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

Amr Sabry

February 21, 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	$::=$	`int`	*integer type*
		$\mid$	`bool`	*boolean type*
		$\mid$	$t \rightarrow t$	*function type*
Operators	o	$::=$	$+ \mid - \mid * \mid / \mid = \mid <$	
Program Expressions	p	$::=$	n	*integer constants*
		$\mid$	x	*variables*
		$\mid$	$o(p,p)$	*primitive applications*
		$\mid$	true $\mid$ false	*boolean constants*
		$\mid$	**if** p **then** p **else** p	*conditional expressions*
		$\mid$	$($**fun** $t\ f\ (t\ x)\ \{p\})$	*user-defined recursive functions*
		$\mid$	$p(p)$	*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	$=$	$(\textbf{fun}\,\mathtt{int}\ f\ (\mathtt{int}\ x)\ \{\textbf{if}\ =(x,0)\ \textbf{then}\ 1\ \textbf{else}\ *(x,f(-(x,1)))\})$	factorial
p_5	$=$	$(\textbf{fun}\,\mathtt{int}\ f\ (\mathtt{int}\ x)\ \{f(x)\})(0)$	infinite loop
p_6	$=$	$\textbf{if}\ \mathsf{true}\ \textbf{then}\ 1\ \textbf{else}\ (\textbf{fun}\,\mathtt{int}\ f\ (\mathtt{int}\ x)\ \{f(x)\})(0)$	evaluates to 1
p_7	$=$	$+(\mathsf{true},1)$	type error
p_8	$=$	$(\textbf{fun}\,\mathtt{int}\ f\ (\mathtt{bool}\ x)\ \{x\})$	another type error

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 $\textbf{if}\ \mathsf{true}\ \textbf{then}\ 1\ \textbf{else}\ +(2,\mathsf{false})$ 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:

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

where the derivation C is:

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

$$\text{INT}\ \frac{}{\Gamma \vdash n : \texttt{int}} \qquad \text{TRUE}\ \frac{}{\Gamma \vdash \textsf{true} : \texttt{bool}} \qquad \text{FALSE}\ \frac{}{\Gamma \vdash \textsf{false} : \texttt{bool}}$$

$$\text{VAR}\ \frac{}{\Gamma \vdash x : t}\ \text{if } (x : t) \in \Gamma$$

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

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

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

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

Figure 1: Typing Rules. p ranges over program expressions. For static semantics of program expressions, e ranges over program expressions. For typings of runtime expressions, e ranges over runtime expressions.

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 $\textsf{true}$ 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*
		$\mid$	$\underline{\textsf{true}} \mid \underline{\textsf{false}}$	*boolean values*
		$\mid$	$\langle \underline{\mathit{clos}}(\textbf{fun}\ t\ f\ (t\ x)\ \{p\}), \rho\rangle$	*function values (closures)*
		$\mid$	$\underline{\mathit{DivZero}}$	*error condition for division by zero*
Proper Values	v^*	$::=$	$\underline{n}$	*integer values*
		$\mid$	$\underline{\textsf{true}} \mid \underline{\textsf{false}}$	*boolean values*
		$\mid$	$\langle \underline{\mathit{clos}}(\textbf{fun}\ t\ f\ (t\ x)\ \{p\}), \rho\rangle$	*function values (closures)*
Environments	ρ	$::=$	$\{x_1 = v_1, \cdots, x_n = v_n\}$	
Runtime Expressions	e	$::=$	n	*integer constants*
		$\mid$	x	*variables*
		$\mid$	$o(e, e)$	*primitive applications*
		$\mid$	$\textsf{true} \mid \textsf{false}$	*boolean constants*
		$\mid$	$\textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$	*conditional expressions*
		$\mid$	$(\textbf{fun}\ t\ f\ (t\ x)\ \{p\})$	*user-defined recursive functions*
		$\mid$	$e(e)$	*function applications*
		$\mid$	$\underline{n}$	*integer values*
		$\mid$	$\underline{\textsf{true}} \mid \underline{\textsf{false}}$	*boolean values*
		$\mid$	$\langle \underline{\mathit{clos}}(\textbf{fun}\ t\ f\ (t\ x)\ \{p\}), \rho\rangle$	*function values (closures)*
		$\mid$	$\underline{\mathit{DivZero}}$	*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*
		$\mid$	$o(E, e)$	*evaluate left argument first*
		$\mid$	$o(v^*, E)$	*when done with left argument, go to right*
		$\mid$	$\textbf{if}\ E\ \textbf{then}\ e\ \textbf{else}\ e$	*need the value of the test*
		$\mid$	$E(e)$	*left first*
		$\mid$	$v^*(E)$	*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 $($**fun** $t\ f\ (t\ x)\ \{p\})$,*
  5. *a primitive operation where the first argument is an exception $o($<u>DivZero</u>$, e)$,*
  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* **if** *v* **then** *e_1* **else** *e_2,*
  8. *an application where the function position is an exception* <u>DivZero</u>(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:

$$\begin{array}{rcl}
\langle v, \rho', (E, \rho) : \kappa \rangle & \longmapsto & \langle E[v], \rho, \kappa \rangle
\end{array}$$

$$\begin{array}{rcl}
\langle E[n], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{n}], \rho, \kappa \rangle \\
\langle E[x], \rho, \kappa \rangle & \longmapsto & \langle E[\rho(x)], \rho, \kappa \rangle \text{ if } x \in dom(\rho) \\
\langle E[\textsf{true}], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textsf{true}}], \rho, \kappa \rangle \\
\langle E[\textsf{false}], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textsf{false}}], \rho, \kappa \rangle \\
\langle E[(\textbf{fun}\ t'\ f\ (t\ x)\ \{p\})], \rho, \kappa \rangle & \longmapsto & \langle E[\langle \underline{\textit{clos}}(\textbf{fun}\ t'\ f\ (t\ x)\ \{p\}), \rho \rangle], \rho, \kappa \rangle
\end{array}$$

