

The F Machine: an Abstract Machine for System F

Jeremy G. Siek

January 2, 2012

1 Introduction

This short note describes an abstract machine for System F, named the F Machine. The abstract machine was designed with two purposes in mind. First, the machine was designed to be reasonably efficient so that it can be used to execute non-trivial programs. (Of course, it's not a compiler, so it does not provide a high performance implementation.) Second, the machine was designed to make a mechanized proof of type safety in Isabelle relatively straightforward.

Here's a list of the important design decision in the F Machine:

- The machine never substitutes anything into a term. Substitution is inefficient and it is tedious to prove lemmas about substitution. The machine uses environments instead.
- It's an abstract machine with a stack, as opposed to a reduction semantics. The advantage here is that we don't need evaluation contexts and a unique decomposition lemma.
- The machine uses a mixed representation for type variables: names for free variables and de Bruijn indices for bound variables, as promoted by Conor McBride and James McKinna in the paper *I am not a number: I am a free variable*. This representation is nice because type equality is just syntactic equality while at the same time avoiding some of the nastiness of de Bruijn indices.
- Terms are explicitly typed so that it is easy to implement a type checking algorithm. The machine itself is also explicitly typed so that it is

easy to run the type checker at every step of evaluation. Because type safety has been proved, this kind of testing is no longer necessary. However, when someone were to use this formalization as a basis for a larger language, then being able to test via type checking is quite valuable.

The two standard operational semantics for System F are the erasure-based semantics and the type-passing semantics. The F Machine uses neither approach. The F Machine can be viewed as an abstract machine for the version of System F that was presented in the *Blame for All* paper at POPL 2011. The main idea behind the F Machine is that it keeps track of the border between an instantiated generic and its context using a special kind of cast. In the Blame for All paper, the casts were built into the terms, whereas here they show up in the machine and in the values. Thus, in the F Machine there is a nice separation between the terms of the language and the stuff that is needed for the run-time system.

The proof of type safety in Isabelle is also available from my web page.

2 Type System

Terms are flattened out. Stuff relating to function calls and instantiating generics appears in statements whereas everything else is in the expressions.

Type rules for expressions.

$$\frac{\text{lookup } x \ \Gamma = A}{\Gamma \vdash_e x : A} \quad \Gamma \vdash_e c : \text{typeof } c$$

$$\frac{\text{typeof-opr } f = A \rightarrow B \quad \Gamma \vdash_e e : A}{\Gamma \vdash_e f(e) : B}$$

$$\frac{(x, A) \cdot \Gamma \vdash_s s : B \quad x \notin \text{dom } \Gamma \quad \Gamma \vdash A}{\Gamma \vdash_e \lambda x. s : A \rightarrow B : A \rightarrow B}$$

$$\frac{(\alpha, \text{type}) \cdot \Gamma \vdash_s s : A \quad \alpha \notin \text{dom } \Gamma}{\Gamma \vdash_e \Lambda \alpha. s : \forall ([\alpha \mapsto \theta] A) : \forall ([\alpha \mapsto \theta] A)}$$

Type rules for statements.

$$\frac{\Gamma \vdash_e e : A \quad x \notin \text{dom } \Gamma \quad (x, A) \cdot \Gamma \vdash_s s : B}{\Gamma \vdash_s \text{let } x=e \text{ in } s : B} \quad \frac{\Gamma \vdash_e e : A}{\Gamma \vdash_s \text{return } e : A}$$

$$\begin{array}{c}
\Gamma \vdash_e e1 : A \rightarrow B \\
\Gamma \vdash_e e2 : A \quad x \notin \text{dom } \Gamma \quad (x, B) \cdot \Gamma \vdash_s s : C \\
\hline
\Gamma \vdash_s \text{let } x=e1(e2) \text{ in } s : C \\
\\
\Gamma \vdash_e e : \forall A \quad \Gamma \vdash B \quad x \notin \text{dom } \Gamma \quad (x, ([\theta \mapsto B]A)) \cdot \Gamma \vdash_s s : C \\
\hline
\Gamma \vdash_s \text{let } x=e\langle B \rangle \text{ in } s : C
\end{array}$$

3 Reduction Rules

The following is the definition of the evaluation function, which turns expressions into values. The *return* and *option-bind* are there because we're in a monad for the option type to deal with errors.

$$\begin{array}{l}
\llbracket x \rrbracket_{\varrho} \Sigma = \text{lookup } x \ \varrho \\
\llbracket c \rrbracket_{\varrho} \Sigma = \text{return } (c) \\
\llbracket f(e) \rrbracket_{\varrho} \Sigma = \text{option-bind } (\llbracket e \rrbracket_{\varrho} \Sigma) (\delta f) \\
\llbracket \lambda x.s : A \rrbracket_{\varrho} \Sigma = \text{return } (\langle \lambda x.s, \varrho, \Sigma, A \Rightarrow A \rangle) \\
\llbracket \Lambda x.s : A \rrbracket_{\varrho} \Sigma = \text{return } (\langle \Lambda x.s, \varrho, \Sigma, A \Rightarrow A \rangle)
\end{array}$$

The following is the definition of the cast function.

$$\begin{array}{l}
c : A \Rightarrow \alpha = [c:\alpha] \\
c : A \Rightarrow B = c \\
[c:\alpha] : A \Rightarrow \beta = [c:\beta] \\
[c:\alpha] : A \Rightarrow B = c \\
\langle \lambda x.s, \varrho, \Sigma, A \Rightarrow B \rangle : B' \Rightarrow C = \langle \lambda x.s, \varrho, \Sigma, A \Rightarrow C \rangle \\
\langle \Lambda x.s, \varrho, \Sigma, A \Rightarrow B \rangle : B' \Rightarrow C = \langle \Lambda x.s, \varrho, \Sigma, A \Rightarrow C \rangle
\end{array}$$

