Full Type Inference for GADTs

by Łukasz Stafiniak

University of Wrocław

Email: `lukstafi@gmail.com`

October 5, 2007

Examples

split

```
newtype Bar
newtype List : nat

newcons LNil : List 0
newcons LCons : for all (n) : Bar * List(n) --> List(n+1)

let rec split =
  function LNil -> LNil, LNil
    | LCons (x, LNil) as y -> y, LNil
    | LCons (x, LCons (y, z)) ->
        match split z with (l1, l2) ->
          LCons (x, l1), LCons (y, l2)
```

$$\mathrm{split} : \forall m, n, k[k = m + n].\mathrm{List}(k) \to (\mathrm{List}(m), \mathrm{List}(n))$$

filter

```
newtype Boolean
newtype List : type * nat

newcons B_true : Boolean
newcons B_false : Boolean

newcons LNil : for all a: List(a, 0)
newcons LCons : for all (n, a): a * List(a, n) --> List(a, n+1)

newtype Bar
external f : Bar -> Boolean

let rec filter =
  function LNil -> LNil
    | LCons (x, l) -> match f x with
          B_true -> LCons (x, filter l)
        | B_false -> filter l
```

$$\text{filter}: \forall n, k[n \leqslant k].\text{List}(\text{Bar}, k) \rightarrow \text{List}(\text{Bar}, n)$$

mergesort

```
newtype Ordered : nat * nat
newtype OList : nat * nat
newtype Nat : nat

newcons Leq : for all (a, b) with a <= b: Ordered (a, b)
newcons Geq : for all (a, b) with b <= a: Ordered (a, b)

external compare :
  for all (c, d) with c <= d : Nat(c) -> Nat(d) -> Ordered (c, d)

newcons ONil : OList(0, 0)
newcons OCons :
  for all (n, a, b) with b <= a:
    Nat(a) * OList(n, b) --> OList(n+1, a)

newtype List : nat
newtype EList : nat
newtype Impossible
```

```
newcons LNil : List(0)
newcons LCons : for all (n, a): Nat(a) * List(n) --> List(n+1)

newcons Ex : for all (n, a) : OList(n,a) --> EList(n)

newcons Impossible : with false: Impossible

let rec mergesort =
  function LNil -> Ex (ONil)
    | LCons (x0, l0) as main ->
        let rec split =
          function LNil -> LNil, LNil
            | LCons (x, LNil) as y -> y, LNil
            | LCons (x, LCons (y, z)) ->
                match split z with (l1, l2) ->
                  LCons (x, l1), LCons (y, l2) in
```

```
let rec merge =
  function ONil -> (fun l -> l)
    | OCons (a, l1) as l ->
        function ONil -> l
          | OCons (b, l3) as l2 ->
              match compare a b with
                  Leq -> OCons (a, merge l1 l2)
                | Geq -> OCons (b, merge l l3) in
match split main with
    LNil, LNil -> Impossible
  | LCons (x, LNil), LNil -> Ex (OCons (x, ONil))
  | LNil, LCons (x, LNil) -> Ex (OCons (x, ONil))
  | l1, l2 ->
      match mergesort l1 with Ex ol1 ->
        match mergesort l2 with Ex ol2 ->
          Ex (merge ol1 ol2)
```

$$\text{mergesort}: \forall n. \text{List}(n) \rightarrow \text{EList}(n)$$

eval

```
newtype Term : type
newtype Int
newtype Bool

external plus : Int -> Int -> Int
external is_zero : Int -> Bool
external if : for all a : Bool -> a -> a -> a

newcons Lit : Int --> Term Int
newcons Plus : Term Int * Term Int --> Term Int
newcons IsZero : Term Int --> Term Bool
newcons If : for all a : Term Bool * Term a * Term a --> Term a
newcons Pair : for all (a, b) : Term a * Term b --> Term (a, b)
newcons Fst : for all (a, b) : Term (a, b) --> Term a
newcons Snd : for all (a, b) : Term (a, b) --> Term b
```

```
let rec eval = function
  | Lit i -> i
  | IsZero x -> is_zero (eval x)
  | Plus (x, y) -> plus (eval x) (eval y)
  | If (b, t, e) -> if (eval b) (eval t) (eval e)
  | Pair (x, y) -> eval x, eval y
  | Fst p -> (match eval p with x, y -> x)
  | Snd p -> (match eval p with x, y -> y)
```

