

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;
```

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

or

```
 $\Rightarrow \text{let } f' = (\lambda f''. \text{let } f = f'' f'' \text{ in}$   
     $\text{let } f = \lambda x. x + (f(x - 1))$   
 $) \text{ in let } f = f' f' \text{ in } f\ 5$ 
```

Types

- ▶ Mono, poly and environment.
- ▶ Generalization and instantiation.
- ▶ Let polymorphism and type hierarchy (\sqsubseteq).

$$\tau ::= a \mid \tau \rightarrow \tau \mid C\tau_1 \dots \tau_n$$

$$\sigma ::= \tau \mid \forall a. \sigma$$

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

Hindley-Milner

$$\begin{array}{c} \text{Var} \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \\[1em] \text{App} \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \\[1em] \text{Abs} \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \\[1em] \text{Let} \frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \\[1em] \text{Inst} \frac{\Gamma \vdash e : \sigma_1 \quad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash e : \sigma_2} \\[1em] \text{Gen} \frac{\Gamma \vdash e : \sigma \quad a \notin \text{free}(\Gamma)}{\Gamma \vdash e : \forall a. \sigma} \end{array}$$

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 instantiation over quantified type variables.

Damas-Milner

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

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

App

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

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

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).

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

(a) Abs in Hindley-Milner (b) Abs in System F [Wel99]

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

(c) App in Hindley-Milner

$$\text{App} \frac{\Gamma \vdash e_1 : \sigma \rightarrow \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau}$$

(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 need and call by value.
- ▶ Storing labelled expressions.
- ▶ Freshness.

Errata

- ▶ "Considerations of functional programming language implementations."
- ▶ Multi let for shared scopes (mutual recursion) (the reconstruction algorithm).

$$\frac{\Gamma, x_1 : \tau_1 \dots, x_n : \tau_n \vdash e_1 : \sigma_1, \dots e_n : \sigma_n \quad \Gamma, x_1 : \sigma_1 \dots, x_n : \sigma_n \vdash e : \tau}{\Gamma \vdash \text{let } \{x_1 = e_1, \dots x_n = e_n\} \text{ in } e : \tau}$$

(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  $\forall c0.\forall d0.\forall e0.\forall g0.(c0 \rightarrow$   
   $(d0 \rightarrow (e0 \rightarrow ((c0 \rightarrow (d0 \rightarrow g0)) \rightarrow g0))))$   
+introduced Cons with type  $\forall \text{adt\_f0} . (\text{adt\_f0} \rightarrow$   
   $((\text{ADT}(\text{List}) \text{adt\_f0}) \rightarrow (\text{ADT}(\text{List}) \text{adt\_f0})))$ 
```

```
-applied ((Cons 1) ((Cons 2) ((Cons 3) Nil)))  
with type ((Integer )  $\rightarrow (((\text{Integer } ) \rightarrow ((h1 \rightarrow$   
   $((\text{Integer } ) \rightarrow ((n1 \rightarrow (((\text{Integer } ) \rightarrow ((p1 \rightarrow$   
   $(q1 \rightarrow p1)) \rightarrow o1)) \rightarrow o1)) \rightarrow i1)) \rightarrow i1)) \rightarrow$   
   $(\text{Integer } ))) \rightarrow (\text{Integer } ))$ 
```



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](https://doi.org/10.1016/S0168-0072(98)00047-5). URL: <https://www.sciencedirect.com/science/article/pii/S0168007298000475>.