Lectures Notes on
Type Safety

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 6
September 11, 2003

In this lecture we discuss and prove several language properties of MinML that connect the type system to the operational semantics. In particular, we will show

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 $e = e''$.

Usually, preservation and progress together are called *type safety*. 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.

Preservation. This is the most fundamental property, and it would be difficult to see how one could accept a type system in which this would fail. Failure of this property amounts to a missing connection between the type system and the operational semantics, and it is unclear how we would even interpret the statement that $e : \tau$. If preservation holds, we can usually interpret a typing judgment as a partial correctness assertion about the expression:

> *If expression e has type τ and e evaluates to a value v, then v also has type τ.*

Progress. This property tells us that evaluation of an expression does not get stuck in any unexpected way: either we have a value (and are done), or there is a way to proceed. If a language is to satisfy progress it should not have any expressions whose operational meaning is undefined. For example, if we added division to MinML we could simply not specify any transition rule that would apply for the expression $\texttt{divide}(\texttt{num}(k), \texttt{num}(0))$. Not specifying the results of such a computation, however, is a bad idea because presumably an implementation will do *something*, but we can no longer know what. This means the behavior is implementation-dependendent and code will be unportable. To describe the behavior of such partial expressions we usually resort to introducing error states or exceptions into the language.

There are other situations where progress may be violated. For example, we may define a non-deterministic language that includes failure (non-deterministic choice between zero alternatives) as an explicit outcome.

Determinism. There are many languages, specifically those with concurrency or explicit non-deterministic choice, for which determinism fails, and for which it makes no sense to require it. On the other hand, we should always be aware whether our language is indeed deterministic or not. There are also situations where the language semantics explicitly violates determinism in order to give the language implementor the freedom to choose convenient strategies. For example, the *Revised^5 Definition of Scheme*[1] states that the arguments to a function may be evaluated in any order. In fact, the order of evaluation for every single procedure call may be chosen differently!

While every implementation conforming to such a specification is presumably deterministic (and the language satisfies both preservation and progress), code which accidentally or consciously relies on the order of evaluation of a particular compiler will be non-portable between Scheme implementations. Moreover, the language provides absolutely no help in discovering such inadvisable implementation-dependence. While one is easily willing to accept this for concurrent languages, where different interleavings of computation steps are an unavoidable fact of life, it is un-

[1] `http://www.swiss.ai.mit.edu/~jaffer/r5rs_toc.html`

fortunate for a language which could quite easily be deterministic, and is intended to be used deterministically.

Preservation. For the proof of preservation we need two properties about the substitution operation as it occurs in the cases of `let`-expressions and function application. We state them here in a slightly more general form than we need, but a slightly less general form than what is possible.

Theorem 1 (Properties of Typing)

(i) (Weakening) If $\Gamma_1, \Gamma_2 \vdash e' : \tau'$ *the* $\Gamma_1, x{:}\tau, \Gamma_2 \vdash e' : \tau'$.

(ii) (Substitution)
If $\Gamma_1, x{:}\tau, \Gamma_2 \vdash e' : \tau'$ *and* $\cdot \vdash e : \tau$ *then* $\Gamma_1, \Gamma_2 \vdash \{e/x\}e' : \tau'$.

Proof: Property *(i)* follows directly by rule induction on the given derivation: we can insert the additional hypothesis in every hypothetical judgment occuring in the derivation without invalidating any rule applications.

Property *(ii)* allso follows by a rule induction on the given derivation of $\Gamma_1, x{:}\tau, \Gamma_2 \vdash e' : \tau'$. Since typing and substitution are both compositional over the structure of the term, the only interesting cases is where e' is the variable x.

Case: (Rule *VarTyp*) with $e' = x$. Then $\tau' = \tau$ and $\{v/x\}e' = \{v/x\}x = v$. So we have to show $\Gamma_1, \Gamma_2 \vdash v : \tau$. But our assumption is $\cdot \vdash v : \tau$ so we can conclude this by weakening (Property *(i)*). ■

Both the weakening and substitution properties arise directly from the nature of reasoning from assumption. They are special cases of very general properties of hypothetical judgments.

Weakening is a valid principle, because when we reason from assumption nothing compels us to actually use any given assumption. Therefore we can always add more assumptions without invalidating our conclusion.

Substitution is a valid principle, because we can always replace the use of an assumption by its derivation.

Theorem 2 (Preservation)
If $\cdot \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot \vdash e' : \tau$.

Proof: By rule induction on the derivation of $e \mapsto e'$. In each case we apply inversion to the given typing derivation and then apply either the induction hypothesis or directly construct a typing derivation for e'.

Critical in this proof is the syntax-directed nature of the typing rules: for each construct in the language there is exactly one typing rule. Preservation is significantly harder for languages that do not have this property, and there are many advanced type systems that are *not* a priori syntax-directed.

We only show the cases for booleans and functions, leaving integers and `let`-expressions to the reader.

Case

$$\frac{e_1 \mapsto e_1'}{\mathtt{if}(e_1, e_2, e_3) \mapsto \mathtt{if}(e_1', e_2, e_3)}$$

This case is typical for search rules, which compute on some subexpression.

