

CSCI 384: Type Inference

The following page provides pseudocode for a type inference engine for MiniML, one that provides a type system with `int`, `bool`, function, product, `list` types, and the "null product" type `unit`. It also has `let`-bounded polymorphism with the addition of type variables in the subterms of a "forall" type. The code defines a function $\text{INFER}(\Gamma, e)$ which takes a variable-type context Γ and a MiniML expression term e and gives a type term for the most general type of e within that context. The type returned is built using the case class type constructors `UNIT`, `INT`, `BOOL`, `ARROW`, `PRODUCT`, and `LIST`, and may also have type variables encoded as `VAR` terms to express polymorphic types, bound with `FORALL`.

```

INFER( $\Gamma, e$ )
1  case  $e$  of
2    ()  $\Rightarrow$ 
3      return UNIT
4  true  $\Rightarrow$ 
5      return BOOL
6   $n$  where  $n \in \mathcal{Z}$   $\Rightarrow$ 
7      return INT
8  nil  $\Rightarrow$ 
9       $\tau := \text{VAR}(\text{FRESH-TYPE-VAR}())$ 
10     return LIST( $\tau$ )
11   $e_1 + e_2 \Rightarrow$ 
12      $\tau_1 := \text{INFER}(\Gamma, e_1)$ 
13      $\tau_2 := \text{INFER}(\Gamma, e_2)$ 
14     UNIFY( $\tau_1, \text{INT}$ )
15     UNIFY( $\tau_2, \text{INT}$ )
16     return INT
17   $e_1 < e_2 \Rightarrow$ 
18      $\tau_1 := \text{INFER}(\Gamma, e_1)$ 
19      $\tau_2 := \text{INFER}(\Gamma, e_2)$ 
20     UNIFY( $\tau_1, \text{INT}$ )
21     UNIFY( $\tau_2, \text{INT}$ )
22     return BOOL
23   $e_1$  andalso  $e_2 \Rightarrow$ 
24      $\tau_1 := \text{INFER}(\Gamma, e_1)$ 
25      $\tau_2 := \text{INFER}(\Gamma, e_2)$ 
26     UNIFY( $\tau_1, \text{BOOL}$ )
27     UNIFY( $\tau_2, \text{BOOL}$ )
28     return BOOL
29  not  $e' \Rightarrow$ 
30      $\tau' := \text{INFER}(\Gamma, e)$ 
31     UNIFY( $\tau', \text{BOOL}$ )
32     return BOOL
33  print  $e' \Rightarrow$ 
34      $\tau' := \text{INFER}(\Gamma, e')$ 
35     return UNIT
36   $e_1 ; e_2 \Rightarrow$ 
37      $\tau_1 := \text{INFER}(\Gamma, e_1)$ 
38     return INFER( $\Gamma, e_2$ )
39  if  $c$  then  $e_1$  else  $e_2 \Rightarrow$ 
40      $\sigma := \text{INFER}(\Gamma, c)$ 
41     UNIFY( $\sigma, \text{BOOL}$ )
42      $\tau_1 := \text{INFER}(\Gamma, e_1)$ 
43      $\tau_2 := \text{INFER}(\Gamma, e_2)$ 
44     UNIFY( $\tau_1, \tau_2$ )
45     return  $\tau_1$ 
46   $e_1 = e_2 \Rightarrow$ 
47      $\tau_1 := \text{INFER}(\Gamma, e_1)$ 
48      $\tau_2 := \text{INFER}(\Gamma, e_2)$ 
49     UNIFY( $\tau_1, \tau_2$ )  $\triangleright$  Not quite right! Want equality types.
50     return BOOL
51  fn  $x \Rightarrow d \Rightarrow$ 
52      $\sigma := \text{VAR}(\text{FRESH-TYPE-VAR}())$ 
53      $\tau := \text{INFER}([x : \sigma] \cdot \Gamma, d)$ 
54     return ARROW( $\sigma, \tau$ )
55   $e_1 \ e_2 \Rightarrow$ 
56      $\tau_1 := \text{INFER}(\Gamma, e_1)$ 
57      $\tau_2 := \text{INFER}(\Gamma, e_2)$ 
58      $\tau := \text{VAR}(\text{FRESH-TYPE-VAR}())$ 
59     UNIFY( $\tau_1, \text{ARROW}(\tau_2, \tau)$ )
60     return  $\tau$ 
61  let val  $x = d$  in  $b$  end  $\Rightarrow$ 
62      $\sigma := \text{INFER}(\Gamma, d)$ 
63      $\sigma' := \text{GENERALIZE}(\sigma)$ 
64      $\tau := \text{INFER}([x : \sigma'] \cdot \Gamma, b)$ 
65     return  $\tau$ 
66  let fun  $f \ x = r$  in  $b$  end  $\Rightarrow$ 
67      $\tau_1 := \text{VAR}(\text{FRESH-TYPE-VAR}())$ 
68      $\tau_2 := \text{VAR}(\text{FRESH-TYPE-VAR}())$ 
69      $\tau := \text{ARROW}(\tau_1, \tau_2)$ 
70      $\tau'_2 := \text{INFER}([x : \tau_1, f : \tau] \cdot \Gamma, d)$ 
71     UNIFY( $\tau_2, \tau'_2$ )
72      $\tau' := \text{GENERALIZE}(\tau)$ 
73      $\sigma := \text{INFER}([f : \tau'] \cdot \Gamma, b)$ 
74     return  $\sigma$ 
75   $x$  where  $x \in \text{Var}$   $\Rightarrow$ 
76     return INSTANTIATE( $\Gamma(x)$ )