$$\begin{array}{rcl}
\langle E[+(\underline{n_1}, \underline{n_2})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{n_1 + n_2}], \rho, \kappa \rangle \\
\langle E[*(\underline{n_1}, \underline{n_2})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{n_1 * n_2}], \rho, \kappa \rangle \\
\langle E[-(\underline{n_1}, \underline{n_2})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{n_1 - n_2}], \rho, \kappa \rangle \\
\langle E[/(\underline{n_1}, \underline{n_2})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{n_1 / n_2}], \rho, \kappa \rangle \text{ if } n_2 \neq 0 \\
\langle E[/(\underline{n}, \underline{0})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textit{DivZero}}], \rho, \kappa \rangle \\
\langle E[= (\underline{n}, \underline{n})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textsf{true}}], \rho, \kappa \rangle \\
\langle E[= (\underline{n_1}, \underline{n_2})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textsf{false}}], \rho, \kappa \rangle \text{ if } n_1 \neq n_2 \\
\langle E[< (\underline{n_1}, \underline{n_2})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textsf{true}}], \rho, \kappa \rangle \text{ if } n_1 < n_2 \\
\langle E[< (\underline{n_1}, \underline{n_2})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textsf{false}}], \rho, \kappa \rangle \text{ if } n_1 \geq n_2 \\
\langle E[o(v, \underline{\textit{DivZero}})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textit{DivZero}}], \rho, \kappa \rangle \\
\langle E[o(\underline{\textit{DivZero}}, e)], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textit{DivZero}}], \rho, \kappa \rangle
\end{array}$$

$$\begin{array}{rcl}
\langle E[\textbf{if}\ \underline{\textsf{true}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2], \rho, \kappa \rangle & \longmapsto & \langle E[e_1], \rho, \kappa \rangle \\
\langle E[\textbf{if}\ \underline{\textsf{false}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2], \rho, \kappa \rangle & \longmapsto & \langle E[e_2], \rho, \kappa \rangle \\
\langle E[\textbf{if}\ \underline{\textit{DivZero}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textit{DivZero}}], \rho, \kappa \rangle
\end{array}$$

$$\begin{array}{rcl}
\langle E[cl(v)], \rho, \kappa \rangle & \longmapsto & \langle e, \rho'[x := v, f := cl], (E, \rho) : \kappa \rangle \\
 & & \text{where } cl = \langle \underline{\textit{clos}}(\textbf{fun}\ t'\ f\ (t\ x)\ \{p\}), \rho' \rangle \\
 & & \text{and } v \neq \underline{\textit{DivZero}} \\
\langle E[\underline{\textit{DivZero}}(e)], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textit{DivZero}}], \rho, \kappa \rangle \\
\langle E[\langle \underline{\textit{clos}}(\textbf{fun}\ t'\ f\ (t\ x)\ \{p\}), \rho' \rangle(\underline{\textit{DivZero}})], \rho, \kappa \rangle & \longmapsto & \langle E[\underline{\textit{DivZero}}], \rho, \kappa \rangle
\end{array}$$

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(\mathbf{fun}\ \mathtt{int}\ f\ (\mathtt{int}\ x)\ \{\mathbf{if}\ =(x,0)\ \mathbf{then}\ 1\ \mathbf{else}\ *(x,f(-(x,1)))\})(1),\emptyset,[]\rangle \\
\longmapsto & \langle\langle\underline{\mathit{clos}}(\mathbf{fun}\ \mathtt{int}\ f\ (\mathtt{int}\ x)\ \{\mathbf{if}\ =(x,0)\ \mathbf{then}\ 1\ \mathbf{else}\ *(x,f(-(x,1)))\}),\emptyset\rangle(1),\emptyset,[]\rangle \\
\longmapsto & \langle\langle\underline{\mathit{clos}}\ldots,\emptyset\rangle(\underline{1}),\emptyset,[]\rangle \\
\longmapsto & \langle\mathbf{if}\ =(x,0)\ \mathbf{then}\ 1\ \mathbf{else}\ *(x,f(-(x,1))),\{x=\underline{1},f=\langle\underline{\mathit{clos}}\ldots,\emptyset\rangle\},([],\emptyset):[]\rangle \\
\longmapsto & \langle\mathbf{if}\ =(\underline{1},0)\ \mathbf{then}\ 1\ \mathbf{else}\ *(x,f(-(x,1))),\{x=\underline{1},f=\langle\underline{\mathit{clos}}\ldots,\emptyset\rangle\},([],\emptyset):[]\rangle \\
\longmapsto & \langle\mathbf{if}\ =(\underline{1},\underline{0})\ \mathbf{then}\ 1\ \mathbf{else}\ *(x,f(-(x,1))),\{x=\underline{1},f=\langle\underline{\mathit{clos}}\ldots,\emptyset\rangle\},([],\emptyset):[]\rangle \\
\longmapsto & \langle\mathbf{if}\ \underline{\mathsf{false}}\ \mathbf{then}\ 1\ \mathbf{else}\ *(x,f(-(x,1))),\{x=\underline{1},f=\langle\underline{\mathit{clos}}\ldots,\emptyset\rangle\},([],\emptyset):[]\rangle \\
\longmapsto & \langle*(x,f(-(x,1))),\{x=\underline{1},f=\langle\underline{\mathit{clos}}\ldots,\emptyset\rangle\},([],\emptyset):[]\rangle \\
\longmapsto & \langle*(\underline{1},f(-(x,1))),\{x=\underline{1},f=\langle\underline{\mathit{clos}}\ldots,\emptyset\rangle\},([],\emptyset):[]\rangle \\
\longmapsto & \langle*(\underline{1},\langle\underline{\mathit{clos}}\ldots,\emptyset\rangle(-(x,1))),\{x=\underline{1},f=\langle\underline{\mathit{clos}}\ldots,\emptyset\rangle\},([],\emptyset):[]\rangle \\
\longmapsto & \langle*(\underline{1},\langle\underline{\mathit{clos}}\ldots,\emptyset\rangle(-(\underline{1},1))),\{x=\underline{1},f=\langle\underline{\mathit{clos}}\ldots,\emptyset\rangle\},([],\emptyset):[]\rangle \\
\longmapsto & \langle*(\underline{1},\langle\underline{\mathit{clos}}\ldots,\emptyset\rangle(-(\underline{1},\underline{1}))),\{x=\underline{1},f=\langle\underline{\mathit{clos}}\ldots,\emptyset\rangle\},([],\emptyset):[]\rangle \\
\longmapsto & \langle*(\underline{1},\langle\underline{\mathit{clos}}\ldots,\emptyset\rangle(\underline{0})),\{x=\underline{1},f=\langle\underline{\mathit{clos}}\ldots,\emptyset\rangle\},([],\emptyset):[]\rangle \\
\longmapsto & \langle\mathbf{if}\ =(x,0)\ \mathbf{then}\ 1\ \mathbf{else}\ *(x,f(-(x,1))),\{x=\underline{0},f=\ldots\},(*(\underline{1},[]),\{x=\underline{1},f=\ldots\}):([],\emptyset):[]\rangle \\
\longmapsto & \langle\mathbf{if}\ =(\underline{0},0)\ \mathbf{then}\ 1\ \mathbf{else}\ *(x,f(-(x,1))),\{x=\underline{0},f=\ldots\},(*(\underline{1},[]),\{x=\underline{1},f=\ldots\}):([],\emptyset):[]\rangle \\
\longmapsto & \langle\mathbf{if}\ =(\underline{0},\underline{0})\ \mathbf{then}\ 1\ \mathbf{else}\ *(x,f(-(x,1))),\{x=\underline{0},f=\ldots\},(*(\underline{1},[]),\{x=\underline{1},f=\ldots\}):([],\emptyset):[]\rangle \\
\longmapsto & \langle\mathbf{if}\ \underline{\mathsf{true}}\ \mathbf{then}\ 1\ \mathbf{else}\ *(x,f(-(x,1))),\{x=\underline{0},f=\ldots\},(*(\underline{1},[]),\{x=\underline{1},f=\ldots\}):([],\emptyset):[]\rangle \\
\longmapsto & \langle1,\{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}$$

