Lectures Notes on Progress

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 7
September 21, 2004

In this lecture we prove the progress property for MinML, discuss type safety, and consider which other language features may be desirable or undesirable in a language definition. As a reminder, type safety consists of preservation (proved in the last lecture) and progress in the following form.

1. (Preservation) If $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau$
2. (Progress) If $\cdot \vdash e : \tau$ then either
   (i) $e \mapsto e'$ for some e', or
   (ii) e value
3. (Determinism) If $\cdot \vdash e : \tau$ and $e \mapsto e'$ and $e \mapsto e''$ then $e' = e''$.

Determinism is of particular interest for sequential languages, where we generally expect it to hold.

Not all these properties are of equal importance, and we may have perfectly well-designed languages in which some of these properties fail. However, we want to clearly classify languages based on these properties and understand if they hold, or fail to hold. Please consult the notes of the previous lecture for a further discussion of some of these issues.

Progress. We now turn our attention to the progress theorem. This asserts that the computation of closed well-typed expressions will never get stuck, although it is quite possible that it does not terminate. For example,

$$\texttt{rec}(\texttt{int}, x.x)$$

reduces in one step to itself.

The critical observation behind the proof of the progress theorem is that a value of function type will indeed be a function, a value of boolean type will indeed by either `true` or `false`, etc. If that were not the case, then we might reach an expression such as

$$\texttt{apply}(\texttt{num}(0), \texttt{num}(1))$$

which is a stuck expression because $\texttt{num}(0)$ and $\texttt{num}(1)$ are values, so neither any of the search rules nor the reduction rule for application can be applied. We state these critical properties as an inversion lemmas, because they are not immediately syntactically obvious.

Lemma 1 (Value Inversion)

(i) If $\cdot \vdash v : \texttt{int}$ *and* v value *then* $v = \texttt{num}(n)$ *for some integer* n*.*

(ii) If $\cdot \vdash v : \texttt{bool}$ *and* v value *then* $v = \texttt{true}$ *or* $v = \texttt{false}$*.*

(iii) If $\cdot \vdash v : \texttt{arrow}(\tau_1, \tau_2)$ *and* v value *then* $v = \texttt{fn}(\tau_1, x.e)$ *for some* $x.e$*.*

Proof: We distinguish cases on v value and then apply inversion to the given typing judgment. We show only the proof of property *(ii)*.

Case: $v = \textbf{num}(n)$**.** Then we would have $\cdot \vdash \texttt{num}(n) : \texttt{bool}$, which is impossible by inspection of the typing rules.

Case: $v = \textbf{true}$**.** Then we are done, since, indeed $v = \texttt{true}$ or $v = \texttt{false}$.

Case: $v = \textbf{false}$**.** Symmetric to the previous case.

Case: $v = \textbf{fn}(\tau, x.e)$**.** As in the first case, this is impossible by inspection of the typing rules. ■

The preceding value inversion lemmas is also called the *canonical forms theorem* [Ch. 10.2]. Now we can prove the progress theorem.

Theorem 2 (Progress)
If $\cdot \vdash e : \tau$ *then*

(i) either $e \mapsto e'$ *for some* e'*,*

(ii) *or* e value.

Proof: By rule induction on the given typing derivation. Again, we show only the cases for booleans and functions.

Case

$$\frac{x{:}\tau \in \cdot}{\cdot \vdash x : \tau}\ \textit{VarTyp}$$

This case is impossible since the context is empty.

Case

$$\frac{}{\cdot \vdash \texttt{true} : \texttt{bool}}\ \textit{TrueTyp}$$

Then `true` value.

Case

$$\frac{}{\cdot \vdash \texttt{false} : \texttt{bool}}\ \textit{FalseTyp}$$

Then `false` value.

