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. We also consider how we have to change the operational semantics and the statement of the progress theorem when run-time errors are permitted in the language, such as division by zero. 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: $v = \texttt{num}(n)$**.** Then we would have $\cdot \vdash \texttt{num}(n) : \texttt{bool}$, which is impossible by inspection of the typing rules.

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

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

Case: $v = \texttt{fn}(\tau, x.e)$**.** 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

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

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

Case

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

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

Case

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

$\mathtt{rec}(\tau, x.e') \mapsto \{\mathtt{rec}(\tau, x.e')/x\}e'$	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.

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

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

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

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.

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

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

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

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.

Accounting for Errors [1] It is not always possible to avoid run-time errors, due to limitations in type systems. To illustrate how they can be accounted

[1] This section adapted from notes by Daniel Spoonhower, Fall 2003.

for we will add another primitive operator over integers, division. Unlike addition, subtraction, and multiplication, the division of integers is a *partial* function. That is, it does not yield a result for all possible inputs. In particular, consider the expression `div(num(2), num(0))`. We would like to include division in our type-safe language, but so far we have no way of accounting for what "happens" when we evaluate a division by zero.

(One possibility is to add an additional value of type `int` that is the result of such an expression. This value is sometimes called "NaN" or "not-a-number" when it appears in specifications of floating-point arithmetic. If we were to do so, however, we would have other problems to consider; for example, what is the result of `num(1) = NaN`?)

We will add a new expression to our language, shown below, to capture the state when an expression is "undefined". (This expression is also sometimes known `wrong` or as the "stuck state.")

$$e ::= \ldots \mid \texttt{error}$$

(Is `error` a value? Why or why not? It may become more clear when we introduce a typing rule for `error` below.)

With `error` in hand, we can give an evaluation rule that applies to the expression above.

$$\frac{}{\texttt{div(num(}k\texttt{), num(0))} \mapsto \texttt{error}}\ DivZero$$

We haven't quite finished with evaluation yet, however: consider the following expression:

$$\texttt{if(div(num(2), num(0)),...)} \mapsto \texttt{if(error,...)} \mapsto ?$$

Even though we've made progress with division, we still are stuck at the `if`. We will need to add new rules to *propagate* errors through all of our existing constructs. Analogously to our search evaluation rules, we add:

$$\frac{}{\texttt{apply(error,}e_2\texttt{)} \mapsto \texttt{error}} \qquad \frac{v_1 \text{ value}}{\texttt{apply(}v_1\texttt{,error)} \mapsto \texttt{error}}$$

$$\frac{}{\texttt{if(error,}e_1\texttt{,}e_2\texttt{)} \mapsto \texttt{error}} \qquad \frac{}{\texttt{let(error,}x.e\texttt{)} \mapsto \texttt{error}}$$

$$\frac{v_1 \text{ value} \quad \ldots \quad v_{j-1} \text{ value}}{o\texttt{(}v_1\texttt{,}\ldots\texttt{,}v_{j-1}\texttt{,error,}e_{j+1}\texttt{,}\ldots\texttt{,}e_n\texttt{)} \mapsto \texttt{error}}$$

Here, o stands for a primitive operations with n arguments.

Typing For Errors Before we can go ahead and extend our safety proof, we must give a type to our new expression. Since no actual computation is performed once we have encountered an `error`, we can assign *any* type to an expression that has failed (i.e., there is no way to distinguish one error from another).

$$\frac{}{\Gamma \vdash \texttt{error} : \tau}\ ErrorTyp$$

Preservation

If $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau$. We have previously shown this proof by induction over the derivation of $e \mapsto e'$, so we have six new cases to consider. We show only two.

Rule *DivZero* $e' = \texttt{error}$

There are no assumptions to this rule, so we have no subderivations to consider. However, we only need to show that $\cdot \vdash e' : \tau$. Since $e' = \texttt{error}$, this is easy enough.

$\cdot \vdash \texttt{error} : \tau$ By rule

Rule *IfError* $e' = \texttt{error}$

Again we have no assumptions and so, again, no subderivations. In fact, this case looks just like the last case!

$\cdot \vdash \texttt{error} : \tau$ By rule

All of our new cases for preservation look exactly like this since each evaluates (in one step) to the `error` expression. With these new cases, our extended proof of preservation is complete.

Progress

Here we must extend the theorem: if $\cdot \vdash e : \tau$ then either

i. e value or

ii. $e \mapsto e'$ for some e' or

iii. e is `error`

This proof was given by rule induction over the derivation of $\cdot \vdash e : \tau$, and we have one new typing rule to consider, so we have one additional case.

Rule *ErrorTyp* $e = \texttt{error}$

e is `error` By assumption

Easy enough! Have we finished? No, because we have extended the induction hypothesis, we have an additional subcase to consider each time we applied it.

Consider the case for *IfTyp*:

$$\frac{\begin{array}{ccc}\vdots & \vdots & \vdots \\ \cdot \vdash e : \texttt{bool} & \cdot \vdash e_1 : \tau & \cdot \vdash e_2 : \tau\end{array}}{\cdot \vdash \texttt{if}(e, e_1, e_2) : \tau}$$

Previously, we applied the induction hypothesis to the first subderivation to conclude:

$$\text{Either } e \text{ value or } e \mapsto e'$$

Now must must consider each of:

$$\text{Either } e \text{ value or } e \mapsto e' \text{ or } e \text{ is } \texttt{error}$$

The first two subcases are identical to those in our old proof, but we must finish the third.

e is `error` By case (iii) of i.h.
$\texttt{if}(\texttt{error}, e_1, e_2) \mapsto \texttt{error}$ By rule

We have shown that there is a step to be made and so progress is maintained.

In each of the applications of the induction hypothesis, we will have a new subcase, and (if we've set things up correctly) we should have a new rule to apply. If we find a subcase and no rule to apply, it probably means that we've forgotten a rule; conversely, if a new rule doesn't apply anywhere, it was probably unnecessary.

(Is it clear now why we don't want `error` to be a value? Think about value inversion with respect to `error`.)