = \mathrm{fix}\,(C N_1 \ldots N_n)$
- row variables to extend the object
- no modification of the state, since self is bound to the initial object
- typing model of OCAML

Language proposition

- Wand's recursive record semantics
- ML-like references to hold the state of the object
- examples:

$$\begin{aligned} \mathit{point} = \lambda x \lambda \mathrm{self} \\ & \{\mathit{pos} = \mathrm{ref}\ x, \\ & \ \mathit{move} = \lambda y(\mathrm{self}.\mathit{pos} := !\mathrm{self}.\mathit{pos} + y)\} \end{aligned}$$

$$p = \mathrm{fix}\ (\mathit{point}\ 4)$$

$$\begin{aligned} \mathit{color\_point} = \lambda x \lambda c \lambda \mathrm{self} \\ & \{\mathit{point}\ x\ \mathrm{self}, \mathit{color} = \mathrm{ref}\ c\} \end{aligned}$$

Evaluating the fixpoint

- Problem: how can we evaluate the fixpoint ?

$$\mathrm{fix} = \lambda f\ (\mathrm{let\ rec}\ x = f x\ \mathrm{in}\ x)$$

- In SML, only allowed construct:

$$\mathrm{let\ rec}\ x = \lambda y N\ \mathrm{in}\ M$$

- We need a generalised recursion operator
- But some recursions are dangerous:

$$\mathrm{let\ rec}\ x = xV\ \mathrm{in}\ M$$

$$\mathrm{let\ rec}\ x = x + 1\ \mathrm{in}\ M$$

Type system with degrees

- Boudol, 2001
- degree = boolean information in function types and in typing contexts

$$\theta^{d} \longrightarrow \tau$$

- 0 = "dangerous", 1 = "sure"
- intuitively: is the value required or not when evaluating
- $(\text{let rec } x = N \text{ in } M)$ is typable iff N is typable with a degree 1 for x
- $(\text{let rec } x = f\,x \text{ in } M)$ is typable iff f has type $\theta^{1} \longrightarrow \tau$ ("protective" function)

Degrees - examples

- example of protective function:

$$
\begin{aligned}
point0 = \lambda\text{self}\ & \\
& \{pos = \text{ref } 0, \\
& \ \ move = \lambda y(\text{self}.pos := !\text{self}.pos + y)\}
\end{aligned}
$$

- $\text{fix} = \lambda f(\text{let rec } x = f x \text{ in } x)$
  has type: $(\tau^1 \to \tau)^0 \to \tau$
- $\lambda\text{self}\{x = 0, y = \text{self}.x\}$
  has type: $\{\rho, x : \tau\}^0 \to \{x : int, y : \tau\}$
  where ρ is a row variable
  with the constraint $\rho :: \{x\}$

Degrees - results

- subject reduction
- safety: the evaluation of a typable term never leads to an error (recursion, field access, applications...)
- algorithm for infering principal types, extension of ML's one

Unification and inference algorithms

- more “realistic” and efficient versions
- working on graphs (recursive types)
- unification of degrees, records, types
- polymorphism similar to ML, on degree, row or type variables; generalising for:

$$\text{let (rec) } x = V \text{ in } M$$

- constraints on row variables ($\rho :: L$) and degree variables;
  example: $\lambda f \lambda x(f x)$ has type
  $(\theta^\alpha \to \tau)^\beta \to \theta^\gamma \to \tau$ with $\gamma \leq \alpha$

INRIA

Abstract machine

- we need to evaluate terms with the shape

$$(\lambda \mathrm{self}\, M)\ o$$

  where o is a still unevaluated variable, knowing that the value of self is not needed to evaluate M
- usual machines for λ-calculus or ML do not allow the evaluation of generalised recursion

Abstract machine

$$\mathcal{M} = (S, \sigma, M, \xi)$$

- S: control stack
- σ: environment
- M: term to evaluate
- ξ: memory for recursive values (and references)

