MinML: Syntax, Static Semantics, Dynamic Semantics, and Type Safety

Amr Sabry

February 12, 2002

1 Introduction

This note describes the syntax, static semantics, and dynamic semantics of a small functional language, and then proves type safety. The material is based on the early chapters of Robert Harper's recent manuscript *Programming Languages: Theory and Practice* [1] but the dynamic semantics has been adapted to be as close as possible to the style used by the Jbook [2] for describing the semantics of Java. The proof of type safety is more involved because the semantics uses several intermediate structures like environments and stacks whose invariants have to be maintained during subject reduction.

2 Syntax

The syntax is given by the following BNF. In the following n ranges over integer constants; and both x and f range over identifiers:

Types	t	::=	$\texttt{int}$	*integer type*
		|	$\texttt{bool}$	*boolean type*
		|	$t \to t$	*function type*
Operators	o	::=	$+ \mid - \mid * \mid / \mid = \mid <$	
Expressions	e	::=	n	*integer constants*
		|	x	*variables*
		|	$o(e,e)$	*primitive applications*
		|	true | false	*boolean constants*
		|	**if** e **then** e **else** e	*conditional expressions*
		|	(**fun** $t\ f\ (t\ x)\ \{e\}$)	*user-defined recursive functions*
		|	$e(e)$	*function applications*

Here are some examples of programs with their intuitive semantics:

p_1	=	5	a trivial program
p_2	=	$+(+(1,2),+(3,4))$	another trivial program
p_3	=	$+(+(1,2),+(/(1,0),4))$	divides by zero
p_5	=	(**fun** $\texttt{int}\ f\ (\texttt{int}\ x)\ \{$**if** $=(x,0)$ **then** 1 **else** $*(x,f(-(x,1)))\}$)	factorial
p_5	=	(**fun** $\texttt{int}\ f\ (\texttt{int}\ x)\ \{f(x)\})(0)$	infinite loop
p_6	=	**if** true **then** 1 **else** (**fun** $\texttt{int}\ f\ (\texttt{int}\ x)\ \{f(x)\})(0)$	evaluates to 1
p_7	=	$+(\text{true},1)$	type error
p_8	=	(**fun** $\texttt{int}\ f\ (\texttt{bool}\ x)\ \{x\}$)	another type error

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash x : t} \text{ if } (x : t) \in \Gamma$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash = (e_1, e_2) : \texttt{bool}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash < (e_1, e_2) : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash o(e_1, e_2) : \texttt{int}} \text{ if } o \text{ is one of } \{+, -, *, /\}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : t}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t}$$

$$\frac{\Gamma, x : t_x, f : t_x \to t_r \vdash e : t_r}{\Gamma \vdash (\textbf{fun } t_r \ f \ (t_x \ x) \ \{e\}) : t_x \to t_r} \qquad \frac{\Gamma \vdash e_1 : t_2 \to t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1(e_2) : t}$$

Figure 1: Typing Rules

3 Static Semantics

The static semantics filters the set of syntactically correct programs to exclude those programs that are not well-typed according to the rules in Figure 1. The intention (which we formalize and prove in Section 5) is that the evaluation of a well-typed program will be guaranteed not to encounter a certain class of errors. Note that a well-typed program may still encounter errors in another class: non-termination and division by zero in our small language. Also note that a program that fails to typecheck according to our rules may still evaluate without any errors. (For example **if** true **then** 1 **else** $+(2, \mathsf{false})$ does not typecheck but would evaluate to 1 if allowed to execute.) In summary, the type rules are a static approximation of what constitutes "good behavior." The fact that such static approximations can never be exact means that one can always develop a more sophisticated type system that accepts a different class of programs.

Before getting to the type rules, note that the syntax forces every variable declaration to be associated with a type: the formal parameter of a function must be given a type, and the function declaration itself must be given a return type. Intuitively speaking the process of type checking is to make sure that every use of a variable is consistent with its declaration. To facilitate this process, variable declarations and their types are collected in a table (called an environment and denoted with the letter Γ) which is propagated by the type rules following the usual scoping rules. The judgment $\Gamma \vdash e : t$ means that given the environment Γ, we can prove that expression e has type t. Each rule proves a judgment for a certain kind of expression making assumptions about the judgments for the subexpressions.

