Logic Programming with Names and Binding

James Cheney, Cornell University

(joint work with Christian Urban, University of Cambridge)

September 7, 2004

Problem

- Names can't be handled easily in traditional logic programming.
- Consider the typing rules for the λ-calculus:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e : \tau' \rightarrow \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau} \qquad \frac{\Gamma, x : \tau \vdash \tau'}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}$$

- Here's an easy (but incorrect) Prolog translation:

$$\begin{array}{lll} tc(G, var(X), T) & :- & mem((X, T), G). \\ tc(G, app(E, E'), T) & :- & tc(G, E, arr(T', T)), tc(G, E', T'). \\ tc(G, lam(X, E), arr(T, T')) & :- & tc([(X, T)|G], E, T'). \end{array}$$

Some bugs

- This program is incorrect. For example,

$$lam(x, lam(x, var(x)))$$

  can be assigned two types $\alpha \rightarrow \beta \rightarrow \beta$, $\alpha \rightarrow \beta \rightarrow \alpha$.

- Also,

$$lam(x, lam(x, app(var(x), var(x))))$$

  may succeed with answer $\alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$!

- What happened?

I lied

- Technically speaking, I should have said

$$\frac{\Gamma, x : \tau \vdash \tau' \qquad x \notin FV(\Gamma)}{\Gamma \vdash \lambda x.e : \tau \to \tau'}$$

- It's not hard to define not_in_fv in Prolog, giving

  $tc(G, lam(X, E), arr(T, U)) :- not\_in\_fv(X, G), tc([(X, T)|G], E, U).$

- But this rules out both $\lambda x.\lambda x.x$ and $\lambda x.\lambda x.(x\ x)$.

- We need to freshen bound variables. Back to the drawing board?

Revisionist history

- If you know about higher-order abstract syntax (and I assume many of you do), please pretend you don't for the next few minutes.

- What if HOAS had *never been invented*?

- Perhaps someone would have invented an alternative way of thinking about binding.

- Recently someone did.

Gabbay-Pitts theory

- Recently Gabbay and Pitts (LICS 1999) developed a new theory of names and binding.

- Names are an *abstract data type* with constants $a, b, \ldots$

- $(a\ b) \cdot t$: the result of swapping names a and b in t.

- $a \# t$: *freshness* predicate asserting "a is fresh for t"

- $\langle a \rangle t$: the *abstraction* of a over t, i.e. t with a bound.

Examples

- Swapping:

$$(a\ b)\cdot(f(a,b,c)) \approx f(b,a,c) \qquad (a\ b)\cdot\langle a\rangle f(b,a) \approx \langle b\rangle f(a,b)$$

- Freshness ("not-free-in")

$$\neg(a \# a) \qquad a \# f(b,c) \qquad a \# \langle a\rangle t$$

- Abstraction (equal up to "α-equivalence"):

$$\langle a\rangle a \approx \langle b\rangle b = \cdots \qquad \langle a\rangle f(a,b) \approx \langle a'\rangle f(a',b) \not\approx \langle b\rangle f(b,a)$$

αProlog

- αProlog is a logic programming language with built-in freshness, swapping, and abstraction.

- In αProlog, the tricky λ-typechecking rule can be written as:

$$tc(G, lam(\langle x\rangle E), arr(T, U)) \mathrel{:-} x \mathrel{\#} G, tc([(x, T)|G], E, U).$$

- We use $x \mathrel{\#} G$ for $x \notin FV(\Gamma)$.

- We use abstraction for binding in $lam(\langle x\rangle E)$.

Nominal unification (1)

- *Nominal unification* algorithm developed by Urban, Pitts, Gabbay solves such problems.
- Answers are substitutions θ paired with sets of freshness constraints $F = \{x \mathrel{\#} V, \ldots\}$.
- Example: $\langle x\rangle Z$ and $\langle y\rangle W$ unify with $\theta = [W := (x\ \ y) \cdot Z]$, $F = \{x \mathrel{\#} Z\}$.
- Example: $a \mathrel{\#} f(X, \langle a\rangle Y)$ provided $F = \{a \mathrel{\#} X\}$.

Nominal unification (2)

- Based on first-order unification.

- Key new rules for unifying abstractions:

$$\begin{aligned}\langle a\rangle t \approx ?\ \langle a\rangle u, P &\implies t \approx ?\ u, P\\ \langle a\rangle t \approx ?\ \langle b\rangle u, P &\implies t \approx ?\ (a\ b)\cdot u, a \mathrel{\#?} u, P\end{aligned}$$

- Some new rules for freshness problems:

$$\begin{aligned}a \mathrel{\#?} b, P &\implies P\\ a \mathrel{\#?} \langle a\rangle t, P &\implies P\\ a \mathrel{\#?} \langle b\rangle t, P &\implies a \mathrel{\#?} t, P\end{aligned}$$

Proof search in αProlog

- As in Prolog, we solve a goal formula A' by looking for a clause

$$A :- G_1, \ldots, G_m$$

  unifying A and A' to θ, and solving subgoals $\theta(G_1), \ldots, \theta(G_m)$.

