Lectures Notes on Type Safety

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 6
September 16, 2004

Before we discuss type safety, we introduce recursion into the language. Without recursion, the set of functions that can be defined on natural numbers is of course very limited. Rather than tying recursion to functions, the way it is done in [Ch. 9], we introduce it here as a separate concept. In concrete syntax, we write `rec x:t => e` for a recursive expression. The same is represented in abstract syntax as $\mathtt{rec}(\tau, x.e)$, which makes explicit that x is a bound variable with scope e. The intuitive meaning of $\mathtt{rec}(\tau, x.e)$ is that it should be "equal" to its unfolding, that is, the result of substituting the whole expressions for x in e. That is, in some sense we would like to equate $\mathtt{rec}(\tau, x.e)$ with $\{\mathtt{rec}(\tau, x.e)/x\}e$. In the operational semantics, this is manifest in the rule

$$\frac{\{\mathtt{rec}(\tau, x.e)/x\}e \Downarrow v}{\mathtt{rec}(\tau, x.e) \Downarrow v}$$

As an example, consider the exponential function

$$\begin{array}{lcl} 2^0 & = & 1 \\ 2^n & = & 2 \times 2^{n-1} \quad \text{for } n > 0. \end{array}$$

In order to express this with the recursion construction, we write

$$\begin{array}{l} \mathtt{rec}(\mathtt{arrow}(\mathtt{int}, \mathtt{int}), p.\mathtt{fn}(\mathtt{int}, n. \\ \quad \mathtt{if}(\mathtt{equals}(n, \mathtt{num}(0)), \\ \qquad \mathtt{num}(1), \\ \qquad \mathtt{times}(\mathtt{num}(2), \mathtt{apply}(p, \mathtt{minus}(x, \mathtt{num}(1)))))). \end{array}$$

or, in concrete syntax:

```
rec p:int -> int => fn n:int =>
  if n = 0
     then 1
     else 2 * p (x - 1)
```

You should convince yourself on the example above that unfolding yields the correct behavior. As for the typing rule: the whole expression must have the same type as x, so that the substitution $\{\texttt{rec}(\tau, x.e)/x\}$ makes sense. The same type τ must also be the type of e, because the value of e is returned as the value of the recursive expression.

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

As in function expressions, the type τ is recorded in the syntax so that type-checking can be implemented in a simple manner.

In MinML, most useful recursions have the form

$$\texttt{rec}(\texttt{arrow}(\tau_1, \tau_2), f.\texttt{fn}(\tau_1, x.e)),$$

because most other recursive expressions will not terminate (try, for example, $\texttt{rec}(\texttt{int}, x.x)$). We therefore introduce a new form of concrete syntax, `fun f(x:`τ_1`):`τ_2 `=>` e, as "syntactic sugar". During parsing it is expanded into $\texttt{rec}(\texttt{arrow}(\tau_1, \tau_2), f.\texttt{fn}(\tau_1, x.e))$. This means that a `fun`-expression does not have first-class status. For example, we do not give any typing or evaluation rules since we type-check and evaluate the result of the syntactic expansion, not the original form.

At this point we can prove type preservation and value soundness for our language in the following form:

1. (Preservation) If $\cdot \vdash e : \tau$ and $e \Downarrow v$ then $\cdot \vdash v : \tau$.
2. (Value Soundness) If $e \Downarrow v$ then v value.

There are some other properties of interest, such as $v \Downarrow v$ for any value v. Other natural properties do not hold. For example:

1. (Failure of Termination) There is an e such that $\cdot \vdash e : \tau$, but there is no v such that $e \Downarrow v$.
2. (Failure of Reverse Preservation) There are values v and expressions e such that $\cdot \vdash v : \tau$ and $e \Downarrow v$ but not $\cdot \vdash e : \tau$.

It is instructive to find such counterexamples and consider the reasons why neither termination nor reverse preservation can be expected for a practical programming language.

We will not prove either the positive or negative properties of our language in the form given above. The reason is that preservation, while certainly expected to hold, is somewhat weak as a language property: It only talks about expressions e that already are known to have a value. For example, if we omit the rule for function application (which is at the very heart of our language), then preservation would still hold! Moreover, any non-terminating computation is not addressed in this theorem at all.