- set of 11 transition rules, among which a "magic" rule:

$$\begin{array}{rll} & (S :: (\sigma \lambda y M[]), \rho :: \{x \mapsto \ell\}, x, \xi) & \\ \to & (S, \sigma :: \{y \mapsto \ell\}, M, \xi) & \text{if } \xi(\ell) = \bullet \end{array}$$

Abstract machine

- operational correspondence
- determinism
- no infinite “silent” reductions
- correction:
  if the starting term is typable, then both the machine and the calculus semantics go through the same reductions

MLOBJ

`http://www-sop.inria.fr/mimosa/Pascal.Zimmer/mlobj.html`

OCAML-like interpreter...

Mixins

- goal: use higher-order constructs to build more powerful objects
- generator: $\lambda s\ \{\ldots\}$
- mixin: generator modifier

$$C = \lambda x_1 \ldots \lambda x_n \lambda g \lambda s\ \{\ldots \text{fields} \ldots \text{methods} \ldots\}$$

- instance ($\lambda s\ \{\}$ is the initial generator):

$$\text{fix}\ (C N_1 \ldots N_n (\lambda s\ \{\}))$$

- new operator:

$$\text{new} = \lambda m\ \text{fix}\ (m\ (\lambda s\ \{\}))$$

Mixins - definition

Implemented by syntactic sugar rules.

mixin	
var $l = N$	non-constant data
cst $l = N$	constant data
meth $l(\mathrm{super}, \mathrm{self}) \ = N$	method
meth $l(\mathrm{super}, \mathrm{self}) \ \leftarrow N$	method override
inherit N	inheritance
without l	field suppression
rename l **as** l'	field renaming
end	

Method call: $M\#l$

INRIA

Mixins - examples

$$
\begin{array}{ll}
\mathit{point} = \lambda x & \mathit{coloring} = \lambda c \\
\quad \textbf{mixin} & \quad \textbf{mixin} \\
\quad\quad \textbf{var}\ \mathit{pos} = x & \quad\quad \textbf{var}\ \mathit{color} = c \\
\quad\quad \textbf{meth}\ \mathit{move} \ldots & \quad\quad \textbf{meth}\ \mathit{paint} \ldots \\
\quad \textbf{end} & \quad \textbf{end}
\end{array}
$$

$$
\begin{array}{l}
\mathit{colorPoint} = \lambda x \lambda c \\
\quad \textbf{mixin} \\
\quad\quad \textbf{inherit}\ \mathit{point}\ x \\
\quad\quad \textbf{inherit}\ \mathit{coloring}\ c \\
\quad \textbf{end}
\end{array}
$$

$\Rightarrow$ multiple inheritance

Mixins - examples

$$
\begin{array}{l}
reset = \\
\quad \textbf{mixin} \\
\qquad \textbf{meth}\ reset(\mathrm{super}, \mathrm{self}) = \mathrm{self}.pos := 0 \\
\quad \textbf{end}
\end{array}
$$

$$
\begin{array}{ll}
resetPoint = \lambda x & resetColorPoint = \lambda x \lambda c \\
\quad \textbf{mixin} & \quad \textbf{mixin} \\
\qquad \textbf{inherit}\ point\ x & \qquad \textbf{inherit}\ colorPoint\ x\ c \\
\qquad \textbf{inherit}\ reset & \qquad \textbf{inherit}\ reset \\
\quad \textbf{end} & \quad \textbf{end}
\end{array}
$$

$\Rightarrow$ code sharing

Mixins - examples

mixin

meth $reset(\text{super}, \text{self}) \leftarrow$

$$\lambda d\ (\text{super}\#reset;\ \text{super}\#paint\ d)$$

end

- Typing determines which mixins can be instantiated and which cannot.
- By changing the initial generator, one can get initialisers.
- Mixins = first order values
  $\Rightarrow$ a huge expressive power
  still to be explored !

And after ?