It is useful to classify the states of the abstract machine in the following way:

Definition 4.1 (Start State, Terminal state, Final state, Stuck state) *Let s be a state.*
s *is* terminal *if there are no transitions from s. Otherwise, it is* non-terminal.
s *is* initial *if it is of the form $\langle p,\emptyset,[]\rangle$.*
s *is* final*, or is a* value state *if it is of the form $\langle v,\rho,[]\rangle$.*
s *is* stuck *if it is terminal but not final.*

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 *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.

5.1 Definitions

We begin by providing the definitions for the type rules of environments, runtime values, and stacks.

Definition 5.1 (Environments match) *A typing environment Γ matches a value environment ρ (written $\Gamma \sim \rho$) if the domains of both environments are identical and for every variable x in that domain, we have that $\emptyset \triangleright \rho(x) : \Gamma(x)$.*

Definition 5.2 (Value typing) *All values are closed and hence can be typed in an empty type environment but the type rules are valid in any environment Γ.*

$$\underline{\text{INT}}\ \frac{}{\Gamma \vdash \underline{n} : \texttt{int}} \qquad \underline{\text{TRUE}}\ \frac{}{\Gamma \vdash \underline{\textit{true}} : \texttt{bool}} \qquad \underline{\text{FALSE}}\ \frac{}{\Gamma \vdash \underline{\textit{false}} : \texttt{bool}}$$

$$\underline{\text{DIVZ}}\ \frac{}{\Gamma \vdash \underline{\textit{DivZero}} : t}\ \textit{for any } t$$

$$\underline{\text{CLOS}}\ \frac{\Gamma', x : t_1, f : t_1 \to t_2 \vdash p : t_2 \qquad \Gamma' \sim \rho'}{\Gamma \vdash \langle \underline{\textit{clos}}(\textbf{\textit{fun}}\ t_r\ f\ (t_x\ x)\ \{p\}), \rho' \rangle : t_1 \to t_2}$$

Note that when we type closures we do not rely on the given signature, but instead rely on the available runtime information. To infer that a closure is of type $t_1 \to t_2$, we must find a typing environment that matches the closure's environment and prove that the body has the right type. The rule for exceptions says that we can assign any type to *<u>DivZero</u>*.

Definition 5.3 (Frame typing) *We view each stack frame as a function that expects a return value called x from the callee and then deliver its own value to its caller.*

$$\frac{\Gamma, x : t_1 \vdash E[x] : t_2 \qquad \Gamma \sim \rho}{(E, \rho) : t_1 \to t_2}$$

Note that the name x cannot occur in Γ. This is not a problem because we can rename variables if there is a clash.

Definition 5.4 (Stack Typing) *The stack is just a sequence of frames where each frame expects a return value from the frame after it and passes it to the frame before it. The entire stack can thus be seen as taking a return value from the current expression and producing the final answer:*

$$\frac{}{[] : t \to t} \text{ for any } t \qquad \frac{(E, \rho) : t_1 \to t \qquad \kappa : t \to t_2}{((E, \rho) : \kappa) : t_1 \to t_2}$$

The final answer can be of any type, which explains the typing we give to the bottom of the stack.

5.2 Little Lemmas

Before getting into the main proof of type safety, we provide without proof some useful auxiliary lemmas. We write $D \triangleright J$ to denote that J is a well-typing judgment whose derivation is D.

Lemma 5.1 (Replacement) *If*

1. $D \triangleright \Gamma \vdash E[e] : t$*, such that the hole in* E *occurs at position* p*,*
2. $D' \triangleright \Gamma \vdash e : t'$*, and*
3. D' *is a subderivation of* D *occurring at position* p*,*
4. $\Gamma \vdash e' : t'$

then, $\Gamma \vdash E[e'] : t$.

This basically says that if we can typecheck an expression like $E[5]$ then we can certainly also typecheck $E[6]$ since 5 and 6 have the same type. The lemma fails if we allow e to be a value. As a counterexample in that case, take $p =$ *DivZero*.

Lemma 5.2 (Substitution) *If*

1. x *is not free in* E*,*
2. $\Gamma, x : t \vdash E[x] : t'$ *and*
3. $\Gamma \vdash e : t$*,*

then $\Gamma \vdash E[e] : t'$*. Conversely, if*

1. x *does not occur in* $dom(\Gamma)$ *or in* $FV(E[e])$

2. *$D \triangleright \Gamma \vdash E[e] : t$, such that the hole in E occurs at position p, and*

3. *$D' \triangleright \Gamma \vdash e : t'$, and*

4. *D' is a subderivation of D occurring at position p,*

then $\Gamma, x : t' \vdash E[x] : t$.

This is almost the same as the previous lemma. It says that if we can typecheck $E[x]$ under the assumption that x is an `int`, then we can certainly typecheck $E[5]$ where we have replaced x by something of the same type, and vice versa.