Here is an example derivation for the factorial example:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash x : \texttt{int}} \quad \dfrac{}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash 0 : \texttt{int}}}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash = (x, 0) : \texttt{bool}} \quad \dfrac{}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash 1 : \texttt{int}} \quad C}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash \textbf{if } = (x, 0) \textbf{ then } 1 \textbf{ else } * (x, f(-(x, 1))) : \texttt{int}}}{\emptyset \vdash (\textbf{fun } \texttt{int } f \ (\texttt{int } x) \ \{\textbf{if } = (x, 0) \textbf{ then } 1 \textbf{ else } * (x, f(-(x, 1)))\}) : \texttt{int} \to \texttt{int}}$$

where the derivation C is:

$$\dfrac{\dfrac{}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash x : \texttt{int}} \quad \dfrac{\dfrac{}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash f : \texttt{int} \to \texttt{int}} \quad \dfrac{\dfrac{}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash x : \texttt{int}} \quad \dfrac{}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash 1 : \texttt{int}}}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash -(x, 1) : \texttt{int}}}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash f(-(x, 1)) : \texttt{int}}}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash *(x, f(-(x, 1))) : \texttt{int}}$$

4 Dynamic Semantics

The dynamic semantics is a function that maps a syntactically valid program to a value. The function is partial since programs may diverge, cause errors like division by zero, or get stuck if a nonsensical operation such as adding 1 to true is attempted at runtime. In the next section we will be concerned with the proof that well-typed programs can never get stuck during evaluation.

To be close to the ASM framework, the dynamic semantics is specified using an abstract machine. The machine has three components: the code being evaluated, the environment that holds the values for the free variables in the code, and the stack of activation records. The evaluation proceeds in steps: at each step, a subexpression of the entire program is chosen for evaluation. The subexpressions of the program are gradually replaced by their values until the entire program reduces to a value or evaluation gets stuck.

To specify this evaluation formally, we first need to define the set of values and then extend the syntax of expressions to accommodate the fact that some subexpressions may be replaced by values or runtime errors.

Values	v	::=	$\underline{n}$	*integer values*
		|	$\underline{\mathsf{true}} \mid \underline{\mathsf{false}}$	*boolean values*
		|	$\langle \underline{clos}(\textbf{fun}\ t\ f\ (t\ x)\ \{e\}), \rho\rangle$	*function values (closures)*
		|	$\underline{DivZero}$	*error condition for division by zero*
Proper Values	v^*	::=	$\underline{n}$	*integer values*
		|	$\underline{\mathsf{true}} \mid \underline{\mathsf{false}}$	*boolean values*
		|	$\langle \underline{clos}(\textbf{fun}\ t\ f\ (t\ x)\ \{e\}), \rho\rangle$	*function values (closures)*
Environments	ρ	::=	$\{x_1 = v_1, \cdots, x_n = v_n\}$	
Runtime Expressions	e	::=	n	*integer constants*
		|	x	*variables*
		|	$o(e, e)$	*primitive applications*
		|	$\mathsf{true} \mid \mathsf{false}$	*boolean constants*
		|	**if** e **then** e **else** e	*conditional expressions*
		|	(**fun** $t\ f\ (t\ x)\ \{e\}$)	*user-defined recursive functions*
		|	$e(e)$	*function applications*
		|	$\underline{n}$	*integer values*
		|	$\underline{\mathsf{true}} \mid \underline{\mathsf{false}}$	*boolean values*
		|	$\langle \underline{clos}(\textbf{fun}\ t\ f\ (t\ x)\ \{e\}), \rho\rangle$	*function values (closures)*
		|	$\underline{DivZero}$	*error condition for division by zero*

The process of choosing one subexpression to evaluate is best explained using the notion of evaluation contexts defined below.

Evaluation contexts	E	::=	$[\,]$	*empty context*
		|	$o(E, e)$	*evaluate left argument first*
		|	$o(v^*, E)$	*when done with left argument, go to right*
		|	**if** E **then** e **else** e	*need the value of the test*
		|	$E(e)$	*left first*
		|	$v^*(E)$	*then right*

It is easy to verify that every runtime expression e has a *unique decomposition* into an evaluation context E and a subexpression of interest. This is formalized in the following lemma.