- advanced functionalities: cloning, binary methods... :

$$\textbf{meth } eq(\mathrm{super}, \mathrm{self}) = \lambda p \; (\mathrm{self}.pos == p.pos)$$

- operationally, no problem
- typing: not enough polymorphism !
- System F ?
  type inference undecidable...
- intersection types ?
  finite-rank inference is decidable...

Second part

Inference of intersection types

History

- system $\mathcal{D}$: Coppo, Dezani, 1980; Pottinger, 1980
- principal typing: Coppo, Dezani and Venneri, 1980; Ronchi della Rocca and Venneri, 1984
- inference: Ronchi della Rocca, 1988
- system $\mathbb{I}$: Kfoury and Wells, 1999
- system E: Carlier, Kfoury, Polakow and Wells, 2004

Motivation:

to find an algorithm simpler to understand and to prove

Types syntax

$$\tau, \sigma ::= t \mid \tau_1, \ldots, \tau_n \rightarrow \sigma$$

- conjunction only at the left of an arrow
- empty sequence denoted by ω
- $\tau_1, \ldots, \tau_n \rightarrow \sigma$: type of a function waiting for an argument having *all* types τ_i

Typing rules

$$\frac{}{x : \tau \vdash x : \tau}\text{(Typ Id)}$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \setminus x \vdash \lambda x M : \Gamma(x) \to \tau}\text{(Typ } \lambda\text{)}$$

$$\frac{\Gamma \vdash M : \tau_1, \ldots, \tau_n \to \sigma \quad \forall i,\ \Gamma_i \vdash N : \tau_i}{\Gamma, \Gamma_1, \ldots, \Gamma_n \vdash MN : \sigma}\text{(Typ Appl Gen)} \qquad (n \geq 1)$$

$$\frac{\Gamma \vdash M : \omega \to \sigma \quad \Gamma_1 \vdash N : \tau_1}{\Gamma, \Gamma_1 \vdash MN : \sigma}\text{(Typ Appl } \omega\text{)}$$

INRIA

Examples

- $\vdash I : t \rightarrow t$
  $(I = \lambda x x)$

- $\vdash \mathbf{2} : (t_1 \rightarrow t_2), (t_2 \rightarrow t_3) \rightarrow t_1 \rightarrow t_3$
  $(\mathbf{2} = \lambda f \lambda x\ f(fx))$

- $\vdash \Delta : t_1, (t_1 \rightarrow t_2) \rightarrow t_2$
  $(\Delta = \lambda x(xx))$

- $\vdash K : t \rightarrow \omega \rightarrow t$
  $(K = \lambda x \lambda y\ x)$

- $\nvdash \Omega :?$ $\qquad\qquad$ $\nvdash Kx\Omega :?$
  $(\Omega = \Delta\Delta)$

Properties

- Subject reduction: If $M \to M'$, then

$$\Gamma \vdash M : \tau \implies \Gamma \vdash M' : \tau$$

- Theorem: A term M is typable in $\mathcal{D}$ if and only if M is strongly normalising (i.e. iff it has no diverging reduction).

Properties

- Subject reduction: If $M \rightarrow M'$, then

$$\Gamma \vdash M : \tau \implies \Gamma \vdash M' : \tau$$

- Theorem: A term M is typable in $\mathcal{D}$ if and only if M is strongly normalising (i.e. iff it has no diverging reduction).
- Trivial algorithm: try to strongly normalise, then type.
- Problem: does not work for an extended calculus (recursion...)
- We have the type, but not the typing tree...