- In Prolog, clause variables are freshened and first-order unification is used.

- In αProlog, clause variables and names are freshened and nominal unification is used.

Examples

- Consider $tc([], lam(\langle x\rangle lam(\langle x\rangle var(x))), T)$
- Unique solution: $T = arr(T_1, arr(T_2, T_2))$.
- Consider $tc([], lam(\langle x\rangle lam(\langle x\rangle app(var(x), var(x)))), T)$
- No solution

So what?

- Can other interesting programs be written more easily in αProlog?

- Yes. Substitution, evaluation, etc.

- These can be done using HOAS also.

- What can αProlog do that HOAS cannot?

Closure conversion (not in paper)

- Important FP compilation phase
- Idea: Make all functions *closed*
- Translation function to (closed function, environment) pair
- Environment shape depends on context:

$$\Gamma = x_n, \ldots, x_1 \mapsto env = \langle v_n, \langle v_{n-1}, \ldots, v_1 \rangle \cdots \rangle$$

Simple, untyped closure conversion

- A simple approach:

$$\begin{array}{rcl}
C[\![x, \Gamma \vdash x]\!]e & = & \pi_1(e) \\
C[\![y, \Gamma \vdash x]\!]e & = & C[\![\Gamma \vdash x]\!](\pi_2(e)) \qquad (x \neq y) \\
C[\![\Gamma \vdash t_1 t_2]\!]e & = & let\ c\ =\ C[\![\Gamma \vdash t_1]\!]e\ in\ (\pi_1(c))\ \langle C[\![\Gamma \vdash t_2]\!]e, \pi_2(c)\rangle \\
C[\![\Gamma \vdash \lambda x.t]\!]e & = & \langle \lambda y.C[\![x, \Gamma \vdash t]\!]y, e\rangle \quad (x, y \notin \Gamma)
\end{array}$$

Closure conversion code

- This is no problem for αProlog.

$$
\begin{array}{l}
var : id \to tm. \quad app : (tm, tm) \to tm. \\
lam : \langle id \rangle tm \to tm. \quad let : (tm, \langle id \rangle tm) \to tm. \\
\vdots \\
cconv([X|G], var(X), E, pi_1(E)). \\
cconv([X|G], var(Y), E, T) \\
\qquad :- \; X \# Y, cconv(G, var(Y), pi_2(E), T). \\
cconv(G, app(T_1, T_2), E, T') \\
\qquad :- \; cconv(G, T_1, E, T_1'), \\
\qquad\quad\; cconv(G, T_2, E, T_2'), \\
\qquad\quad\; T' = let(T_1', \langle c \rangle app(pi_1(var(c)), pair(T_2', pi_2(var(c))))). \\
cconv(G, lam(\langle x \rangle T), E, pair(lam(\langle y \rangle F), E)) \\
\qquad :- \; x \# G, y \# G, cconv([x|G], T, var(y), F).
\end{array}
$$

Closure conversion appears hard in HOAS

- Need access to context Γ as data structure: order of name binding matters!

- Need name-(in)equality tests

- HOAS doesn't help with this

More examples in paper

- $\lambda\mu$-calculus with “continuation call” substitution operation

- π-calculus terms and operational semantics

Problem: Equivariance

- *Equivariance*: Validity preserved by name-swapping.

- For example,

$$tc([], lam(\langle x\rangle lam(\langle y\rangle var(x))), \tau) \overset{(a\ b)}{\Longleftrightarrow} tc([], lam(\langle y\rangle lam(\langle x\rangle var(y))), \tau)$$

- Logical *equivalence* does not imply nominal term *equality*.

- Nominal unification does not (and should not) allow this

Equivariant Unification is NP-Complete

- **Graph 3-Colorability** reduces to equivariant unification.
- So, EV-unification is hard in general.
- Is it hard in practice? Open problem.
- Nominal unification works in many cases.
- Future/current work: practical EV-unification.

Summary

- Gabbay-Pitts + logic programming provides advanced facilities for programming with names and binding

- There are significant technical challenges in this approach.

- But, may be easier to automate or assist reasoning about languages using GP approach.

- αProlog: A reasonable first step in this direction

Church...

- It's true that HOAS has a long and honorable history.
- The idea dates at least to Church (1940).
- It is powerful and broadly applicable.
- But *not* all-powerful.

… or Frege?

- But the permutation approach has a longer (but less well-known) history!

- Frege (1879) said

> Replacing a German letter [bound name] everywhere in its scope by some other one is, of course, permitted, so long as in places where different letters initially stood different ones also stand afterward. This has no effect on the content.