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 = x + (f (x - 1)); fun main = f 5;`

  $\Rightarrow \texttt{let } f = \lambda x.x + (f\,(x-1)) \texttt{ in } f\,5$

  or

  $\Rightarrow \texttt{let } f' = (\lambda f''.\texttt{let } f = f''f'' \texttt{ in}$

  $\qquad \texttt{let } f = \lambda x.x + (f\,(x-1))$

  $) \texttt{ in let } f = f'f' \texttt{ in } f\,5$

Types

- Mono, poly and environment.
- Generalization and instantitation.
- Let polymorphism and type hierachy ($\sqsubseteq$).

$$\tau ::= a \,|\, \tau \to \tau \,|\, C\tau_1 \ldots \tau_n$$
$$\sigma ::= \tau \,|\, \forall a.\sigma$$
$$\Gamma ::= \epsilon \,|\, \Gamma, x : \sigma$$

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.

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].

Errata

- "Considerations of functional programming language implmentations."
- Multi let and mutual recursion (the reconstruction algorithm being trivial).

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

(a) Multi Let proof rule.

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.