Lemma 5.3 (Environment Extension) *If $\Gamma \vdash e : t$ and $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash e : t$.*

This lemma says that adding more “junk” variables to the environment doesn’t affect typing. So if we can prove that $\emptyset \vdash 5 : \texttt{int}$ then we can also prove that $\{x : \texttt{bool}\} \vdash 5 : \texttt{int}$.

Lemma 5.4 (Environment Contraction) *If $\Gamma, x : t \vdash e : t$ and $x \notin FV(e)$, then $\Gamma \vdash e : t$.*

This lemma is the converse of Lemma 5.3. It says that if a variable doesn’t occur free in an expression, omitting it from the set of type assumptions doesn’t affect the expression’s typing. So, since we can prove $\emptyset, x : \texttt{int} \vdash 5 : \texttt{int}$, we can also prove $\emptyset \vdash 5 : \texttt{int}$.

Lemma 5.5 (Subterm typing) *If $\Gamma \vdash E[e] : t$ then $\Gamma \vdash e : t'$*

In other words, if we type a term, then all its subterms must have types. The environment does not change in the statement since the evaluation context E does not bind any variables.

Lemma 5.6 (Inversion Lemma)

- *If $\Gamma \vdash n : t$, then $t = \texttt{int}$.*
- *If $\Gamma \vdash \textsf{true} : t$, then $t = \texttt{bool}$.*
- *If $\Gamma \vdash \textsf{false} : t$, then $t = \texttt{bool}$.*
- *If $\Gamma \vdash x : t$, then $\Gamma(x) = t$.*
- *If $\Gamma \vdash o(e_1, e_2) : t$ where o is one of $\{=, <\}$, then $t = \texttt{bool}$ and $\Gamma \vdash e_1 : \texttt{int}$ and $\Gamma \vdash e_2 : \texttt{int}$.*
- *If $\Gamma \vdash o(e_1, e_2) : t$ where o is one of $\{+, -, *, /\}$, then $t = \texttt{int}$ and $\Gamma \vdash e_1 : \texttt{int}$ and $\Gamma \vdash e_2 : \texttt{int}$.*
- *If $\Gamma \vdash$ **if** e_1 **then** e_2 **else** $e_3 : t$, then $\Gamma \vdash e_1 : \texttt{bool}$ and $\Gamma \vdash e_2 : t$ and $\Gamma \vdash e_3 : t$.*
- *If $\Gamma \vdash$ (**fun** $t_r\ f\ (t_x\ x)\ \{p\}) : t$, then $t = t_x \to t_r$ and $\Gamma, x : t_x, f : t_x \to t_r \vdash p : t_r$.*

- *If* $\Gamma \vdash e_1(e_2) : t$*, then there exists a* t_2 *such that* $\Gamma \vdash e_1 : t_2 \to t$ *and* $\Gamma \vdash e_2 : t_2$*.*
- *If* $\Gamma \vdash \underline{n} : t$*, then* $t = \texttt{int}$*.*
- *If* $\Gamma \vdash \underline{\textsf{true}} : t$*, then* $t = \texttt{bool}$*.*
- *If* $\Gamma \vdash \underline{\textsf{false}} : t$*, then* $t = \texttt{bool}$*.*
- *If* $\Gamma \vdash \langle \underline{\textit{clos}}(\textbf{\textit{fun}}\ t_r\ f\ (t_x\ x)\ \{p\}), \rho' \rangle : t$*, then there exist* $\Gamma' \sim \rho'$ *and* t_1 *and* t_2 *such that* $t = t_1 \to t_2$ *and* $\Gamma', x : t_1, f : t_1 \to t_2 \vdash p : t_2$*.*

Lemma 5.7 (Canonical Forms Lemma) *Suppose that* $\emptyset \vdash v : t$*:*

- *If* $t = \texttt{int}$*, then* $v = \underline{n}$ *for some* n *or* $v = \underline{\textit{DivZero}}$*.*
- *If* $t = \texttt{bool}$*, then* $v = \underline{\textsf{true}}$ *or* $v = \underline{\textsf{false}}$ *or* $v = \underline{\textit{DivZero}}$*.*
- *If* $t = t_1 \to t_2$*, then* $v = \langle \underline{\textit{clos}}(\textbf{\textit{fun}}\ t_r\ f\ (t_x\ x)\ \{p\}), \rho \rangle$ *for some* t_r*,* f*,* t_x*,* x*,* p*, and* ρ*, or* $v = \underline{\textit{DivZero}}$*.*

In other words, the type of a value predicts its form.

5.3 Main Lemma and Theorem

The most interesting thing here is the definition of what constitutes a safe state, and Lemma 5.9 which describes the evolution of safe states. Given Lemma 5.9, the actual proof of type safety (Theorem 5.8) is rather straightforward.

Definition 5.5 (Safe state) *A state* $s = \langle e, \rho, k \rangle$ *is* safe with type t_f *if there exist* Γ *and* t *such that*

1. $\Gamma \vdash e : t$
2. $\Gamma \sim \rho$
3. $k : t \to t_f$

We write $s : t_f$ *to denote that* s *is safe with type* t_f*.* s *is* safe *if* $s : t_f$ *for some* t_f*. Otherwise* s *is* unsafe.

Theorem 5.8 (Type Safety) *If* $\emptyset \vdash p : t_f$*,*

$$\langle p, \emptyset, [] \rangle \longmapsto^* s_f$$

and s_f *is a terminal state, then* s_f *is a final state* $\langle v, \rho, [] \rangle$*, where* v *is a value of type* t_f*.*

Note that our definition of values includes the *DivZero* exception which has every type.

Proof. From Lemma 5.9, it follows by induction on number of transitions that if $s : t$ and $s \longmapsto^* s'$, then $s' : t$ and s' is not stuck.

Since $\langle p, \emptyset, [] \rangle : t_f$, and $\langle p, \emptyset, [] \rangle \longmapsto^* s_f$, it follows that s_f is not stuck and $s : t_f$. Since s_f is terminal and not stuck, s_f is a final state $\langle v, \rho, [] \rangle$. Since $s_f : t_f$, v has type t_f.

Lemma 5.9 (Progress and Subject Reduction Lemma) *Let $s : t_f$. Then,*

- **Progress:** *s is not stuck, i.e., s is either final or non-terminal.*
- **Subject Reduction:** *If $s \longmapsto s'$ then $s' : t_f$.*

Proof.