Case

$$\frac{\cdot \vdash e_1 : \texttt{bool} \quad \cdot \vdash e_2 : \tau \quad \cdot \vdash e_3 : \tau}{\cdot \vdash \texttt{if}(e_1, e_2, e_3) : \tau}\ \textit{IfTyp}$$

In this case it is clear that $\texttt{if}(e_1, e_2, e_3)$ cannot be a value, so we have to show that $\texttt{if}(e_1, e_2, e_3) \mapsto e'$ for some e'.

Either $e_1 \mapsto e_1'$ for some e_1' or e_1 value	By i.h.
$e_1 \mapsto e_1'$	First subcase
$\texttt{if}(e_1, e_2, e_3) \mapsto \texttt{if}(e_1', e_2, e_3)$	By rule
e_1 value	Second subcase
$e_1 = \texttt{true}$ or $e_1 = \texttt{false}$	By value inversion
$e_1 = \texttt{true}$	First subsubcase
$\texttt{if}(\texttt{true}, e_2, e_3) \mapsto e_2$	By rule
$e_1 = \texttt{false}$	Second subsubcase
$\texttt{if}(\texttt{false}, e_2, e_3) \mapsto e_3$	By rule

Case

$$\frac{\cdot, x{:}\tau_1 \vdash e_2 : \tau_2}{\cdot \vdash \texttt{fn}(\tau_1, x.e_2) : \texttt{arrow}(\tau_1, \tau_2)}\ \textit{FnTyp}$$

Then $\texttt{fn}(\tau_1, x.e_2)$ value.

Case

$$\frac{\cdot \vdash e_1 : \texttt{arrow}(\tau_2, \tau) \quad \cdot \vdash e_2 : \tau_2}{\cdot \vdash \texttt{apply}(e_1, e_2) : \tau}\ \textit{AppTyp}$$

Either $e_1 \mapsto e_1'$ for some e_1' or e_1 value	By i.h.
$e_1 \mapsto e_1'$	First subcase
$\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1', e_2)$	By rule
e_1 value	Second subcase
Either $e_2 \mapsto e_2'$ for some e_2' or e_2 value	By i.h.
$e_2 \mapsto e_2'$	First subsubcase
$\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1, e_2')$	By rule (since e_1 value)
e_2 value	Second subsubcase
$e_1 = \texttt{fn}(\tau_2, x.e_1')$	By value inversion
$\texttt{apply}(e_1, e_2) \mapsto \{e_2/x\}e_1'$	By rule (since e_2 value)

Case

$$\frac{\cdot, x{:}\tau \vdash e' : \tau}{\cdot \vdash \texttt{rec}(\tau, x.e') : \tau}\ \textit{RecTyp}$$

$\texttt{rec}(\tau, x.e') \mapsto \{\texttt{rec}(\tau, x.e')/x\}e'$	By rule

■

Determinism. We will leave the proof of determinism to the reader—it is not difficult given all the examples and techniques we have seen so far.

Call-by-Value vs. Call-by-Name. The MinML language as described so far is a *call-by-value* language because the argument of a function call is evaluated before passed to the function. This is captured the following rules.

$$\frac{e_1 \mapsto e_1'}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1', e_2)}\ cbv.1$$

$$\frac{v_1\ \textsf{value} \quad e_2 \mapsto e_2'}{\texttt{apply}(v_1, e_2) \mapsto \texttt{apply}(v_1, e_2')}\ cbv.2$$

$$\frac{v_2\ \textsf{value}}{\texttt{apply}(\texttt{fn}(\tau_2, x.e_1), v_2) \mapsto \{v_2/x\}e_1}\ cbv.f$$

We can create a call-by-name variant by *not* permitting the evaluation of the argument (rule $cbv.2$ disappears), but just passing it into the function (replace $cbv.r$ by $cbn.r$). The first rule just carries over.

$$\frac{e_1 \mapsto e_1'}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1', e_2)}\ cbn.1$$

$$\frac{}{\texttt{apply}(\texttt{fn}(\tau_1, x.e_1), e_2) \mapsto \{e_2/x\}e}\ cbn.f$$