Lemma 4.1 (Unique Decomposition) *Every runtime expression e is in one (and only one) of the following forms:*

- *a value* v,
- *an evaluation context* E *filled with:*
  1. *an integer constant* n,
  2. *a variable* x,
  3. *a boolean constant* true *or* false,
  4. *a function declaration* ($\textbf{\textit{fun}}\ t\ f\ (t\ x)\ \{e\}$),
  5. *a primitive operation where the first argument is an exception* $o(\underline{\textit{DivZero}}, e)$,
  6. *an application of a primitive operation to two values* $o(v_1^*, v_2)$ *where the first value is guaranteed to be a proper value,*
  7. *a conditional expression with an evaluated test position* $\textbf{\textit{if}}\ v\ \textbf{\textit{then}}\ e_1\ \textbf{\textit{else}}\ e_2$,
  8. *an application where the function position is an exception* $\underline{\textit{DivZero}}(e)$,
  9. *an application of two values* $v_1^*(v_2)$ *where the function position is guaranteed to be a proper value.*

Each activation record is of the form (E, ρ). In other words, when a function call occurs within an evaluation context E, we save the evaluation context on the stack together with the environment needed for its free variables.

To evaluate a program e, the abstract machine is put in the initial state $\langle e, \emptyset, [] \rangle$. A successful evaluation terminates with a state of the form $\langle v, \rho, [] \rangle$ for some value v and environment ρ. The transitions of the machine are:

$$
\begin{array}{rcl}
\langle v, \rho', (E, \rho) : \kappa \rangle & \longmapsto & \langle E[v], \rho, \kappa \rangle \\
\\
\langle E[n], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{n}], \rho, \kappa \rangle \\
\langle E[x], \rho, \kappa \rangle & \longmapsto & \langle E[\rho(x)], \rho, \kappa \rangle \text{ if } x \in dom(\rho) \\
\langle E[\mathsf{true}], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\mathsf{true}}], \rho, \kappa \rangle \\
\langle E[\mathsf{false}], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\mathsf{false}}], \rho, \kappa \rangle \\
\langle E[(\textbf{fun}\ t'\ f\ (t\ x)\ \{e\})], \rho, \kappa \rangle & \longmapsto & \langle E[\langle \underline{\textit{clos}}(\textbf{fun}\ t'\ f\ (t\ x)\ \{e\}), \rho \rangle], \rho, \kappa \rangle \\
\\
\langle E[+(\underline{n_1}, \underline{n_2})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{n_1 + n_2}], \rho, \kappa \rangle \\
\langle E[*(\underline{n_1}, \underline{n_2})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{n_1 * n_2}], \rho, \kappa \rangle \\
\langle E[-(\underline{n_1}, \underline{n_2})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{n_1 - n_2}], \rho, \kappa \rangle \\
\langle E[/(\underline{n_1}, \underline{n_2})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{n_1/n_2}], \rho, \kappa \rangle \text{ if } n_2 \neq 0 \\
\langle E[/(\underline{n}, \underline{0})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textit{DivZero}}], \rho, \kappa \rangle \\
\langle E[=(\underline{n}, \underline{n})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\mathsf{true}}], \rho, \kappa \rangle \\
\langle E[=(\underline{n_1}, \underline{n_2})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\mathsf{false}}], \rho, \kappa \rangle \text{ if } n_1 \neq n_2 \\
\langle E[<(\underline{n_1}, \underline{n_2})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\mathsf{true}}], \rho, \kappa \rangle \text{ if } n_1 < n_2 \\
\langle E[<(\underline{n_1}, \underline{n_2})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\mathsf{false}}], \rho, \kappa \rangle \text{ if } n_1 \geq n_2 \\
\langle E[o(v, \underline{\textit{DivZero}})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textit{DivZero}}], \rho, \kappa \rangle \\
\langle E[o(\underline{\textit{DivZero}}, e)], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textit{DivZero}}], \rho, \kappa \rangle \\
\\
\langle E[\textbf{if}\ \underline{\mathsf{true}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2], \rho, \kappa \rangle & \longmapsto & \langle E[e_1], \rho, \kappa \rangle \\
\langle E[\textbf{if}\ \underline{\mathsf{false}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2], \rho, \kappa \rangle & \longmapsto & \langle E[e_2], \rho, \kappa \rangle \\
\langle E[\textbf{if}\ \underline{\textit{DivZero}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textit{DivZero}}], \rho, \kappa \rangle \\
\\
\langle E[cl(v)], \rho, \kappa \rangle & \longmapsto & \langle e, \rho'[x := v, f := cl], (E, \rho) : \kappa \rangle \\
 & & \text{where } cl = \langle \underline{\textit{clos}}(\textbf{fun}\ t'\ f\ (t\ x)\ \{e\}), \rho' \rangle \\
 & & \text{and } v \neq \underline{\textit{DivZero}} \\
\langle E[\underline{\textit{DivZero}}(e)], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textit{DivZero}}], \rho, \kappa \rangle \\
\langle E[\langle \underline{\textit{clos}}(\textbf{fun}\ t'\ f\ (t\ x)\ \{e\}), \rho' \rangle(\underline{\textit{DivZero}})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textit{DivZero}}], \rho, \kappa \rangle
\end{array}
$$

