Type inference, proof search, and unification

Dale Miller
Penn State University
In September:
INRIA Futurs/Saclay and Ecole Polytechnique

Outline

1. Specification of type inference
2. Encoding type inference into proof search
3. HOAS and λ-tree syntax
4. Unification of simply typed lambda-terms
5. Pattern unification (L_λ)

Theme

There is a spectrum between predicates and types. In many ways, type systems are designed to deal with some "well understood" predicates. Reasoning about certain predicates might be "simple" and can be factored out of more general logical reasoning.

This spectrum can be exploited to provide *implementations* of a type system: simply map types into logic using predicates.

We explore the general setting of using logic to specify type systems and look at some detail at the role of proof search and unification to provide actual implementations of type inference and type checking.

Much of what is presented here is also the basis for work on checking the correctness of proof systems, in the sense of, say, proof carrying code.

Untyped λ-calculus: A simple functional programming language

We shall consider the untyped λ-calculus as a small functional programming language.

$$M = x \mid (M\ N) \mid \lambda x.M$$

Here, x ranges over variables and M over untyped λ-terms.

Simple typing for this language is often given as follows:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash M : B \to A \qquad\qquad \Gamma \vdash N : B}{\Gamma \vdash (M\ N) : A}$$

$$\frac{x : A, \Gamma \vdash M : B}{\Gamma \vdash (\lambda x.M) : A \to B}\ (*)$$

The proviso $(*)$ reads: x does not have an occurrence in Γ. As it is, this type system does not give a type to, for example, $\lambda x \lambda x.x$. Formally, we need to add another rule, such as,

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M' : A}\ \alpha,$$

where M and M' are α-convertible.

Approaches to α-conversion

Barendregt's convention: Ignore α-conversion unless it matters.

First-order terms with names: variable are represented as names and the λ-binder takes such a name as it's first argument.

de Bruijn's nameless dummies: replace variable occurrence with indexes that are offsets to their binding occurrence. Eg.

$$\lambda x \lambda y (f (\lambda w (g\ x\ y\ w))\ x) \qquad \lambda\lambda(f(\lambda(g\ 3\ 2\ 1))\ 2)$$

Higher-order abstract syntax and λ-tree syntax. Work within a rich logic (such as Church's Simple Theory of Types) that already incorporates notions of λ-binding, α-conversion, and substitution (via β-conversion). Specifications can reuse the features of the "meta-logic."

First-order terms with names

```
kind  var      type.
type  a, b, c  var.

kind fp     type.
type app    fp  -> fp -> fp.
type lam    var -> fp -> fp.
type v      var -> fp.

 lam a (v a)
 lam a (lam b (v b))
 lam a (app (v a) (v a))
 lam a (lam b (lam c (app (app (v a) (v c)) (app (v b) (v c)))))

kind  ty     type.
type  arr    ty -> ty -> ty.
type  i, j   ty.
```

A Prolog implementation

```
kind tyjud    type.
type jud      var -> ty -> tyjud.

type inf      list tyjud -> fp -> ty -> o.
type member   A -> list A -> o.

inf Gamma (app M N) B       :- inf Gamma M (arr A B), inf Gamma N A.
inf Gamma (lam X M) (arr A B) :- inf (jud X A :: Gamma) M B.
inf Gamma (v X)     A       :- member (jud X A) Gamma.

member X (X::L) :- !.
member X (_::L) :- member X L.
```

The good and the bad

What is good about this specification?

1. Unification is immediately and declaratively available.
2. Backtracking can uncover other solutions if there are any.
3. One specification will handle type checking and type inference.

What is bad about this specification?

1. It's not really logic. The member program uses Prolog's cut or pruning operator.
2. Prolog does not implement occur-check: `(lam x (app (v x) (v x)))` gets an illegal "circular type" (fortunately, λProlog implements the occur-check).
3. What is the meaning of variable names and the operator `v`?
4. All the "dynamics" of the typing judgment are handled within the scope of the non-logical constant `inf`. Meta-logic cannot help much.

To illustrate some of these deficiencies, let us plan to be more ambitious than simply getting our type specification to "run".

Call-by-name evaluation

The following two inference rules present the *natural semantic* (aka, *big step semantics*) for call-by-name evaluation.

$$\frac{}{(\lambda x.M) \Downarrow (\lambda x.M)} \qquad \frac{M \Downarrow (\lambda x.R) \qquad R[N/x] \Downarrow V}{(M\ N) \Downarrow V}$$

```
eval (lam X M) (lam X M).
eval (app M N) V :- eval M (lam X R), subst N X R S, eval S V.


subst ... :- ...
new_var_name ... :- ...
free_occur ... :- ...
   ... etc ...
```

Call-by-value is also easy to specify.

$$\frac{}{(\lambda x.M) \Downarrow (\lambda x.M)} \qquad \frac{M \Downarrow (\lambda x.R) \qquad N \Downarrow U \qquad R[U/x] \Downarrow V}{(M\ N) \Downarrow V}$$

Subject reduction theorem

Proposition: If $\vdash P \Downarrow V$ and $\vdash \mathit{inf}\ \mathit{nil}\ P\ T$, then $\vdash \mathit{inf}\ \mathit{nil}\ V\ T$.

(Also known as the type preservation theorem.)

A proof of this using directly the specifications for type inference and evaluation gets rather complicated.