Evaluation Order. Our specification of MinML requires the we first evaluate e_1 and then e_2 in application $\texttt{apply}(e_1, e_2)$. We can also reduce from right to left by switching the two search rules. The last one remains the same.

$$\frac{e_2 \mapsto e_2'}{\texttt{apply}(e_1, e_2') \mapsto \texttt{apply}(e_1, e_2')}\ cbvr.1$$

$$\frac{e_1 \mapsto e_1' \quad v_2\ \textsf{value}}{\texttt{apply}(e_1, v_2) \mapsto \texttt{apply}(e_1', v_2)}\ cbvr.2$$

$$\frac{v_2\ \textsf{value}}{\texttt{apply}(\texttt{fn}(\tau_2, x.e_1), v_2) \mapsto \{v_2/x\}e_1}\ cbvr.f$$

The O'Caml dialect of ML indeed evaluates from right-to-left, while Standard ML evaluates from left-to-right. There does not seem to be an intrinsic reason to prefer one over the other, except perhaps that evaluating a term in the order it is written appears slightly more natural.

Unspecified Evaluation Order. [1] The specification of Scheme, when translated into our setting is more difficult to model accurately. There are two

[1]This optional material was not discussed in lecture.

conditions:

(1) In any application $\texttt{apply}(e_1, e_2)$, either or argument may be evaluated first.

(2) There can be no interleaving of the evaluation of the two arguments. In other words, the constituent we pick to evaluate first must be completely evaluated before picking the other.

As discussed before, such an underspecification has obvious disadvantages with respect to portability, since the code exhibits spurious non-determinism. While modelling part (1) is quite straightforward by simply including the left and right search rules, part (2) does not fit into the form of the rules that we have specified so far. It seems that one would need either an auxiliary judgment or some auxiliary abstract syntax constructors. We show here the latter. We introduce three forms of application: uncommitted apply, left-to-right $\texttt{apply}_1$ and right-to-left $\texttt{apply}_2$. The first two rules commit to the choice between the two constructs.

$$\frac{}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}_1(e_1, e_2)}\ \textit{cbvs.dl}$$

$$\frac{}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}_2(e_1, e_2)}\ \textit{cbvs.dr}$$

The second and third set of rules step according to the left-to-right order for $\texttt{apply}_1$ and according to the right-to-left order for $\texttt{apply}_2$.

$$\frac{e_1 \mapsto e_1'}{\texttt{apply}_1(e_1, e_2) \mapsto \texttt{apply}_1(e_1', e_2)}\ \textit{cbvs.l1}$$

$$\frac{v_1\ \mathsf{value} \quad e_2 \mapsto e_2'}{\texttt{apply}_1(v_1, e_2) \mapsto \texttt{apply}_1(v_1, e_2')}\ \textit{cbvs.l2}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{apply}_2(e_1, e_2') \mapsto \texttt{apply}_2(e_1, e_2')}\ \textit{cbvs.r1}$$

$$\frac{e_1 \mapsto e_1' \quad v_2\ \mathsf{value}}{\texttt{apply}_2(e_1, v_2) \mapsto \texttt{apply}_2(e_1', v_2)}\ \textit{cbvs.r2}$$

The final set of rules carries out the identical reductions for the two committed forms of application.

$$\frac{v_2 \ \mathsf{value}}{\mathtt{apply}_1(\mathtt{fn}(\tau_2, x.e_1), v_2) \mapsto \{v_2/x\}e_1}\ \mathit{cbvs.lr}$$

$$\frac{v_2 \ \mathsf{value}}{\mathtt{apply}_2(\mathtt{fn}(\tau_2, x.e_1), v_2) \mapsto \{v_2/x\}e_1}\ \mathit{cbvs.rr}$$

For this to work properly we must enforce that the constructors $\mathtt{apply}_1$ and $\mathtt{apply}_2$ are only used internally in the semantics, but are not accessible through the concrete syntax of the language. This is because the language does not actually provide the programmer with the explicit choice: his or her program should be correct no matter which order of evaluation is applied.