Let $s = \langle e, \rho, t \rangle$ and let Γ, t be such that

- $\Gamma \vdash e : t$,
- $\Gamma \sim \rho$, and
- $\kappa : t \rightarrow t_f$,

The proof is by case analysis on e using the unique decomposition lemma. In each case, we verify that s is either final or can be updated to a state $s' = \langle e', \rho', \kappa' \rangle$. We verify that $s' : t_f$ by constructing Γ' and t' such that

- $\Gamma' \vdash e' : t'$,
- $\Gamma' \sim \rho'$, and
- $\kappa' : t' \rightarrow t_f$.

Case analysis on e:

- Case e is a value v. We are given $\Gamma \vdash v : t$, $\Gamma \sim \rho$, and $\kappa : t \rightarrow t_f$. If κ is empty then we have a final state and we are done. Else we have $\kappa = (E_1, \rho_1) : \kappa_1$ and the state $\langle v, \rho, (E_1, \rho_1) : \kappa_1 \rangle$ can be updated to $\langle E_1[v], \rho_1, \kappa_1 \rangle$. Because we know that $\kappa : t \rightarrow t_f$, we know that there exists a t' such that $(E_1, \rho_1) : t \rightarrow t'$ and $\kappa_1 : t' \rightarrow t_f$. Because $(E_1, \rho_1) : t \rightarrow t'$, there must exist a Γ_1 such that $\Gamma_1 \sim \rho_1$ and where $\Gamma_1, x : t \vdash E_1[x] : t'$. The result then follows easily using the substitution lemma since v has type t in any environment.
- Case e is $E[n]$. We are given $\Gamma \vdash E[n] : t$, $\Gamma \sim \rho$, and $\kappa : t \rightarrow t_f$. By an application of the subterm typing lemma followed by an application of the inversion lemma $\Gamma \vdash n : \texttt{int}$. The state $\langle E[n], \rho, \kappa \rangle$ can be updated to $\langle E[\underline{n}], \rho, \kappa \rangle$. Since $\Gamma \vdash \underline{n} : \texttt{int}$, we can conclude by the replacement lemma that $\Gamma \vdash E[\underline{n}] : t$.
- Case e is $E[\mathsf{true}]$ or $E[\mathsf{false}]$. These cases are almost identical to the previous case.

- Case e is $E[x]$. We are given $\Gamma \vdash E[x] : t$, $\Gamma \sim \rho$, and $\kappa : t \to t_f$. By the subterm typing lemma and the inversion lemma, the variable x must be in the domain of Γ. Let $t_x = \Gamma(x)$ then we have $\Gamma \vdash x : t_x$. Because $\Gamma \sim \rho$, the variable x must also be in the domain of ρ. This means that it is possible to update the state $\langle E[x], \rho, \kappa\rangle$ to $\langle E[\rho(x)], \rho, \kappa\rangle$. Since $\Gamma \sim \rho$, we know that $\emptyset \vdash \rho(x) : t_x$. By the environment extension lemma $\Gamma \vdash \rho(x) : t_x$ and by the replacement lemma $\Gamma \vdash E[\rho(x)] : t$.

- Case e is $E[(\textbf{fun}\, t_r\ f\ (t_x\ x)\ \{b\})]$. We know that $\Gamma \vdash E[(\textbf{fun}\, t_r\ f\ (t_x\ x)\ \{b\})] : t$, $\Gamma \sim \rho$, and $\kappa : t \to t_f$. By the subterm typing lemma and the inversion lemma, $\Gamma \vdash (\textbf{fun}\, t_r\ f\ (t_x\ x)\ \{b\}) : t_x \to t_r$. Hence $\Gamma, x : t_x, f : t_x \to t_r \vdash b : t_r$. The state $\langle E[(\textbf{fun}\, t_r\ f\ (t_x\ x)\ \{b\})], \rho, \kappa\rangle$ can be updated to $\langle E[\langle \underline{\mathit{clos}}(\textbf{fun}\, t_r\ f\ (t_x\ x)\ \{b\}), \rho\rangle], \rho, \kappa\rangle$. We need to show that $\Gamma \vdash E[\langle \underline{\mathit{clos}}(\textbf{fun}\, t_r\ f\ (t_x\ x)\ \{b\}), \rho\rangle] : t$. Given that $\Gamma, x : t_x, f : t_x \to t_r \vdash b : t_r$ and that $\Gamma \sim \rho$, it follows from the definition of value typing that $\emptyset \vdash \langle \underline{\mathit{clos}}(\textbf{fun}\, t_r\ f\ (t_x\ x)\ \{b\}), \rho\rangle : t_x \to t_r$. The result follows using the environment extension lemma and then the replacement lemma.

- Case $e = E[o(\underline{\mathit{DivZero}}, e_1)]$. The state can be updated to $\langle E[\underline{\mathit{DivZero}}], \rho, \kappa\rangle$ and the result follows.

- Case $e = E[o(v_1^*, v_2)]$. We only consider the case of division; the other cases are similar and somewhat simpler since they do not themselves raise exceptions. We are given $\Gamma \vdash E[/(v_1^*, v_2)] : t$. By the subterm typing lemma and the inversion lemma, we get $\Gamma \vdash /(v_1^*, v_2) : \texttt{int}$. This implies that $\Gamma \vdash v_1^* : \texttt{int}$ and $\Gamma \vdash v_2 : \texttt{int}$. By the canonical forms lemma, the value v_1 must be an integer value and v_2 is either an integer value or an exception. We proceed by cases:

  - $v_1 = \underline{n_1}$, $v_2 = \underline{n_2} \neq \underline{0}$. The state $\langle E[/(\underline{n_1}, \underline{n_2})], \rho, \kappa\rangle$ can be updated to $\langle E[\underline{n_1/n_2}], \rho, \kappa\rangle$. The result follows easily.
  - $v_1 = \underline{n_1}$, $v_2 = \underline{0}$ or $v_2 = \underline{\mathit{DivZero}}$. The state $\langle E[/(\underline{n_1}, v_2)], \rho, \kappa\rangle$ can be updated to the state $\langle E[\underline{\mathit{DivZero}}], \rho, \kappa\rangle$ and the result also follows easily since $\underline{\mathit{DivZero}}$ can have any type including `int`.