```

The code relies on a few functions beyond just recursive calls to `INFER`:

- `FRESH-TYPE-VAR` — provides a new type variable term $\text{VAR}(\alpha)$ for some fresh symbol α . That is, it will produce a type variable term whose name is distinct from any type variable generated so far. Variable type terms are special in that, though initially free, they might get bound to other type terms by the execution of...
- `UNIFY` — takes two type terms, ones that may have type variables, and attempts to make those terms "look the same," possibly by binding any type variables in subterms. The right bottom of the next page gives the pseudocode for `UNIFY`.

The `INFER` code is able to reason correctly about polymorphic types using a strategy of *generalization* and *instantiation* that's prescribed by logicians Hindley and Milner. If you are curious: this code reasons about types

with free variables, giving any `let`-defined name a *forall* type. These `FORALL` type terms serve as a templates that stamp out *instances* of their type when a name gets invoked in a `let` body. This mechanism is supported by the following two functions, and are used when we check the typing of a `let` expression:

- **GENERALIZE** — takes a (possibly) polymorphic term and wraps it with a `FORALLT` type constructor, to distinguish it as a closed type polymorphic in its type variables. For example, when typechecking `fn x => x` we would discover that it is of type $\alpha \rightarrow \alpha$, and **GENERALIZE** would give back the type $\forall \alpha. \alpha \rightarrow \alpha$, that is, `FORALL([α], ARROW(VAR(α), VAR(α))).`
- **INstantiate** — if the given type term is a `FORALL` type term, this builds an equivalent term with fresh free variables, in place of the bound ones. For the identity example just above, this would take the type term $\forall \alpha. \alpha \rightarrow \alpha$ and give back, say, $\alpha_{123} \rightarrow \alpha_{123}$. By building different type term instances for the identity function, an expression like

```
let val id = fn x => x in (id true) andalso ((id 3) > 0) end
```

can be typed correctly. In the first use of `id` we stamp out an instance that unifies as `bool` \rightarrow `bool`. In the second, we stamp out an instance that unifies as `int` \rightarrow `int`.

The pseudocode for **INFER** above does not handle the typechecking related to list and pair types. The code below left continues that code to handle those types.

<pre> 77 e₁ :: e₂ => 78 τ_1 := INFER(Γ, e₁) 79 τ_2 := INFER(Γ, e₂) 80 UNIFY(τ_2, LIST(τ_1)) 81 return τ_2 82 nil => 83 α := VAR(FRESH-TYPE-VAR()) 84 return LIST(α) 85 null e => 86 τ := INFER(e) 87 α := VAR(FRESH-TYPE-VAR()) 88 UNIFY(LIST(α), τ) 89 return BOOL 90 hd e => 91 τ := INFER(e) 92 α := VAR(FRESH-TYPE-VAR()) 93 UNIFY(LIST(α), τ) 94 return α 95 tl e => 96 τ := INFER(e) 97 α := VAR(FRESH-TYPE-VAR()) 98 UNIFY(LIST(α), τ) 99 return τ 100 (e₁, e₂) => 101 τ_1 := INFER(Γ, e₁) 102 τ_2 := INFER(Γ, e₂) 103 return PRODUCT(τ_1, τ_2) 104 fst e => 105 α_1 := VAR(FRESH-TYPE-VAR()) 106 α_2 := VAR(FRESH-TYPE-VAR()) 107 τ := INFER(e) 108 UNIFY(PRODUCT(α_1, α_2), τ) 109 return α_1 110 snd e => 111 α_1 := VAR(FRESH-TYPE-VAR()) 112 α_2 := VAR(FRESH-TYPE-VAR()) 113 τ := INFER(e) 114 UNIFY(PRODUCT(α_1, α_2), τ) 115 return α_2 </pre>	<pre> UNIFY(σ, τ) 1 case σ, τ of 2 VAR(α), VAR(β) => 3 if $\alpha = \beta$ 4 return [] 5 else 6 return [β/α] 7 VAR(α), -- => 8 if $\alpha \in \text{FV}(\tau)$ 9 FAIL 10 else 11 return [τ/α] 12 --, VAR(β) => 13 if $\beta \in \text{FV}(\sigma)$ 14 FAIL 15 else 16 return [σ/β] 17 f($\sigma_1, \dots, \sigma_m$), g($\tau_1, \dots, \tau_\ell$) => 18 if f \neq g or m \neq ℓ 19 FAIL 20 else 21 U := [] 22 for i := 1 to m 23 T := UNIFY(U σ_i, U τ_i) 24 U := T \circ U 25 return U </pre>
--	--