“Substitution lemmas” will be required. But this seems unfortunate since our “meta-logic” could and should know about substitution too. If it did, then meta-theoretic properties of the logic could be of great help.

Let us not embarrass ourselves with trying more to use this particular encoding.

Rather, let us try to find a more logical specification.

Higher-order abstract syntax

Pick a logic, like Church's Simple Theory of Types (or a dependently typed λ-calculus) that include notions of λ-binding, α-conversion, and substitution. Encode the object-level binders into that meta-language.

How rich should that meta-level logic be? In particular, what kinds of functions inhabit the type $\tau \to \sigma$?

- Coq, NuPRL, etc: mathematical functions from τ to σ.
- λProlog, Isabelle, Twelf: weak functions; strongly normalizing; based on α, β, and η conversions. Basis for *higher-order abstract syntax.*
- L_λ: weaker functions; substitution restricted to replacing bound variables with other bound variables; based on α, β_0, and η conversions.

$$(\lambda x.B)x = B \qquad (\beta_0)$$

Basis of λ-*tree syntax.*

A more high-level specification

```
kind  ty   type.
type  arr  ty -> ty -> ty.

kind fp      type.
type app     fp  -> fp -> fp.
type lam     (fp -> fp) -> fp.
type inf     fp -> ty -> o.

inf (app M N) B :- inf M (arr A B), inf N A.
inf (lam M) (arr A B) :- pi x\ inf x A => inf (M x) B.
type eval     fp -> fp -> o.

eval (lam R) (lam R).
eval (app M N) V :- eval M (lam R), eval (R N) V.
```

Proof of subject reduction

Let $\vdash$ denotes provability in intuitionistic logic from the above formulas encoding evaluation and typing.

Proposition: If $\vdash P \Downarrow V$ and $\vdash \mathit{inf}\ P\ T$, then $\vdash \mathit{inf}\ V\ T$.

Proof: We prove this theorem by induction on the structure of the derivation of $P \Downarrow V$. Since $P \Downarrow V$ is atomic, its derivation must end with backchaining with one of the formulas encoding evaluation.

Case 1: If the $\Downarrow$ formula for *lam* is used, then P and V are both equal to $\mathit{lam}\ R$, for some R, and the consequent is immediate.

Proof of subject reduction (continued)

Proposition: If $\vdash P \Downarrow V$ and $\vdash \mathit{inf}\, P\ T$, then $\vdash \mathit{inf}\, V\ T$.

Case 2: If $P \Downarrow V$ was derived using the $\Downarrow$ formula for *app*, then P is of the form $(\mathit{app}\ M\ N)$, and for some R there are shorter derivations of $M \Downarrow (\mathit{lam}\ R)$ and $(R\,N) \Downarrow V$. Since P is $(\mathit{app}\ M\ N)$, $\mathit{inf}\, P\ T$ must have been derived using the formula encoding the typing rule for *app*. Hence, there is a U such that $\vdash \mathit{inf}\, M\ (\mathit{arr}\, U\ T)$ and $\vdash \mathit{inf}\, N\ U$. Applying the inductive hypothesis to the evaluation and typing judgments for M, we have $\vdash \mathit{inf}\,(\mathit{lam}\ R)\ (\mathit{arr}\, U\ T)$. This atomic formula must have been derived using the *inf* formula for *lam*, and, hence, $\vdash \forall x(\mathit{inf}\, x\ U \Rightarrow \mathit{inf}\,(R\,x)\ T)$. Since our specification logic is intuitionistic logic, we can instantiate this quantifier with N and use modus ponens (cut) to conclude that $\vdash \mathit{inf}\,(R\,N)\ T$. Applying the inductive hypothesis to the judgments for $(R\,N)$ yields $\vdash \mathit{inf}\, V\ T$.

Observations

Broadly speaking, logic is a study of substitution. If we think of type inference as deduction, then we should pick a meta-logic that captures (object-level) substitutions. Higher-order abstract syntax refers to a style of encoding discipline that makes this possible.

In the "first-order" encoding, the main judgment was atomic: $\mathit{inf}\,\Gamma\ M\ T$. In the "higher-order" encoding, the equivalent judgment is closer to the logical expression $\Gamma \Rightarrow \mathit{inf}\ M\ T$. As a result, the meta-logic has more to contribute to reasoning about type inference. That is, the substitution lemma is not an instance of a meta-level observation.

We pushed the details of substitutions into the meta-logic and we rely on it's meta-theoretic properties and it implementation.

If one has a notion of linear logic typing, there seems to be a natural candidate here for encoding such typing in logic: try using a linear logic programming meta-language.

References

- Gérard Huet, *The Undecidability of Unification in Third Order Logic*, Information and Control 22(1973), 257-267.
- Gérard Huet, *A Unification Algorithm for Typed λ-Calculus*, Theoretical Computer Science 1 (1975), 27-57.
- Warren Goldfarb, *The Undecidability of the Second-Order Unification Problem*, Theoretical Computer Science 13(1981), 225-230.
- Wayne Snyder and Jean H. Gallier, *Higher Order Unification Revisited: Complete Sets of Transformations*, Journal of Symbolic Computation 8(1989) 1-2, 101-140.
- Dale Miller, *A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification*, Journal of Logic and Computation 1(1991)4, 497-536.
- Dale Miller, *Unification under a mixed prefix*, Journal of Symbolic Computation 14(1992) 321-358.