As a small example, we trace the evaluation of the program:

$$(\textbf{fun}\ \texttt{int}\ f\ (\texttt{int}\ x)\ \{\textbf{if}\ =(x,0)\ \textbf{then}\ 1\ \textbf{else}\ *(x, f(-(x,1)))\})(1)$$

which should compute the factorial of 1:

$$\begin{array}{ll}
 & \langle(\textbf{fun}\ \texttt{int}\ f\ (\texttt{int}\ x)\ \{\textbf{if}\ =(x,0)\ \textbf{then}\ 1\ \textbf{else}\ *(x,f(-(x,1)))\})(1), \emptyset, []\rangle \\
\longmapsto & \langle\langle \underline{clos}(\textbf{fun}\ \texttt{int}\ f\ (\texttt{int}\ x)\ \{\textbf{if}\ =(x,0)\ \textbf{then}\ 1\ \textbf{else}\ *(x,f(-(x,1)))\}), \emptyset\rangle(1), \emptyset, []\rangle \\
\longmapsto & \langle\langle \underline{clos}\ldots, \emptyset\rangle(\underline{1}), \emptyset, []\rangle \\
\longmapsto & \langle\textbf{if}\ =(x,0)\ \textbf{then}\ 1\ \textbf{else}\ *(x,f(-(x,1))), \{x=\underline{1}, f=\langle\underline{clos}\ldots,\emptyset\rangle\}, ([],\emptyset):[]\rangle \\
\longmapsto & \langle\textbf{if}\ =(\underline{1},0)\ \textbf{then}\ 1\ \textbf{else}\ *(x,f(-(x,1))), \{x=\underline{1}, f=\langle\underline{clos}\ldots,\emptyset\rangle\}, ([],\emptyset):[]\rangle \\
\longmapsto & \langle\textbf{if}\ =(\underline{1},\underline{0})\ \textbf{then}\ 1\ \textbf{else}\ *(x,f(-(x,1))), \{x=\underline{1}, f=\langle\underline{clos}\ldots,\emptyset\rangle\}, ([],\emptyset):[]\rangle \\
\longmapsto & \langle\textbf{if}\ \underline{\textsf{false}}\ \textbf{then}\ 1\ \textbf{else}\ *(x,f(-(x,1))), \{x=\underline{1}, f=\langle\underline{clos}\ldots,\emptyset\rangle\}, ([],\emptyset):[]\rangle \\
\longmapsto & \langle*(x,f(-(x,1))), \{x=\underline{1}, f=\langle\underline{clos}\ldots,\emptyset\rangle\}, ([],\emptyset):[]\rangle \\
\longmapsto & \langle*(\underline{1},f(-(x,1))), \{x=\underline{1}, f=\langle\underline{clos}\ldots,\emptyset\rangle\}, ([],\emptyset):[]\rangle \\
\longmapsto & \langle*(\underline{1},\langle\underline{clos}\ldots,\emptyset\rangle(-(x,1))), \{x=\underline{1}, f=\langle\underline{clos}\ldots,\emptyset\rangle\}, ([],\emptyset):[]\rangle \\
\longmapsto & \langle*(\underline{1},\langle\underline{clos}\ldots,\emptyset\rangle(-(\underline{1},1))), \{x=\underline{1}, f=\langle\underline{clos}\ldots,\emptyset\rangle\}, ([],\emptyset):[]\rangle \\
\longmapsto & \langle*(\underline{1},\langle\underline{clos}\ldots,\emptyset\rangle(-(\underline{1},\underline{1}))), \{x=\underline{1}, f=\langle\underline{clos}\ldots,\emptyset\rangle\}, ([],\emptyset):[]\rangle \\
\longmapsto & \langle*(\underline{1},\langle\underline{clos}\ldots,\emptyset\rangle(\underline{0})), \{x=\underline{1}, f=\langle\underline{clos}\ldots,\emptyset\rangle\}, ([],\emptyset):[]\rangle \\
