Incomplete λ-terms

Murdoch J. Gabbay

February 11, 2004

First-order predicate logic with equality

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \qquad \frac{\Gamma, P, Q \vdash C}{\Gamma, P \wedge Q \vdash C} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \qquad \frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}$$

$$\frac{\Gamma, P \vdash C \quad \Gamma, Q \vdash C}{\Gamma, P \vee Q \vdash C} \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} \qquad \frac{\Gamma \vdash P \quad \Gamma, Q \vdash C}{\Gamma, P \Rightarrow Q \vdash C}$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash \forall x.\, P} \qquad \frac{\Gamma, P[x \mapsto t] \vdash C}{\Gamma, \forall x.\, P \vdash C} \qquad \frac{\Gamma \vdash P[x \mapsto t]}{\Gamma \vdash \exists x.\, P} \qquad \frac{\Gamma, P \vdash C}{\Gamma, \exists x.\, P \vdash C}$$

$$\frac{\Gamma \vdash P \quad \Gamma, P \vdash Q}{\Gamma \vdash Q} \qquad \frac{\Gamma, P, P \vdash C}{\Gamma, P \vdash C} \qquad \Gamma, \bot \vdash C \qquad \Gamma \vdash \top$$

$$\frac{\Gamma, t = t \vdash C}{\Gamma \vdash C} \qquad \frac{\Gamma, t' = t, P[x \mapsto t'] \vdash C}{\Gamma, t' = t, P[x \mapsto t] \vdash C} \qquad \Gamma, P \vdash P$$

Encoding

We want a λ-term for proof-search in First-Order Logic.

Introduce constants $\top, \bot, \wedge, \vee, \Rightarrow, \neg, \forall$, and $\exists$, 'predicate constant symbols' p, q, r, 'equality' $=$, and 'term-formers' c. Also introduce a comma , 'entailment' $\vdash$ and pairing $\langle \text{-}, \text{-} \rangle$.

We are spoilt for choice in our encoding:

$$\exists \langle a, P_a \rangle \qquad \exists \lambda a.P_a$$

We could also use FM abstractions, but then we'd have to introduce them into the λ-calculus.

Use $\exists \lambda a.P_a$ and $\forall \lambda a.P_a$. Discuss why later (perhaps).

Our goal

A λ-term which when applied to $\Gamma \vdash P$ returns 'yes' just if a proof exists. We could also exhibit a λ-term which constructs a proof, using meta-variables, but that involves lists of constraints. Existential variables are simpler and test the same principles, see below.

$\forall$-left and $\exists$-right rules have t, deadly for proof-search. Instead use an existential variable X.

$$\frac{\Gamma \vdash P[x \mapsto X]}{\Gamma \vdash \exists x.\, P}$$

At equalities $t = X$ a unification algorithm calculates instantiations $[X \mapsto t]$; these are passed around and applied to remaining goals. Substitution, as we shall see, is not capture-avoiding, because X may occur under the scope of $\forall$ and $\exists$ binders.

λ-calculus with meta-variables

Hypothesise an infinite collection of disjoint countably infinite **sets of atoms** $a_i \in \mathbb{A}_i$. These represent variable symbols in our calculus.

The syntax is:

$$t ::= a_i \mid tt \mid \lambda a_i.t \mid t[a_i \mapsto t] \mid И a_i.\, t$$

None of these constructors bind. For example, $\lambda a_0.a_0 \not\equiv \lambda b_0.b_0$.

λ-calculus with meta-variables

$$
\begin{array}{rrll}
(\beta) & & (\lambda a_i.t)v & \to t[a_i \mapsto v] \\
(\sigma a_i) & & a_i[a_i \mapsto u] & \to u \\
(\sigma x_j) & x_j \# a_i \vdash & x_j[a_i \mapsto u] & \to x_j \qquad j \leq i \\
(\sigma p) & & (st)[a_i \mapsto u] & \to s[a_i \mapsto u]t[a_i \mapsto u] \\
(\sigma \lambda) & n_j \# a_i, u \vdash & (\lambda n_j.t)[a_i \mapsto u] & \to \lambda n_j.(t[a_i \mapsto u]) \qquad j \geq i \\
(\sigma \lambda') & & (\lambda n_j.t)[a_i \mapsto u] & \to \lambda n_j.(t[a_i \mapsto u]) \qquad j < i \\
(\sigma \sigma) & a_i \# v, b_j \vdash & t[a_i \mapsto u][b_j \mapsto v] & \to t[b_j \mapsto v][a_i \mapsto u[b_j \mapsto v]] \qquad j \leq i \\
(\sigma \sigma') & & t[a_i \mapsto u][b_j \mapsto v] & \to t[b_j \mapsto v][a_i \mapsto u[b_j \mapsto v]] \qquad j > i
\end{array}
$$