$e_1 \mapsto e_1'$	Subderivation
$\cdot \vdash \mathtt{if}(e_1, e_2, e_3) : \tau$	Assumption
$\cdot \vdash e_1 : \mathtt{bool}$ and $\cdot \vdash e_2 : \tau$ and $\cdot \vdash e_3 : \tau$	By inversion
$\cdot \vdash e_1' : \mathtt{bool}$	By i.h.
$\cdot \vdash \mathtt{if}(e_1', e_2, e_3) : \tau$	By rule

Case

$$\frac{}{\mathtt{if}(\mathtt{true}, e_2, e_3) \mapsto e_2}$$

$\cdot \vdash \mathtt{if}(\mathtt{true}, e_2, e_3) : \tau$	Assumption
$\cdot \vdash \mathtt{true} : \mathtt{bool}$ and $\cdot \vdash e_2 : \tau$ and $\cdot \vdash e_3 : \tau$	By inversion
$\cdot \vdash e_2 : \tau$	In line above

Case

$$\frac{}{\mathtt{if}(\mathtt{false}, e_2, e_3) \mapsto e_3}$$

Symmetric to the previous case.

Case

$$\frac{e_1 \mapsto e_1'}{\mathtt{apply}(e_1, e_2) \mapsto \mathtt{apply}(e_1', e_2)}$$

$e_1 \mapsto e_1'$	Subderivation
$\cdot \vdash \mathtt{apply}(e_1, e_2) : \tau$	Assumption
$\cdot \vdash e_1 : \mathtt{arrow}(\tau', \tau)$ and $\cdot \vdash e_2 : \tau'$ for some τ'	By inversion
$\cdot \vdash e_1' : \mathtt{arrow}(\tau', \tau)$	By i.h.
$\cdot \vdash \mathtt{apply}(e_1', e_2) : \tau$	By rule

Case

$$\frac{v_1\ \mathsf{value} \quad e_2 \mapsto e_2'}{\mathtt{apply}(v_1, e_2) \mapsto \mathtt{apply}(v_1, e_2')}$$

Analogous to the previous case.

Case

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

$\cdot \vdash \mathtt{apply}(\mathtt{fn}(\tau_2, x.e_1), v_2) : \tau$	Assumption
$\cdot \vdash \mathtt{fn}(\tau_2, x.e_1) : \mathtt{arrow}(\tau', \tau)$ and $\cdot \vdash v_2 : \tau'$ for some τ'	By inversion
$\cdot, x{:}\tau' \vdash e_1 : \tau$ and $\tau_2 = \tau$	By inversion
$\cdot \vdash \{v_2/x\}e_1 : \tau$	By substitution property

Case

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

$\cdot \vdash \mathtt{rec}(\tau', x.e') : \tau$	Assumption
$\cdot, x{:}\tau \vdash e' : \tau$ and $\tau' = \tau$	By inversion
$\cdot \vdash \{\mathtt{rec}(\tau, x.e')/x\}e'$	By substitution property

■

In summary, in MinML preservation comes down to two observations: (1) for the search rules, we just use the induction hypothesis, and (2) for reduction rules, the interesting cases rely on the substitution property. The latter states that substuting a (closed) expression of type τ for a variable of type τ in an expression of type τ' preserves the type of that expression as τ'.

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,

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

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 3 (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. 9.2]. Now we can prove the progress theorem.

Theorem 4 (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

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

This case is impossible since the context is empty.

Case

$$\frac{}{\cdot \vdash \mathtt{true} : \mathtt{bool}}\ \mathit{TrueTyp}$$

Then $\mathtt{true}$ value.

Case

$$\frac{}{\cdot \vdash \mathtt{false} : \mathtt{bool}}\ \mathit{FalseTyp}$$

Then $\mathtt{false}$ value.

Case

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

In this case it is clear that $\mathtt{if}(e_1, e_2, e_3)$ cannot be a value, so we have to show that $\mathtt{if}(e_1, e_2, e_3) \mapsto e'$ 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
$\mathtt{if}(e_1, e_2, e_3) \mapsto \mathtt{if}(e_1', e_2, e_3)$	By rule
e_1 value	Second subcase
$e_1 = \mathtt{true}$ or $e_1 = \mathtt{false}$	By value inversion
$e_1 = \mathtt{true}$	First subsubcase
$\mathtt{if}(\mathtt{true}, e_2, e_3) \mapsto e_2$	By rule
$e_1 = \mathtt{false}$	Second subsubcase
$\mathtt{if}(\mathtt{false}, e_2, e_3) \mapsto e_3$	By rule

Case

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

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

Case

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

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

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

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

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

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

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

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.

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

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

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

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. The specification of Scheme, when translated into our setting is more difficult to model accurately. There are two 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.

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

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

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 $\texttt{apply}_2$.

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

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

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

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

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

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

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

For this to work properly we must enforce that the constructors $\texttt{apply}_1$ and $\texttt{apply}_2$ 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.