- Case $e = E[\textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2]$. Following the previous cases, we first conclude that v has type `bool` and using the canonical forms lemma, consider the three possible values of this type: $\underline{\mathsf{true}}$, $\underline{\mathsf{false}}$, or $\underline{\mathit{DivZero}}$. The result follows for each case.

- Case $e = E[\underline{\mathit{DivZero}}(e)]$. Easy.

- Case $e = E[v_1^*(v_2)]$. Following the previous cases, we first show that $\Gamma \vdash v_1^* : t_2 \to t_1$ and $\Gamma \vdash v_2 : t_2$. If v_2 is an exception, we are done as before. Otherwise, the canonical forms lemma ensures that v_1 is a closure of the form $\langle \underline{\mathit{clos}}(\textbf{fun}\ t_r\ f\ (t_x\ x)\ \{p_c\}), \rho_c\rangle$. Because we know this closure has type $t_2 \to t_1$, we conclude that there exists a $\Gamma_c \sim \rho_c$ such that $\Gamma_c, x : t_2, f : t_2 \to t_1 \vdash p_c : t_1$. The state can be updated to $\langle p_c, \rho_c[x := v_2, f := v_1], (E, \rho) : \kappa\rangle$ and it suffices to prove that $\Gamma_c, x : t_2, f : t_2 \to t_1 \sim \rho_c[x := v_2, f := v_1]$ and that $((E, \rho) : \kappa) : t_1 \to t_f$. The first statement is immediate. To prove the second we need to show that $(E, \rho) : t_1 \to t$ which we can prove if we know that $\Gamma, z : t_1 \vdash E[z] : t$. This follows from the substitution lemma.

Acknowledgments

Thanks to Todd Veldhuizen for correcting the definition of evaluation contexts to only use proper values, and thanks for Venkatesh Choppella for correcting the statements of the substitution and replacement lemmas, and for extensive comments.

References

[1] HARPER, R. *Programming Languages: Theory and Practice.* Draft, 2001.

[2] STÄRK, R. F., SCHMID, J., AND BÖRGER, E. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001.