INRIA

Example

$$M = F(\lambda u\, \Delta(uu))$$

with $F = \lambda x \lambda y\, y$ and $\Delta = \lambda x\, (xx)$

Example

$$M = F(\lambda u\, \Delta(uu))$$

with $F = \lambda x \lambda y\, y$ and $\Delta = \lambda x\, (xx)$

- First step:
  annotate every variable and application with a fresh type variable.

$$M^{\mathbf{t}} = (F^{\mathbf{t}}\, (\lambda u\, (\Delta^{\mathbf{t}}(u : t_4\ u : t_5) : t_6) : t_7)) : t_8$$

where $F^{\mathbf{t}} = \lambda x \lambda y (y : t_0)$
and $\Delta^{\mathbf{t}} = \lambda x\, (x : t_1\ x : t_2) : t_3$

Example - $F(\lambda u\ \Delta(uu))$

- Second step:
  for every application $(M^{\mathbf{t}} N^{\mathbf{t}}) : t$, build the constraint:

$$Typ(N^{\mathbf{t}}) \to t \ \underline{\perp}\ Typ(M^{\mathbf{t}})\ \ [ftv(N^{\mathbf{t}})]$$

Example - $F(\lambda u\ \Delta(uu))$

- Second step:
  for every application $(M^{\mathtt{t}} N^{\mathtt{t}}) : t$, build the constraint:

$$Typ(N^{\mathtt{t}}) \to t \perp Typ(M^{\mathtt{t}})\ \ [ftv(N^{\mathtt{t}})]$$

$$\left\{ \begin{array}{rcll} (t_4, t_5 \to t_7) \to t_8 & \perp & \omega \to t_0 \to t_0 & [t_1, \ldots, t_7], \\ t_6 \to t_7 & \perp & t_1, t_2 \to t_3 & [t_4, t_5, t_6], \\ t_5 \to t_6 & \perp & t_4 & [t_5], \\ t_2 \to t_3 & \perp & t_1 & [t_2] \end{array} \right\}$$

(In ML, we would add $t_4 \perp t_5$ and $t_1 \perp t_2$).

Example - $F(\lambda u\ \Delta(uu))$

Decomposition of:

$$t_6 \to t_7 \ \stackrel{\perp}{=}\ t_1, t_2 \to t_3 \ \ [t_4, t_5, t_6]$$

Updated system:

$$\left\{ \begin{array}{rclll} (t_4^1, t_4^2, t_5^1, t_5^2 \to t_3) \to t_8 & \stackrel{\perp}{=} & \omega \to t_0 \to t_0 & & [T], \\ t_6^2 \to t_3 & \stackrel{\perp}{=} & t_6^1 & & [t_4^2, t_5^2, t_6^2] \\ t_5^1 \to t_6^1 & \stackrel{\perp}{=} & t_4^1 & & [t_5^1], \\ t_5^2 \to t_6^2 & \stackrel{\perp}{=} & t_4^2 & & [t_5^2] \end{array} \right.$$

where $T = \{t_3, t_4^1, t_4^2, t_5^1, t_5^2, t_6^1, t_6^2\}$

Those equations correspond to the term:

$$F(\lambda u\ (uu)(uu))$$

Example - $F(\lambda u\ \Delta(uu))$

Decomposition of:

$$(t_4^1, t_4^2, t_5^1, t_5^2 \to t_3) \to t_8 \ \underline{\perp}\ \omega \to t_0 \to t_0 \ \ [T]$$

We should not “erase” the argument, since it must be typable ! Updated system:

$$\left\{ \begin{array}{llll} t_6^2 \to t_3 & \underline{\perp} & t_6^1 & [t_4^2, t_5^2, t_6^2], \\ t_5^1 \to t_6^1 & \underline{\perp} & t_4^1 & [t_5^1], \\ t_5^2 \to t_6^2 & \underline{\perp} & t_4^2 & [t_5^2] \end{array} \right\}$$

Those equations correspond to the terms:

$$I \ \text{ et } \ \lambda u\ (uu)(uu)$$

and not I alone

$\Lambda_{\mathcal{K}}$-calculus

- Inspired by Klop, 1980.
- Syntax:

$$M, N ::= x \mid M\,N \mid \lambda x M \mid [M, N]$$

- Semantics:

For $x \in fv(M)$:

$$[\lambda x M, N_1, \ldots, N_n]\, N \longrightarrow_{\mathcal{K}} [M\{x \mapsto N\}, N_1, \ldots, N_n]$$