This means we should look for stronger properties to characterize not only the relationship of an expression to its final value, but the process of computation itself. This requires a different form of operational semantics in which the steps of a computation are made explicit. We write $e \mapsto e'$ for the judgment that e steps to e', yet to be defined. It is related to the evaluation judgment in that

$$e \mapsto e_1 \mapsto \cdots \mapsto e_n \mapsto v \text{ for some } e_1, \ldots, e_n \text{ iff } e \Downarrow v$$

Before we give the rules, we state the properties we expect the language to satisfy in the end. This is a useful strategy which can prevent us from going astray and discovering potential problems with our judgments and rules early. The main properties are:

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

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

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

Small-step semantics. An operational semantics that specifies computation step by step is usually called a small-step semantics. We also call it *structural operational semantics*. We retain the value judgment defined in the last lecture and add the new judgment $e \mapsto e'$, as indicated above. When presenting the operational semantics, we proceed type by type.

Integers This is straightforward. We evaluate the arguments to a primitive operation from left to right, and apply the operation once all arguments have been evaluated.

$$\frac{e_1 \mapsto e_1'}{\texttt{equals}(e_1, e_2) \mapsto \texttt{equals}(e_1', e_2)} \qquad \frac{v_1 \text{ value} \quad e_2 \mapsto e_2'}{\texttt{equals}(v_1, e_2) \mapsto \texttt{equals}(v_1, e_2')}$$

$$\frac{(k_1 = k_2)}{\texttt{equals}(\texttt{num}(k_1), \texttt{num}(k_2)) \mapsto \texttt{true}}$$

$$\frac{(k_1 \neq k_2)}{\texttt{equals}(\texttt{num}(k_1), \texttt{num}(k_2)) \mapsto \texttt{false}}$$

We refer to the first two as *search rules*, since they traverse the expression to "search" for the subterm where the actual computation step takes place. The latter two are *reduction rules*.

Booleans For if-then-else we have only one search rule for the condition, since we never evaluate in the branches before we know which one to take.

$$\frac{e \mapsto e'}{\texttt{if}(e, e_1, e_2) \mapsto \texttt{if}(e', e_1, e_2)}$$

$$\frac{}{\texttt{if}(\texttt{true}, e_1, e_2) \mapsto e_1} \qquad \frac{}{\texttt{if}(\texttt{false}, e_1, e_2) \mapsto e_2}$$

Definitions We proceed as in the expression language with the substitution semantics. There are no new values, and only one search rule.

$$\frac{e_1 \mapsto e_1'}{\texttt{let}(e_1, x.e_2) \mapsto \texttt{let}(e_1', x.e_2)}$$

$$\frac{v_1 \text{ value}}{\texttt{let}(v_1, x.e_2) \mapsto \{v_1/x\}e_2}$$

Functions Applications are evaluated from left-to-right, until both the function and its argument are values. This means the language is a *call-by-value* language with a *left-to-right* evaluation order.

$$\frac{e_1 \mapsto e_1'}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1', e_2)} \qquad \frac{v_1 \text{ value} \quad e_2 \mapsto e_2'}{\texttt{apply}(v_1, e_2) \mapsto \texttt{apply}(v_1, e_2')}$$

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

Recursion A recursive expression is evaluated simply by unfolding it.

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

A recursive expression is never a value, but in a typical use of the form

$$\texttt{rec}(\texttt{arrow}(\tau_1, \tau_2), f.\texttt{fn}(\tau_1, x.e))$$

we can make only one step before reaching a value, because unfolding the `rec` exposes an `fn`-abstraction which is always a value. In [Ch. 9], the recursive expression $\texttt{fun}(\tau_1, \tau_2, f.x.e)$ which corresponds to the above is directly a value. This is appropriate in the case of MinML, but would lead to difficulties in a more general setting later in the course where we study recursively defined lists, trees, and other data structures.

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.

The proof below requires the use of the proof principle of *inversion*. Say in the course of a proof you have established that a certain judgment J has a derivation. If you can see, purely syntactically, that there is only one possible inference rule that could have been used to conclude J, then we know the premises of the rule must also be hold. It is called inversion because in a strange way we go from a derivation of the conclusion to a derivation of the premises. The proof of preservation below uses inversion essentially in each case, applying it to the given typing derivation for e. Since the typing judgment is syntax-directed, and there is exactly one rule for each kind of expression, it is usually straightforward to apply inversion.

A word of caution: many mistakes arise in proofs because inversion is used incorrectly. Remember: you can only apply it if you already know, either from and assumption or the induction hypothesis, that a certain jnudgment must have a 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

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

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

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

Case

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

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

Case

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

Symmetric to the previous case.

Case

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

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

Case

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

Analogous to the previous case.

Case

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

$\cdot \vdash \texttt{apply}(\texttt{fn}(\tau_2, x.e_1), v_2) : \tau$	Assumption
$\cdot \vdash \texttt{fn}(\tau_2, x.e_1) : \texttt{arrow}(\tau', \tau)$ 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

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

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

In the next lecture we show the proof of the progress theorem and also extend our language with more type constructors that will be necessary to represent more complex data types.