A AsmGofer Implementation

```
{--

File: minML.gs

A small typed functional language and its formalization in AsmGofer...

by Amr Sabry (based on Ch. 3-5 of Programming Languages: Theory and
   Practice, Robert Harper, Draft of Dec. 2000 and using the style
   developed for the JBook)

--}

-------------------------------------------------------------------------------
-- Annotated abstract syntax trees:
--  * initially the AST has no types and only uses syntactic constructors
--  * after typechecking the nodes are decorated with types
--  * during evaluation syntactic constructors are replaced by semantic values
```

```
-------------------------------------------------------------------------------

data MLType = T_Int | T_Bool | T_Arrow MLType MLType
instance AsmTerm MLType

data MLOp = O_Plus | O_Times | O_Minus | O_Div | O_Equal | O_LessThan
instance AsmTerm MLOp

data MLTerm a = Term a [MLTerm a]
instance AsmTerm a => AsmTerm (MLTerm a)

data MLForm =                            -- a
  -- Syntactic forms
    S_Num Int                            -- []
  | S_Var String                         -- []
  | S_Prim MLOp                          -- [MLTerm a, MLTerm a]
  | S_True                               -- []
  | S_False                              -- []
  | S_If                                 -- [MLTerm a, MLTerm a, MLTerm a]
  | S_Fun MLType String (String,MLType)  -- [MLTerm a]
  | S_App                                -- [MLTerm a, MLTerm a]
  -- Runtime values or errors
  | R_Val MLValue                        -- []
  | R_DivZero                            -- []
instance AsmTerm MLForm

type MLExp = MLTerm MLForm
type TypedMLExp = MLTerm (MLForm,MLType)

data MLValue = V_Num Int | V_True | V_False | V_Closure MLExp VEnv
instance AsmTerm MLValue

type VEnv = [(String,MLValue)]

------------------------------------------------------------------------------
-- Static semantics:
--   * typechecking takes an AST and returns another AST with all the types
--     or aborts if there is an error
-------------------------------------------------------------------------------

type TEnv = [(String,MLType)]

tlookup :: String -> TEnv -> MLType
tlookup v [] = error ("Typechecking: unbound variable " ++ v)
tlookup v ((s,t):r) = if v == s then t else tlookup v r
```

```
typeOf :: TypedMLExp -> MLType
typeOf (Term (_,t) _) = t

typecheck :: MLExp -> TEnv -> TypedMLExp
typecheck exp tenv =
  case exp of

    Term (S_Num i) [] -> Term (S_Num i, T_Int) []

    Term (S_Var v) [] -> Term (S_Var v, tlookup v tenv) []

    Term (S_Prim b) [e1,e2] | b == O_Equal || b == O_LessThan ->
      let e1’ = typecheck e1 tenv
          e2’ = typecheck e2 tenv
          rt = case (typeOf e1’, typeOf e2’) of
                 (T_Int,T_Int) -> T_Bool
                 (t1,t2) ->
                   error ("Typechecking: operator <"
                   ++ show b ++ "> requires operands of type int; found <"
                   ++ show t1 ++ "> and <" ++ show t2 ++ ">")
      in Term (S_Prim b, rt) [e1’,e2’]

    Term (S_Prim b) [e1,e2] | b == O_Plus || b == O_Minus ||
                              b == O_Times || b == O_Div ->
      let e1’ = typecheck e1 tenv
          e2’ = typecheck e2 tenv
          rt = case (typeOf e1’, typeOf e2’) of
                 (T_Int,T_Int) -> T_Int
                 (t1,t2) ->
                   error ("Typechecking: operator <"
                   ++ show b ++ "> requires operands of type int; found <"
                   ++ show t1 ++ "> and <" ++ show t2 ++ ">")
      in Term (S_Prim b, rt) [e1’,e2’]

    Term S_True [] -> Term (S_True, T_Bool) []

    Term S_False [] -> Term (S_False, T_Bool) []

    Term S_If [e1,e2,e3] ->
      let e1’ = typecheck e1 tenv
          e2’ = typecheck e2 tenv
          e3’ = typecheck e3 tenv
          rt = if typeOf e1’ == T_Bool && typeOf e2’ == typeOf e3’
               then typeOf e2’
               else error ("Typechecking: if requires a boolean and "
               ++ "two expressions of the same type; found <"
               ++ show (typeOf e1’) ++ ">, <"
               ++ show (typeOf e2’) ++ ">, and <"
               ++ show (typeOf e3’) ++ ">")
      in Term (S_If,rt) [e1’,e2’,e3’]

    Term (S_Fun rt fn (pn,pt)) [e] ->
      let tenv’ = (fn, T_Arrow pt rt) : (pn, pt) : tenv
          e’ = typecheck e tenv’
          ct = if typeOf e’ == rt
               then T_Arrow pt rt
               else error ("Typechecking: declared return type <" ++ show rt
                           ++ "> does not agree with actual return type <"
                           ++ show (typeOf e’) ++ ">")
      in Term (S_Fun rt fn (pn,pt), ct) [e’]

    Term S_App [e1,e2] ->
      let e1’ = typecheck e1 tenv
          e2’ = typecheck e2 tenv
          ct = case (typeOf e1’, typeOf e2’) of
                 (T_Arrow t2 t, t2’) | t2 == t2’ -> t
                 (t1,t2) ->
                   error ("Typechecking: attempting to apply a "
                   ++ "function of type <"
                   ++ show t1 ++ "> to an argument of type <"
                   ++ show t2 ++ ">")
      in Term (S_App,ct) [e1’,e2’]

    exp -> error ("Typecheck: unexpected expression " ++ show exp)

-------------------------------------------------------------------------------
-- Positions
-------------------------------------------------------------------------------

type Pos = [Int]

up :: Pos -> Pos
up [] = [] -- NOT an error (used to return from top level evaluation)
up ds = init ds

firstPos :: Pos
firstPos  = []

down :: (Pos,Int) -> Pos
down (ds,d) = ds ++ [d]

-- During evaluation we replace syntactic constructors by dynamic values or
-- runtime errors
substMLExp :: (MLExp, MLExp, Pos) -> MLExp
substMLExp (e, (Term _ _), []) = e
substMLExp (e, (Term a ts), p:ps) =
  let (lts,rt:rts) = splitAt p ts
  in Term a (lts ++ [ substMLExp(e,rt,ps) ] ++ rts)
substMLExp (e1, e2, p) =
  error ("substMLExp: unexpected arguments "
         ++ show e1 ++ ", "
         ++ show e2 ++ ", and "
         ++ show p)

-- context takes an expression and a position and returns
-- the subexpression at the position
context :: (MLExp, Pos) -> MLExp
context (e,[]) = e
context (Term _ es, i:is) = context (es!!i, is)
context (e,p) = error ("context: unexpected expression and position "
                        ++ show e ++ " and "
                        ++ show p)

-------------------------------------------------------------------------------
-- ASM states
-------------------------------------------------------------------------------

code :: Dynamic MLExp
code = initVal "code" asmDefault

pos :: Dynamic Pos
pos = initVal "pos" firstPos

env :: Dynamic VEnv
env = initVal "env" []

type SFrame = (MLExp,Pos,VEnv)

stack :: Dynamic [SFrame]
stack = initVal "stack" []

initialize :: MLExp -> IO ()
initialize e =
  fire1 (do code := e
            pos := firstPos
            env := []
            stack := [])

-------------------------------------------------------------------------------
-- Evaluation:
--  * proceeds by finding a position where we can evaluate
--  * performs the evaluation
--  * replaces the constructor at the position with the value
-------------------------------------------------------------------------------

vlookup :: String -> VEnv -> MLValue
vlookup v [] = error ("vlookup: unexpected unbound variable" ++ v)
vlookup v ((s,va):r) = if v == s then va else vlookup v r

execML :: Rule ()
execML =
  case context (code,pos) of

    Term (S_Num i) [] -> yield (Term (R_Val (V_Num i)) [])

    Term (S_Var v) [] -> if v ‘elem‘ map fst env
                         then yield (Term (R_Val (vlookup v env)) [])