\longmapsto & \langle\textbf{if}\ =(x,0)\ \textbf{then}\ 1\ \textbf{else}\ *(x,f(-(x,1))), \{x=\underline{0}, f=\ldots\}, (*(\underline{1},[]), \{x=\underline{1}, f=\ldots\}):([],\emptyset):[]\rangle \\
\longmapsto & \langle\textbf{if}\ =(\underline{0},0)\ \textbf{then}\ 1\ \textbf{else}\ *(x,f(-(x,1))), \{x=\underline{0}, f=\ldots\}, (*(\underline{1},[]), \{x=\underline{1}, f=\ldots\}):([],\emptyset):[]\rangle \\
\longmapsto & \langle\textbf{if}\ =(\underline{0},\underline{0})\ \textbf{then}\ 1\ \textbf{else}\ *(x,f(-(x,1))), \{x=\underline{0}, f=\ldots\}, (*(\underline{1},[]), \{x=\underline{1}, f=\ldots\}):([],\emptyset):[]\rangle \\
\longmapsto & \langle\textbf{if}\ \underline{\textsf{true}}\ \textbf{then}\ 1\ \textbf{else}\ *(x,f(-(x,1))), \{x=\underline{0}, f=\ldots\}, (*(\underline{1},[]), \{x=\underline{1}, f=\ldots\}):([],\emptyset):[]\rangle \\
\longmapsto & \langle 1, \{x=\underline{0}, f=\ldots\}, (*(\underline{1},[]), \{x=\underline{1}, f=\ldots\}):([],\emptyset):[]\rangle \\
\longmapsto & \langle \underline{1}, \{x=\underline{0}, f=\ldots\}, (*(\underline{1},[]), \{x=\underline{1}, f=\ldots\}):([],\emptyset):[]\rangle \\
\longmapsto & \langle*(\underline{1},\underline{1}), \{x=\underline{1}, f=\ldots\}, ([],\emptyset):[]\rangle \\
\longmapsto & \langle\underline{1}, \{x=\underline{1}, f=\ldots\}, ([],\emptyset):[]\rangle \\
\longmapsto & \langle\underline{1}, \emptyset, []\rangle
\end{array}$$

5 Type Safety

Type safety means that if a program typechecks then its evaluation cannot get stuck. Thus what we wish to guarantee is that the evaluation of a program p of a type t can only result in one of the following cases:

- a proper value v^* of type t,
- an exception $\underline{DivZero}$, or
- an infinite loop.

To understand the proof strategy, consider the case of a program p of type t evaluating to a proper value v^* in a million transitions of our abstract machine. To relate the initial program and its type to the value v^*, it is natural to proceed one machine transition at a time. If we can prove that each machine transition preserves the type of the expression, and that well-typed machine states can always make progress until they become final states, then we can conclude our main result by induction on the number of machine transitions. This is the gist of our proof technique. This basic idea has to be extended however since evaluation uses auxiliary structures (runtime values, environments, and stack frames) that affect the current expression being evaluated. Hence to guarantee that the current expression remains well-typed, we must maintain certain type information about runtime values, environments and stack frames and propagate this information at every transition.