Now for the main event, the reduction relation for the F Machine.

$$\begin{array}{c}
\frac{\llbracket e \rrbracket_{\varrho} \Sigma = v}{(\text{let } x=e \text{ in } s, \varrho, \Sigma, \kappa) \mapsto (s, (x, v) \cdot \varrho, \Sigma, \kappa)} \\
\\
\frac{\llbracket e \rrbracket_{\varrho} \Sigma = v}{(\text{return } e, \varrho, \Sigma, (x, A, B, s, \varrho', \Sigma') \cdot \kappa) \mapsto (s, (x, v : A \Rightarrow B) \cdot \varrho', \Sigma', \kappa)} \\
\\
\frac{\llbracket e_1 \rrbracket_{\varrho} \Sigma = (\langle \lambda y.s', \varrho', \Sigma', A_1 \rightarrow A_2 \Rightarrow B_1 \rightarrow B_2 \rangle) \quad \llbracket e_2 \rrbracket_{\varrho} \Sigma = v_2}{(\text{let } x=e_1(e_2) \text{ in } s, \varrho, \Sigma, \kappa) \mapsto (s', (y, v_2 : B_1 \Rightarrow A_1) \cdot \varrho', \Sigma', (x, A_2, B_2, s, \varrho, \Sigma) \cdot \kappa)} \\
\\
\frac{\llbracket e \rrbracket_{\varrho} \Sigma = (\langle \Lambda \alpha.s', \varrho', \Sigma', \forall A \Rightarrow \forall B \rangle)}{(\text{let } x=e\langle C \rangle \text{ in } s, \varrho, \Sigma, \kappa) \mapsto (s', \varrho', (\alpha, \Sigma(C)) \cdot \Sigma', (x, [\theta \mapsto \alpha]A, [\theta \mapsto C]B, s, \varrho, \Sigma) \cdot \kappa)}
\end{array}$$

If you like to have efficient tail recursion, it's easy to add support for tail calls.

4 Type System for the Run-Time Structures

Type rules for values.

$$\frac{\text{typeof } c = A}{\Sigma' \vdash_v c : A} \quad \frac{\Sigma'(\alpha) = \text{typeof } c}{\Sigma' \vdash_v [c:\alpha] : \alpha}$$

$$\frac{\text{wf-env } \Gamma \quad \Gamma; \Sigma \vdash \varrho \quad x \notin \text{dom } \Gamma \quad \Gamma \vdash A_1 \quad \Sigma(A_1 \rightarrow A_2) = \Sigma'(B) \quad (x, A_1) \cdot \Gamma \vdash_s s : A_2}{\Sigma' \vdash_v \langle \lambda x.s, \varrho, \Sigma, A_1 \rightarrow A_2 \Rightarrow B \rangle : B}$$

$$\frac{\text{wf-env } \Gamma \quad \Gamma; \Sigma \vdash \varrho \quad \alpha \notin \text{dom } \Gamma \quad \Sigma(\forall([\alpha \mapsto 0]A)) = \Sigma'(B) \quad (\alpha, \text{type}) \cdot \Gamma \vdash_s s : A}{\Sigma' \vdash_v \langle \Lambda \alpha.s, \varrho, \Sigma, \forall([\alpha \mapsto 0]A) \Rightarrow B \rangle : B}$$

Type rules for environments.

$$\frac{\boxed{\ }; \boxed{\ } \vdash \boxed{\ }}{\frac{x \notin \text{dom } \Gamma \quad \Sigma \vdash_v v : A \quad \Gamma \vdash A \quad \Gamma; \Sigma \vdash \varrho}{(x, A) \cdot \Gamma; \Sigma \vdash (x, v) \cdot \varrho}}$$

$$\frac{\alpha \notin \text{dom } \Gamma \quad \boxed{\ } \vdash T \quad \Gamma; \Sigma \vdash \varrho}{(\alpha, \text{type}) \cdot \Gamma; (\alpha, T) \cdot \Sigma \vdash \varrho}$$

Type rules for the stack.

$$\Sigma \vdash \boxed{\ } : A \Rightarrow A$$

$$\frac{\text{wf-env } \Gamma \quad \Gamma; \Sigma \vdash \varrho \quad \Sigma'(A) = \Sigma(B) \quad (x, B) \cdot \Gamma \vdash_s s : C \quad \Gamma \vdash B \quad x \notin \text{dom } \Gamma \quad \Sigma \vdash k : C \Rightarrow D}{\Sigma' \vdash (x, A, B, s, \varrho, \Sigma) \cdot k : A \Rightarrow D}$$

Type rule for the F Machine's state.

$$\frac{\text{wf-env } \Gamma \quad \Gamma; \Sigma \vdash \varrho \quad \Gamma \vdash_s s : A \quad \Sigma \vdash k : A \Rightarrow B}{\vdash (s, \varrho, \Sigma, k) : B}$$

5 Type Safety

Definition 1. $final\ s = (\exists e\ \varrho\ \Sigma. s = (return\ e, \varrho, \Sigma, []))$

Theorem 1 (Step Safety). *If $\vdash s : A$ then $final\ s \vee (\exists s'. s \mapsto s' \wedge \vdash s' : A)$.*