For $x \notin fv(M)$:

$$[\lambda x M, N_1, \ldots, N_n]\, N \longrightarrow_{\mathcal{K}} [M, N_1, \ldots, N_n, N]$$

$\Lambda_{\mathcal{K}}$-calculus

- $\mathcal{WN}_{\mathcal{K}} = \mathcal{SN}_{\mathcal{K}}$: normalising terms are strongly normalising
- $\mathcal{SN}_{\Lambda} = \Lambda \cap \mathcal{SN}_{\mathcal{K}}$: they correspond to strongly normalising terms in λ-calculus
- We add the typing rule:

$$\frac{\Gamma_1 \vdash M_1 : \tau \qquad \Gamma_2 \vdash M_2 : \sigma}{\Gamma_1, \Gamma_2 \vdash [M_1, M_2] : \tau}\text{(Typ Forget)}$$

Reduction rules

System state: $(\mathcal{E}, \Pi)$ where

- $\mathcal{E}$ is a set of constraints
- Π is a proof skeleton, that will evolve to a valid typing tree

Reduction rules

System state: $(\mathcal{E}, \Pi)$ where

- $\mathcal{E}$ is a set of constraints
- Π is a proof skeleton, that will evolve to a valid typing tree

Rule for $n \geq 1$:

$$\boxed{\begin{array}{c} (\{\tau \to t \perp t_1, \ldots, t_n \to \sigma \ [T]\} \cup \mathcal{E},\ \Pi) \ \longrightarrow\ (S(\mathcal{E}),\ S(\Pi)) \\ \text{with } S = \{t_i \mapsto \langle\tau\rangle^i, \langle T\rangle^i\}_{1\leq i\leq n} :: \{t \mapsto \sigma, \emptyset\} :: D(n, T) \end{array}}$$

$$(R_n)$$

Reduction rules

Rule for $n = 0$:

$$\boxed{\begin{array}{c}(\{\tau \to t \;\underline{\perp}\; \omega \to \sigma \;\; [T]\} \cup \mathcal{E},\ \Pi) \;\longrightarrow\; (S(\mathcal{E}),\ S(\Pi)) \\ \text{with } S = \{t \mapsto \sigma, \emptyset\}\end{array}}$$

$$(R_0)$$

Final rule:

$$\boxed{(\{\tau \;\underline{\perp}\; t\} \cup \mathcal{E},\ \Pi) \;\longrightarrow_f\; (S(\mathcal{E}),\ S(\Pi)) \quad \text{with } S = \{t \mapsto \tau\}}$$

$$(R_f)$$

INRIA

Results

- Theorem: A term M is typable if and only if the initial system corresponding to M converges.

Results

- Theorem: A term M is typable if and only if the initial system corresponding to M converges.
- Theorem: If M is typable, then the final proof skeleton is a valid typing tree for M.

Results

- Theorem: A term M is typable if and only if the initial system corresponding to M converges.
- Theorem: If M is typable, then the final proof skeleton is a valid typing tree for M.
- Theorem: This typing tree is *principal*.

Results

- Theorem: A term M is typable if and only if the initial system corresponding to M converges.
- Theorem: If M is typable, then the final proof skeleton is a valid typing tree for M.
- Theorem: This typing tree is *principal*.
- Rank: Syntactic definition on types; to evaluate the "level" of polymorphism.

Results

- Theorem: A term M is typable if and only if the initial system corresponding to M converges.
- Theorem: If M is typable, then the final proof skeleton is a valid typing tree for M.
- Theorem: This typing tree is *principal*.
- Rank: Syntactic definition on types; to evaluate the “level” of polymorphism.

  Property: The finite-rank algorithm *always stops*.
  Consequence: Finite-rank inference is *decidable*.

