Aspects of efficiency in functional programming languages

Samuel Valdemar Grange

Department of Mathematics and Computer Science (IMADA)

Programming languages

- The basis of functional programming languages.
  - High-level programming language.
- Recursion and lambda lifting.
- Runtime systems.

  `fun f x = f x;`

  $\Rightarrow$ `let` $f = \lambda x.fx$ `in` $\ldots$

  or

  $\Rightarrow$ `let` $f' = (\lambda f''.$`let` $f = f''f''$ `in`

  `let` $f = \lambda x.fx$

  $)$ `in let` $f = f'f'$ `in` $\ldots$

Types

- Minimal type system.
  `let` $id = \lambda x.x$ `in` ...
  where $id : a \to a$ then
  $(id\ id)(id\ 5)$?
- Mono, poly and environment.
- Generalization and instantitation.
- Let polymorphism, recursion and type hierachy ($\sqsubseteq$).

$$\tau ::= a \,|\, \tau \to \tau \,|\, C\tau_1 \ldots \tau_n$$

$$\sigma ::= \tau \,|\, \forall a.\sigma$$

$$\Gamma ::= \epsilon \,|\, \Gamma, x : \sigma$$

Hindley-Milner

$$\text{Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\text{App}\ \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

$$\text{Abs}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}$$

$$\text{Let}\ \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ \ x = e_1\ \ \texttt{in}\ \ e_2 : \tau}$$

$$\text{Inst}\ \frac{\Gamma \vdash e : \sigma_1 \qquad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash e : \sigma_2}$$

$$\text{Gen}\ \frac{\Gamma \vdash e : \sigma \qquad a \notin \mathit{free}(\Gamma)}{\Gamma \vdash e : \forall a.\sigma}$$

Figure: Hindley-Milner type rules

Reconstruction

- Checking is top-down, inference is bottom-up.
- Begin at leaf; variable (introduced by Abs) or number, then go back up while gathering constraints.
- Constraints generated from *most general unifier* in the form of substitutions.
- Constraints are only imposed on non-bound type variables.
- Generalization over (locally) free type variables and instantitation over quantified type variables.

Damas-Milner

$$\text{Var}\ \frac{x : \sigma \in \Gamma \qquad \tau = \mathit{inst}(\sigma)}{\Gamma \vdash x : \tau, \emptyset}$$

$$\text{Abs}\ \frac{\tau_1 = \mathit{fresh} \qquad \Gamma, x : \tau_1 \vdash e : \tau_2, S}{\Gamma \vdash \lambda x.e : S\tau_1 \rightarrow \tau_2, S}$$

App

$$\frac{\Gamma \vdash e_1 : \tau_1, S_1 \ \ \tau_3 = \mathit{fresh} \qquad S_1\Gamma \vdash e_2 : \tau_2, S_2 \ \ S_3 = \mathit{unify}(S_2\tau_1, \tau_2 \rightarrow \tau_3)}{\Gamma \vdash e_1 e_2 : S_3\tau_3, S_3 \cdot S_2 \cdot S_1}$$

$$\text{Let}\ \frac{\Gamma \vdash e_1 : \tau_1, S_1 \qquad S_1\Gamma, x : (S_1\Gamma)(\tau_1) \vdash e_2 : \tau_2, S_2}{\Gamma \vdash \texttt{let}\ \ x = e_1\ \ \texttt{in}\ \ e_2 : \tau_2, S_2 \cdot S_1}$$

Figure: Algorithm W. Note that $\Gamma(\tau)$ means the generalization of τ under Γ.

Polymorphism

- Let vs Abs + App; `let` $x = e$ `in` $y \Leftrightarrow (\lambda x.y)e$?
- Abs (potentially) introduces polymorphic types and App must accept polymorphic parameters (even themselves $\Leftrightarrow$ polymorphic recursion).

$$\text{Abs}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

(a) Abs in Hindley-Milner

$$\text{Abs}\ \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \sigma \to \tau}$$

(b) Abs in System F [Wel99]

$$\text{App}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

(c) App in Hindley-Milner

$$\text{App}\ \frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau}$$

(d) App in System F [Wel99]

Polymorphism cont.

- How does one unify to polymorphic types?.
- Boils down to a problem named semi-unification, which is undecidable [Wel99; KTU93].

Evaluation

- Evaluation by substitution.
- Call by name and call by value.
- Storing labelled expressions.
- Freshness ($k \notin \mathit{bound}(e)$ and clearly $k \notin \mathit{free}(e)$).

Errata

- "Considerations of functional programming language implmentations."
- Case instead of Scott encoding.
- Multi let for shared scopes (mutual recursion) (the reconstruction algorithm), invalidates some of Section 2.

$$\frac{\Gamma \cup \{\gamma \mapsto \{\mathtt{x} \mapsto \gamma\}\mathtt{e}\},\ \{\mathtt{x} \mapsto \gamma\}\mathtt{p} \to \Theta,\ \mathtt{l} \qquad \gamma\ \mathtt{=}\ \mathtt{fresh}}{\Gamma,\ \mathtt{let\ x\ =\ e\ in\ p} \to \Theta,\ \mathtt{l}}$$

(a) Let with recursion

$$\frac{\Gamma, x_1 : \tau_1 \ldots, x_n : \tau_n \vdash e_1 : \sigma_1, \ldots e_n : \sigma_n \qquad \Gamma, x_1 : \sigma_1 \ldots, x_n : \sigma_n \vdash e : \tau}{\Gamma \vdash \mathtt{let}\ \{x_1 = e_1, \ldots x_n = e_n\}\ \mathtt{in}\ e : \tau}$$

(a) Multi Let proof rule.

Errata

- Type constructors and sum-types missing (implemented in the fixes branch as described in the thesis).

```
type Stack a = | Nil
-| Cons a (List a);
+| Cons a (Stack a);

-introduced Cons with type ∀c0.∀d0.∀e0.∀g0.(c0 ->
(d0 -> (e0 -> ((c0 -> (d0 -> g0)) -> g0))))
+introduced Cons with type ∀adt_f0.(adt_f0 ->
((ADT(List) adt_f0) -> (ADT(List) adt_f0)))

-applied ((Cons 1) ((Cons 2) ((Cons 3) Nil)))
with type ((Integer ) -> (((Integer ) -> ((h1 ->
(((Integer ) -> ((n1 -> (((Integer ) -> ((p1 ->
(q1 -> p1)) -> o1)) -> o1)) -> i1)) -> i1)) ->
(Integer ))) -> (Integer )))
```

A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. "The Undecidability of the Semi-unification Problem". English. In: *Information and computation* 102.1 (1993), pp. 83–101.

J.B. Wells. "Typability and type checking in System F are equivalent and undecidable". In: *Annals of Pure and Applied Logic* 98.1 (1999), pp. 111–156. ISSN: 0168-0072. DOI: https://doi.org/10.1016/S0168-0072(98)00047-5. URL: https://www.sciencedirect.com/science/article/pii/S0168007298000475.