Type Safety for NanoML

Assignment 4

Due: February 25, 2003

1 Syntax of NanoML

The syntax is given by the following BNF. In the following n ranges over integer constants:

Types	t	::=	`int`	*integer type*
		|	`bool`	*boolean type*
Operators	o	::=	$+ \mid - \mid * \mid / \mid = \mid <$	
Expressions	e	::=	n	*integer constants*
		|	$o(e, e)$	*primitive applications*
		|	true | false	*boolean constants*
		|	**if** e **then** e **else** e	*conditional expressions*

2 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 4) 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.

3 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.

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.

$$\frac{}{\vdash n : \texttt{int}} \qquad \frac{}{\vdash \mathsf{true} : \texttt{bool}} \qquad \frac{}{\vdash \mathsf{false} : \texttt{bool}}$$

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

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

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

Figure 1: Typing Rules

Values	v	::=	$\underline{n}$	*integer values*
		|	$\underline{\mathsf{true}} \mid \underline{\mathsf{false}}$	*boolean values*
		|	$\underline{\mathit{DivZero}}$	*error condition for division by zero*
Proper Values	v^*	::=	$\underline{n}$	*integer values*
		|	$\underline{\mathsf{true}} \mid \underline{\mathsf{false}}$	*boolean values*
Runtime Expressions	e	::=	n	*integer constants*
		|	$o(e, e)$	*primitive applications*
		|	$\mathsf{true} \mid \mathsf{false}$	*boolean constants*
		|	**if** e **then** e **else** e	*conditional expressions*
		|	$\underline{n}$	*integer values*
		|	$\underline{\mathsf{true}} \mid \underline{\mathsf{false}}$	*boolean values*
		|	$\underline{\mathit{DivZero}}$	*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*
		|	$o(E, e)$	*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*

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 3.1 (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 boolean constant* true *or* false*,*

3. *a primitive operation where the first argument is an exception* $o(\underline{\textit{DivZero}}, e)$,
4. *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,*
5. *a conditional expression with an evaluated test position* ***if*** v ***then*** e_1 ***else*** e_2,

The evaluation rules are:

$$
\begin{array}{rcl}
E[n] & \longmapsto & E[\underline{n}] \\
E[\mathsf{true}] & \longmapsto & E[\underline{\mathsf{true}}] \\
E[\mathsf{false}] & \longmapsto & E[\underline{\mathsf{false}}]
\end{array}
$$

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

$$
\begin{array}{rcl}
E[\textbf{if } \underline{\mathsf{true}} \textbf{ then } e_1 \textbf{ else } e_2] & \longmapsto & E[e_1] \\
E[\textbf{if } \underline{\mathsf{false}} \textbf{ then } e_1 \textbf{ else } e_2] & \longmapsto & E[e_2] \\
E[\textbf{if } \underline{\textit{DivZero}} \textbf{ then } e_1 \textbf{ else } e_2] & \longmapsto & E[\underline{\textit{DivZero}}]
\end{array}
$$

4 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 $\underline{\textit{DivZero}}$, 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. To relate the initial program and its type to the value v^*, it is natural to proceed one transition at a time. If we can prove that each transition preserves the type of the expression, and that well-typed states can always make progress until they become final states, then we can conclude our main result by induction on the number of transitions. This is the gist of our proof technique.

Here is the statement of the proof of type safety. Prove the theorem as well as any supporting lemmas you might need.

Theorem 4.1 (Type Safety) *If* $\vdash e : t_f$ *and* $e \longmapsto^* v$, *then* v *is of type* t_f. *(Note that our definition of values includes the* $\underline{\textit{DivZero}}$ *exception which has every type.)*