Other results

- Implementation of the algorithm: TYPI

  `http://www-sop.inria.fr/mimosa/Pascal.Zimmer/typi.html`

- Variant: by replacing the rule (R_0) with the general rule (R_n); equivalent to the type system $\mathcal{D}\Omega$, with the rule: $$\frac{}{\vdash M : \omega}\,(\text{Typ } \omega)$$

- Extension to references (introducing conjunction only for values, as in ML; less liberty on the order of resolution)

- Extension to recursion $\mu x M$ (additional unification at the end of the algorithm)

in order to type MLOBJ ...

INRIA

Future

- integrate intersection types in the language MLOBJ
- polymorphic methods in MLOBJ
- study the expressivity of mixins more closely
- extend the language with other paradigms

The end

TYPI

`http://www-sop.inria.fr/mimosa/Pascal.Zimmer/typi.html`

Direct implementation of the algorithm...

Rank

$$inc(0) = 0$$
$$inc(n) = n + 1 \quad \text{for } n > 0$$

$$rank(t) = 0$$
$$rank(\tau \rightarrow \sigma) = \max(inc(rank(\tau)), rank(\sigma))$$
$$rank(\tau_1, \ldots, \tau_n \rightarrow \sigma) =$$
$$\max(inc(\max(1, rank(\tau_1), \ldots, rank(\tau_n))), rank(\sigma))$$
$$\text{for } n \neq 1$$

Rank

Syntactic definition on types...

- rank 0: usual types without intersection
- rank 1: empty
- rank $r \geq 2$: there is a non-trivial conjunction under $r - 1$ arrows
  Example:
  $(t_1 \rightarrow t_2), (\omega \rightarrow t_3) \rightarrow t_1 \rightarrow t_3$ has rank 3

Finite-rank algorithm

- Choose a maximal allowed rank r.
- For every intermediate step $(\mathcal{E}, \Pi)$, check that $rank(\Pi) \leq r$.
- Otherwise, the term is not typable at rank r.

INRIA

Finite-rank algorithm

- Choose a maximal allowed rank r.
- For every intermediate step $(\mathcal{E}, \Pi)$, check that $rank(\Pi) \leq r$.
- Otherwise, the term is not typable at rank r.

Property: The finite-rank algorithm *always stops*.

Consequence: Finite-rank inference is *decidable*.

Variant

What happens if we use the general rule also for $n = 0$?

$$\boxed{\begin{array}{c}(\{\tau \to t \perp\!\!\!\!\!- \ t_1, \ldots, t_n \to \sigma \ [T]\} \cup \mathcal{E},\ \Pi) \ \longrightarrow \ (S(\mathcal{E}),\ S(\Pi)) \\ \text{with } S = \{t_i \mapsto \langle\tau\rangle^i, \langle T\rangle^i\}_{1 \le i \le n} :: \{t \mapsto \sigma, \emptyset\} :: D(n, T)\end{array}}$$

$$(R_n)$$

- Leads to "erase" constraints or sub-trees by $D(0, T)$
- Correspondence with the type system $\mathcal{D}\Omega$ (Krivine) or $\lambda\cap$ (Barendregt)

$$\frac{}{\vdash M : \omega}\,(\text{Typ } \omega)$$

Variant

- Property: The variant of the algorithm converges iff the term is normalising.
- Proposition: A term is typable in $\mathcal{D}\Omega$ with a non-trivial type iff it has a head-normal form.
- Caracterisation of normalising terms.
- Corollary: If the algorithm converges, then the term is typable.
- Reciprocal property: not true (example: $x\Omega$)

System $\mathbb{I}$

- System proposed by Kfoury and Wells (variant: System E with Carlier)
- Types contain *expansion variables*:

$$
\begin{array}{rcl}
\psi & ::= & \alpha \mid (\psi \rightarrow \psi) \\
\psi & ::= & \psi \mid (\psi \wedge \psi') \mid (F\psi)
\end{array}
$$

- Algorithm for solving similar constraints and returning a typing tree

System $\mathbb{I}$

- Correspondence expansion variables / territory:

$$F_T \quad \longleftrightarrow \quad T = \{v \mid F_T \in \text{E-path}(v, \Gamma_{\mathbb{I}}(M))\}$$

- Both algorithms perform the same operations, not necessarily in the same order, if we ignore expansion variables
  $\longrightarrow$ operational correspondence
- Used to avoid redoing the proofs of some results (principality, finite rank)

References

The expression

$$(\lambda r\ (r := [\texttt{"chaîne"}]\ ;\ \texttt{hd}(!\,r) + 1))\ (\texttt{ref}\ [\,])$$

is typable, but its execution leads to an error...