λ-calculus with meta-variables

$$
\begin{array}{lrll}
(И p) & n_i\#t' & \vdash (Иn_i.\ t)t' & \rightarrow Иn_i.\ (tt') \\
(И\lambda) & n_j\#a_i & \vdash \lambda a_i.Иn_j.\ t' & \rightarrow Иn_j.\ \lambda a_i.t \\
(И\sigma) & n_j\#a_i, u & \vdash (Иn_j.\ t)[a_i \mapsto u] & \rightarrow Иn_j.\ (t[a_i \mapsto u]) \\
(\alpha И) & b_i\#t & \vdash Иa_i.\ t & \rightarrow Иb_i.\ (b_i\ a_i)t
\end{array}
$$

Swapping $(b_i\ a_i)t$ is a Nominal Rewriting primitive, it is that term obtained by replacing every b_i by a_i, and vice-versa, in t (see discussion in final slices).

Examples

Write a, b, c for variables of level 0, X, Y, Z for variables of level 1, and s, t, r for variables of level 2. t for a_2 clashes with the notation t for unknown terms in the Nominal Rewriting framework; oh well.

$(И a.\ \lambda a.ab)[b \mapsto a] \to (И c.\ \lambda c.cb)[b \mapsto a] \to И c.\ \lambda c.ca$.
$(\lambda a.ab)[b \mapsto a] \to$.
$(\lambda a.aX)[X \mapsto a] \to \lambda a.aa$.
$(И a.\ \lambda a.aX)[X \mapsto a] \to (И c.\ \lambda c.cX)[X \mapsto a] \to И c.\ \lambda c.ca$.

$\lambda a.(a[a \mapsto b]) \to \lambda a.b$.
$\lambda a.(a[X \mapsto b]) \to \lambda a.a$.
$\lambda X.(X[X \mapsto b]) \to \lambda X.b$.

More examples

$\lambda t.(t[X \mapsto b])\rightarrow$.
$(\lambda t.(t[X \mapsto b]))(XY)\rightarrow(XY)[X \mapsto b]\rightarrow bY$.

Now we can write some cases in the proof-search term. Give the calculus pattern-matching $\Longrightarrow$. Then:

$\Gamma \vdash \forall f \Longrightarrow И a.\, \Gamma \vdash fa$.
$\Gamma \vdash \exists f \Longrightarrow И X.\, \Gamma \vdash fX$.

We should write the unification algorithm too.

Related work (i.e. things I knew about before or during making the calculus)

Masahiko Sato, Takafumi Sakurai, Yukiyoshi Kameyama, and Atsushi Igarashi, “Calculi of Meta-variables” Computer Science Logic and 8th Kurt Gödel Colloquium (CSL’03 & KGC), Vienna, Austria. Proceedings. A λ-calculus (two actually) using ‘levels of meta-variables’—no explicit substitutions, with primitive α-equivalence (no $\mathit{И}/(a\ b)$ to control names), and instead of $\#$ conditions a notion of ‘blocked’ transition in the case of name clash. We have Nominal Rewriting.

Sylvain Baro and François Maurel, “The $q\nu$ and $q\nu K$ calculi: name capture and control”. Similar to the level 0 transposition- and explicit substitution-free fragment of the calculus presented here?

Related work

Gueorgui Jojgov, “Tactics and Parameters”, ENCTS vol 85 issue 7. We might implement this using our calculus.

Masatomo Hashimoto and Atsushi Ohori, “A Typed Context Calculus”. A level 0,1 calculus. No explicit substitutions, transpositions, or $И$. α-equivalence is something unorthodox taking account of the scope into which variables *may* be substituted, as well as the scope of the binding λ. We use $И$ to make scope explicit.

Cézar Muñoz, “Un Calcul de Substitutions. . . ”, PhD Thesis, Paris VII. A program which we should be able to implement.

Applications of this kind of work

The Japanese: Linking. Dynamic linking. Modules (as first-class objects).

The French and the Russians: Tactic (programming) languages, especially for theorem provers. Dependently typed calculus.

The British: Pitts' and others work on operational techniques for contextual equivalence. E.g. Pitts "Operational Reasoning for Functions on Local State". Can these works and others be generalised to a contextual equivalence result for data values in this calculus? If not, what does such a result mean? Can meta-variables represent macros? What does an "ML with holes" look like?

Fragments of the calculus

Also, various fragments of the calculus have interest. For example without levels we have an explicit substitution calculus with explicit α-equivalence. Without $\mathcal{N}$ (and with α-equivalence using Nominal Renaming FM abstractions) we obtain fresh explicit substitution calculi.

Nominal Signatures

Fernández, Gabbay, Mackie, "Nominal Rewriting". Critical pair theorem and definitions of the system we used above. Fix $\mathcal{S}$ **base data sorts** typically called s, for example integer, boolean.