                         else skip

    Term (S_Prim op) [Term (R_Val v1) [], Term (R_Val v2) []] ->
      if op == O_Div && v2 == V_Num 0
      then yield (Term R_DivZero [])
      else yield (Term (applyOp op v1 v2) [])

    Term (S_Prim op) [Term (R_Val v1) [], Term R_DivZero []] ->
      yield (Term R_DivZero [])

    Term (S_Prim op) [Term (R_Val v1) [], e2] -> pos := down (pos,1)

    Term (S_Prim op) [Term R_DivZero [], e2] -> yield (Term R_DivZero [])

    Term (S_Prim op) [e1,e2] -> pos := down (pos,0)

    Term S_True [] -> yield (Term (R_Val V_True) [])

    Term S_False [] -> yield (Term (R_Val V_False) [])

    Term (S_If) [Term R_DivZero [], e2, e3] -> yield (Term R_DivZero [])
    Term (S_If) [Term (R_Val v1) [], e2, e3] ->
      case v1 of
        V_True -> yield e2
        V_False -> yield e3
        _ -> skip

    Term (S_If) [e1,e2,e3] -> pos := down (pos,0)

    Term (S_Fun rt fn (pn,pt)) [e] ->
      yield (Term (R_Val (V_Closure (Term (S_Fun rt fn (pn,pt)) [e]) env)) [])

    Term S_App [Term (R_Val v1) [], Term R_DivZero []] ->
      yield (Term R_DivZero [])

    Term S_App [Term (R_Val v1) [], Term (R_Val v2) []] ->
      case v1 of
        V_Closure (Term (S_Fun rt fn (pn,pt)) [e]) lenv ->
          do stack := (code,pos,env) : stack
             code := e
             pos := firstPos
             env := (pn,v2) : (fn,v1) : lenv
        _ -> skip

    Term S_App [Term R_DivZero [], e2] -> yield (Term R_DivZero [])

    Term S_App [Term (R_Val v) [], e2] -> pos := down (pos,1)

    Term S_App [e1,e2] -> pos := down (pos,0)

    Term (R_Val v) ts ->
      case stack of
        [] -> skip
        (c1,p1,e1):s1 -> do code := substMLExp (Term (R_Val v) ts, c1, p1)
                            pos := up p1
                            env := e1
                            stack := s1

    Term R_DivZero [] ->
      case stack of
        [] -> skip
        (c1,p1,e1) : s1 -> do code := substMLExp (Term R_DivZero [], c1, p1)
                              pos := up p1
                              env := e1
                              stack := s1
    e -> error ("execML: unexpected expression " ++ show e)

yield :: MLExp -> Rule ()
yield result = do
   code := substMLExp (result , code , pos)
   pos := up pos

-------------------------------------------------------------------------------
-- Evaluation
-------------------------------------------------------------------------------

eval :: MLExp -> IO ()
eval e = do putStr "-----------------------------------------\nTypechecking..."
            putStr "\nExpression: "
            print e
            putStr "has type: "
            print (typeOf (typecheck e []))
            putStr "-----------------------------------------\nEvaluating...\n"
            initialize e
            fixpoint (trace printState execML)
            printValue

applyOp :: MLOp -> MLValue -> MLValue -> MLForm
applyOp O_Plus (V_Num i1) (V_Num i2) = R_Val (V_Num (i1+i2))
applyOp O_Times (V_Num i1) (V_Num i2) = R_Val (V_Num (i1*i2))
applyOp O_Minus (V_Num i1) (V_Num i2) = R_Val (V_Num (i1-i2))
applyOp O_Div (V_Num i1) (V_Num i2) = R_Val (V_Num (i1/i2))
applyOp O_Equal (V_Num i1) (V_Num i2) =
  R_Val (if i1 == i2 then V_True else V_False)
applyOp O_LessThan (V_Num i1) (V_Num i2) =
  R_Val (if i1 < i2 then V_True else V_False)
applyOp op v1 v2 =
  error ("applyOp: unexpected operator and values"
  ++ show op ++ ", " ++ show v1 ++ ", and " ++ show v2)

printState :: IO ()
printState = do putStr "<code = "
                print code
                putStr ",pos  = "
                print pos
                putStr ",env  = "
                print env
                putStr ",stack size = "
                print (length stack)
                putStr ">\n---------------------\n"
```

```
printValue :: IO ()
printValue = do putStr "VALUE = "
                case code of
                  Term (R_Val v) [] -> print v
                  Term R_DivZero [] -> putStr "Exception: division by zero"
                  _ -> error ("Unexpected value"  ++ show code)

-------------------------------------------------------------------------------
-- Examples
-------------------------------------------------------------------------------

numE e = Term (S_Num e) []
varE e = Term (S_Var e) []
trueE = Term S_True []
falseE = Term S_False []
primE b e1 e2 = Term (S_Prim b) [e1,e2]
addE e1 e2 = primE O_Plus e1 e2
divE e1 e2 = primE O_Div e1 e2
lessE e1 e2 = primE O_LessThan e1 e2
ifE e1 e2 e3 = Term S_If [e1,e2,e3]
funE rt fn (pn,pt) b = Term (S_Fun rt fn (pn,pt)) [ b ]
appE e1 e2 = Term S_App [e1,e2]

t1 = addE (addE (numE 1) (numE 2)) (addE (numE 3) (numE 4))
t2 = addE (addE (numE 1) (numE 2)) (addE (divE (numE 1) (numE 0)) (numE 4))
t3 = appE (funE T_Int "f" ("x",T_Int) (appE (varE "f") (varE "x"))) (numE 0)
t4 = ifE trueE (numE 1) t3
t5 = funE T_Int "f" ("x",T_Int)
       (ifE (primE O_Equal (varE "x") (numE 0))
            (numE 1)
            (primE O_Times (varE "x")
              (appE (varE "f")
                    (primE O_Minus (varE "x") (numE 1)))))
t6 = appE t5 (numE 5) -- factorial of 5
t7 = funE T_Int "f" ("x",T_Int)
       (ifE (primE O_Equal (varE "x") (numE 0))
            (numE 1)
            (primE O_Times
              (appE (varE "f")
                    (primE O_Minus (varE "x") (numE 1))
              (varE "x")))
t8 = appE t7 (numE 5) -- checking environment after popping stack

t9 = divE (numE 1) (numE 0)
```

```
t10 = lessE t9 (numE 0) -- a boolean DivZero
t11 = ifE t10 t5 t5 -- an int->int divZero

t12 = appE t11 (numE 0)

-------------------------------------------------------------------------------
-- Printing
-------------------------------------------------------------------------------

showSepBy :: String -> [ShowS] -> ShowS
showSepBy _ []       = id
showSepBy _ [x]      = x
showSepBy sep (x:xs) = x . showString sep . showSepBy sep xs

instance Text MLType where
  showsPrec _ T_Int = showString "int"
  showsPrec _ T_Bool = showString "bool"
  showsPrec _ (T_Arrow t1 t2) = shows t1 . showString " -> " . shows t2
  showsPrec _ t = error ("show: unexpected type " ++ show t)

instance Text MLOp where
  showsPrec _ O_Plus = showString "+"
  showsPrec _ O_Times = showString "*"
  showsPrec _ O_Minus = showString "-"
  showsPrec _ O_Div = showString "/"
  showsPrec _ O_Equal = showString "=="
  showsPrec _ O_LessThan = showString "<"
  showsPrec _ op = error ("show: unexpected operator " ++ show op)

instance Text a => Text (MLTerm a) where
  showsPrec _ (Term e []) = shows e
  showsPrec _ (Term e es) =
    shows e . showString "{" . showSepBy " , " (map shows es) . showString "}"

instance Text MLForm where
  showsPrec _ (S_Num i) = shows i
  showsPrec _ (S_Var v) = showString v
  showsPrec _ (S_Prim bop) = shows bop
  showsPrec _ S_True = showString "true"
  showsPrec _ S_False = showString "false"
  showsPrec _ S_If = showString "if"
  showsPrec _ (S_Fun rt fn (pn,pt)) =
    shows rt .
    showString (" " ++ fn ++ " (" ++ pn) .
    showString ":" .
    shows pt .
    showString ") "
  showsPrec _ S_App = showString "@"
  showsPrec _ (R_Val v) = shows v
  showsPrec _ R_DivZero = showString "DivZero"
  showsPrec _ c  = error ("show: unexpected form " ++ show c)

instance Text MLValue where
  showsPrec _ (V_Num i) = shows i
  showsPrec _ V_True = showString "true"
  showsPrec _ V_False = showString "false"
  showsPrec _ (V_Closure exp env) = showString "<closure>"
  showsPrec _ v = error ("show: unexpected value " ++ show v)

-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
```