References

The expression

$$(\lambda r\ (r := [\texttt{"chaîne"}]\ ;\ \texttt{hd}(!\,r) + 1))\ (\texttt{ref}\ [\,])$$

is typable, but its execution leads to an error...

Solution similar to the one for polymorphism in ML: introducing conjunction only for *values* (Davies and Pfenning).

$$\frac{\Gamma \vdash V : A \qquad \Gamma \vdash V : B}{\Gamma \vdash V : A \wedge B}$$

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M N : B}$$

INRIA

References

- Distinguish the types of terms-variables and applications: t_v and $t_@$
- Extended syntax for types:

$$t_b ::= t_v \mid t_b\, ref \mid cte \mid t_b\, list$$

$$\tau, \sigma ::= t_v \mid \tau\, ref \mid cte \mid \tau\, list \mid t_@ \mid t_b, \ldots, t_b \rightarrow \tau$$

- Decomposible equations:

$$\tau \rightarrow t_@ \perp t_{b_1}, \ldots, t_{b_n} \rightarrow \sigma \;\; [T]$$

References

$$(\{\tau \to t_{@} \stackrel{\perp}{\longrightarrow} t_{b_1}, \ldots, t_{b_n} \to \sigma \;\; [T]\} \cup \mathcal{E},\ \Pi) \;\longrightarrow\; (S(\mathcal{E}),\ S(\Pi))$$

$$\text{with } S = \begin{cases} mgu(t_{b_i}, \langle\tau\rangle^i, \langle T\rangle^i)_{1 \le i \le n} :: \{t_{@} \mapsto \sigma, \emptyset\} :: D(n, T) & \text{if } ValueType(\tau) \\ mgu(t_{b_i}, \tau, T)_{1 \le i \le n} :: \{t_{@} \mapsto \sigma, \emptyset\} & \text{otherwise} \end{cases}$$

References

$$(\{\tau \rightarrow t_@ \stackrel{\perp}{\longrightarrow} t_{b_1}, \ldots, t_{b_n} \rightarrow \sigma \ [T]\} \cup \mathcal{E},\ \Pi) \longrightarrow (S(\mathcal{E}),\ S(\Pi))$$

$$\text{with } S = \begin{cases} mgu(t_{b_i}, \langle\tau\rangle^i, \langle T\rangle^i)_{1\leq i\leq n} :: \{t_@ \mapsto \sigma, \emptyset\} :: D(n,T) & \text{if } ValueType(\tau) \\ mgu(t_{b_i}, \tau, T)_{1\leq i\leq n} :: \{t_@ \mapsto \sigma, \emptyset\} & \text{otherwise} \end{cases}$$

but we also need to impose an order for solving the constraints, corresponding more or less to call-by-value...

Recursion

- We add an operator $\mu x M$
- Solution: infer types as for M, then additional unification algorithm
- Modify the type system:

$$\frac{\Gamma, x : \sigma_1, \ldots, x : \sigma_n \vdash M : \tau}{\Gamma \vdash \mu x\, M : \tau}\text{(REC)} \quad \text{with } \forall i\ \sigma_i \equiv \tau$$

- Equality modulo commutativity and contraction:

$$\ldots, \tau_1, \tau_2, \ldots \longrightarrow \sigma \quad \equiv \quad \ldots, \tau_2, \tau_1, \ldots \longrightarrow \sigma$$

$$\ldots, \tau, \tau, \ldots \longrightarrow \sigma \quad \equiv \quad \ldots, \tau, \ldots \longrightarrow \sigma$$