$$\text{eval}\colon \forall t.\text{Term}(t) \rightarrow t$$

Program-shaped constraints

The environment is split into polymorphic E and monomorphic B bindings.

- Polymorphic binding can have
  - an open type scheme $(\forall)\alpha$ (everything to be inferred),
  - a closed type scheme $\forall\bar{\beta}[\sigma].\tau$ (known type and conditions),

  and comes from
  - constructor or external declaration (closed),
  - recursive definition (open).
- Monomorphic binding comes from λ-abstraction (realized by pattern-matching clauses).

The type language is multi-sorted (currently only proper types and natural numbers). Atomic constraints:

- equality $\tau_1 \doteq \tau_2$ (for subtyping constraints, no subtyping)
- inequality $n_1 \leq n_2$ on natural numbers
- colored semi-equality $[c]\,\tau_1 \dot{\leqslant} \tau_2$ (semi-unificational constraint)
- falsehood $\perp$ (only in user-provided constraints).

Structural constraints

- conjunction
- clauses $\overline{\sigma_i \Rightarrow \rho_i(\tau_i)}(\tau)$ where τ – expected type of the clauses, τ_i – type of branch i, σ_i – premises in branch i, ρ_i – conditions to hold in branch i
- negation $\sim\sigma$ (used when $\sigma \Rightarrow \perp$ is inferred)
- pattern implication $\sigma_1 \Rightarrow \sigma_2$ (rather not important)
- recursive definition **rec** α **def** ρ_1 **in** ρ_2 where α – type variable representing the defined function, ρ_1 – defining constraints, ρ_2 – all other constraints where α can be used
- **call** α: $\tau_1 \dot{\leqslant} \tau_2(\bar{\gamma_i})$, use of recursive definition identified by variable α, where τ_1 – actual type of definition (initially $= \alpha$), τ_2 – expected type of use, γ_i are types which cannot be changed by instantiation of the call (their variables cannot be parts of semi-substitution)

Building constraints

Expressions

$$
\begin{array}{rcl}
\big\langle E \ni x\colon \forall\bar\beta[\sigma].\tau_1,\, B \vdash x\colon \tau, & & \\
\bar{\gamma_i}\big\rangle & = & \sigma[\bar\beta := \bar\alpha] \wedge \tau_1[\bar\beta := \bar\alpha] \doteq \tau, \\
 & & \bar\alpha \text{ fresh} \\
\langle E \ni x\colon (\forall)\alpha, B \vdash x\colon \tau, \bar{\gamma_i}\rangle & = & \mathbf{call}\, \alpha\colon \alpha \dot{\leqslant} \tau(\bar{\gamma_i}) \\
\langle E, B \ni x\colon \tau_1 \vdash x\colon \tau, \bar{\gamma_i}\rangle & = & \tau_1 \doteq \tau \\
\langle E, B \vdash \lambda\bar{c}\colon \tau, \bar{\gamma_i}\rangle & = & \langle E,\, B \vdash \bar{c},\, \bar{\gamma_i}.\alpha\rangle(\alpha \rightarrow \beta) \wedge \\
 & & \alpha \rightarrow \beta \doteq \tau, \alpha \text{ fresh} \\
\langle E, B \vdash e_1\, e_2\colon \tau, \bar{\gamma_i}\rangle & = & \langle E,\ B\ \vdash\ e_1\colon\ \alpha\ \rightarrow\ \tau,\ \bar{\gamma_i}\rangle\ \wedge \\
 & & \langle E, B \vdash e_2\colon \alpha, \bar{\gamma_i}\rangle, \alpha \text{ fresh} \\
\langle E, B \vdash i\colon \tau, \bar{\gamma_i}\rangle & = & \mathrm{Nat}(i) \doteq \tau \\
\langle E,\, B \vdash \text{let } x = e_1 \text{ in } e_2\colon \tau, & & \\
\bar{\gamma_i}\rangle & = & \mathbf{rec}\quad \alpha\quad \mathbf{def}\quad \langle E.x\colon\quad (\forall)\alpha, \\
 & & B \vdash e_1\colon \alpha, \bar{\gamma_i}\rangle\ \mathbf{in}\ \langle E.x\colon (\forall)\alpha, \\
 & & B \vdash e_2\colon \tau, \bar{\gamma_i}\rangle
\end{array}
$$