A **Nominal Signature** Σ is:

1. A set of **sorts of atoms** typically written ν.
2. A set of **data sorts** typically written δ. δ can be a base data sort or a product:

   (1) $$\delta ::= s \mid \delta \times \delta.$$

3. **Compound (data) sorts** typically written τ are then *defined* by the following grammar:

   (2) $$\tau ::= \nu \mid \delta \mid 1 \mid \tau \times \tau \mid [\nu]\tau.$$

Terms

Fix Σ. For each τ fix $\mathcal{X}_\tau$ term variables X_τ, Y_τ, Z_τ. They will represent meta-level unknowns. For each ν fix $\mathcal{A}_\nu$ atoms $a_\nu, b_\nu, c_\nu, f_\nu, g_\nu, h_\nu, \ldots$. We shall drop the subscripts.

A **swapping** is a pair $(a_\nu\ b_\nu)$. **Permutations** π are lists of swappings, write **Id** for the empty list. Define:

$$(a\ b)(a) \stackrel{\text{def}}{=} b \qquad (a\ b)(b) = a \quad \text{and} \quad (a\ b)(c) = c\ (c \neq a, b)$$

$$ds(\pi, \pi', \stackrel{\text{def}}{=}) \{n \mid \pi(n) \neq \pi'(n)\}.$$

For example $ds((a\ b), \mathbf{Id}, =)\{a, b\}$.

Nominal Terms are generated by the following grammar:

$$t ::= a_\nu \mid \pi \cdot X_\tau \mid *_1 \mid \langle t_\tau, t'_{\tau'} \rangle_{\tau \times \tau'} \mid ([a_\nu]t_\tau)_{[\nu]\tau} \mid (f_{\tau \to \delta} t_\tau)_\delta$$

Terms

However we do not need abstractions to express our λ-calculus, so we excise them!

Nominal Terms *without abstractions* are generated by the following grammar:

$$t ::= a \mid \pi \cdot X \mid *_1 \mid \langle t, t' \rangle \mid ft$$

Write $T(\Sigma, \mathcal{A}, \mathcal{X})$ for the set of these terms over a signature Σ.

Terms

We write X for $\mathbf{Id} \cdot X$. The intuition of $(a\ b) \cdot X$ is "swap a and b in the syntax of X, when it is instantiated". These X are *not* the 'meta-variables' of the calculus which we shall define in a moment.

We shall write $(a\ b) \cdot t$ for $t \not\equiv X$. This is sugar:

$$(a\ b) \cdot n = (a\ b)(n) \quad (a\ b) \cdot ft = f(a\ b) \cdot t \quad (a\ b) \cdot * = * \qquad (3)$$
$$(a\ b) \cdot \langle s, t\rangle = \langle (a\ b) \cdot s, (a\ b) \cdot t\rangle$$

An apartness condition is a pair $a\#X$. Apartness contexts $\Delta, \nabla, \Gamma, \ldots$ are finite sets of apartness conditions. $a\#X$ means "a does not occur in X, when it is instantiated".

Rewrite rules

Write $V(s)$ and $V(\nabla)$ for 'variables of'.

A nominal rewrite rule is $\nabla \vdash l \to r$, such that $V(r) \cup V(\nabla) \subseteq V(l)$.
If $\nabla = \emptyset$ we may write $l \to r$.

We take rules up to permutative renaming. Thus

$$a\#X \vdash (\lambda a.X)Y \to X \quad \text{and} \quad a\#Y \vdash (\lambda a.Y)X \to Y,$$
$$a\#X \vdash X \to \lambda a.(Xa) \quad \text{and} \quad b\#X \vdash X \to \lambda b.(Xb)$$

are 'morally' the same, where $\lambda : \nu \times \tau \to \tau$ and application $\tau \times \tau \to \tau$ are constructors in Σ.

Formalise morality: a set of rewrite rules $\mathcal{S}$ is equivariant when if $R \in \mathcal{S}$ then $R' \in \mathcal{S}$ for all permutative renamings of variable symbols and atoms.

A **nominal rewrite system** $(\Sigma, \mathcal{R})$ consists of: a nominal signature Σ, and an equivariant set $\mathcal{R}$ of nominal rewrite rules over Σ.

Rewrite rules

1. $a\#X \vdash (\lambda a.X)Y \rightarrow X$ is a form of trivial β-reduction.
2. $a\#X \vdash X \rightarrow \lambda a.(Xa)$ is η-expansion.
3. Of course a rewrite rule may define any arbitrary transformation of terms, and may have an empty context, for example $\emptyset \vdash XY \rightarrow XX$.
4. $a\#Z \vdash X\lambda a.Y \rightarrow X$ is not a rewrite rule, because $Z \notin V(X\lambda a.Y)$. $\emptyset \vdash X \rightarrow Y$ is also not a rewrite rule.
5. $\emptyset \vdash a \rightarrow b$ *is* a rewrite rule.

OK, done!

Without abstractions, we have a normal first-order rewriting system *enriched with*: swappings $(a\ b) \cdot X$, a well-behavedness restriction of equivariance (which we generally kind of assume anyway), and freshness assumptions $a\#X$.

We are still manipulating concrete syntax trees using relatively standard notions of matching and unification.