Clauses

$$
\begin{array}{rcl}
\langle E, B \vdash \bar{c}, \bar{\gamma_i} \rangle & = & \overline{\langle E, B \vdash c\!: \alpha \rightarrow \beta, \bar{\gamma_i} \rangle (\alpha \rightarrow \beta)}, \\
 & & \alpha, \beta \, \text{fresh} \\
\langle E, B \vdash p.e\!: \tau_1 \rightarrow \tau_2, \bar{\gamma_i} \rangle & = & \langle E \vdash p{\downarrow}\tau_1 \rangle \wedge \text{skolem}(\bar{\beta})(\sigma \Rightarrow \\
 & & \langle E, \quad B.B' \quad \vdash \quad e\!: \quad \tau_2, \quad \bar{\gamma_i} \rangle), \\
 & & \quad \text{where } \bar{\beta}, \sigma, B' = \langle E \vdash p{\uparrow}\tau_1 \rangle
\end{array}
$$

I will not go into details of $\langle E \vdash p{\downarrow}\tau_1 \rangle$ and $\langle E \vdash p{\uparrow}\tau_1 \rangle$...

Solving: infering types

- Constraints are manipulated in several passes. First, I solve equalities.
  - Constraints from implications are local to them, inferred substitutions are applied to the whole "implication subtree".
  - When solving premises, I treat constants as variables, but I always substitute-out variables if there is choice. (soundness)
  - I only substitute RHS of the **call**s with branch-local (implication subtree) substitutions.
- While solving equalities, I "solve" clauses (branchings) by generalization.
  - Each branch (implication from pattern matching) has its type approximated by solved equalities; the branching has its expected type too.
  - When there is no conflict between all branch types and the branching type, I unify; if there would be a conflict, I generalize.
  - Substitutions of each branch and of the branching are kept separately. Each branch substitution is applied to the branch as usual.
  - Only the branching subst. is applied to LHS of **call**s (and outside).

- Now I turn **call**s into semi-equalities. Each call has its own color.

$$\textbf{call}\,\alpha\colon \tau_1 \dot{\lessgtr} \tau_2(\bar{\gamma_i}) \;=:\; [\alpha_k]\,\tau_1 \dot{\lessgtr} \tau_2 \,\wedge\, \bigwedge_i\, [\alpha_k]\,\gamma_i \dot{\lessgtr} \gamma_i \,\wedge\, [\alpha_k]\mathrm{SV}(\alpha)$$

  I remember which functions were called by "second-order variables" $\mathrm{SV}(\alpha)$.

- Now I solve semi-equalities by semi-unification, with the same branch-locality restrictions as for equality.

- Imagine that a function is defined by a single branching.
  - There must be a base branch, without recursive calls.
  - I generate saturated structures for base branches. Their intersection is the "initial guess".
  - I substitute the "initial guess" for each occurrence $[\alpha_k]$ of $\mathrm{SV}(\alpha)$ in a recursive branch, applying the semi-substitution for color α_k to it.
  - I generate saturated structures for recursive branches and intersect them. This forms the inferred condition for the recursive function.

  I use a generalization of this idea to other recursive functions.

- I still do not know how to solve mutual recursion.