Lecture Notes on
A Functional Language

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 5
September 14, 2004

We now introduce MinML, a small fragment of ML that serves to illustrates key points in its design and key techniques for verifying its properties. The treatment here is somewhat cursory; see [Ch. 9] for additional material. Roughly speaking, MinML arises from the arithmetic expression language by adding booleans, functions, and recursion. Functions are (almost) first-class in the sense that they can occur anywhere in an expression, rather than just at the top-level as in other languages such as C. This has profound consequences for the required implementation techniques (to which we will return later), but it does not affect typing in an essential way.

First, we give the grammar for the higher-order abstract syntax. For the concrete syntax, please refer to Assignment 2.

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{arrow}(\tau_1, \tau_2) \\
\text{Integers} & n & ::= & \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \\
\text{Primops} & o & ::= & \texttt{plus} \mid \texttt{minus} \mid \texttt{times} \mid \texttt{negate} \\
 & & \mid & \texttt{equals} \mid \texttt{lessthan} \\
\\
\text{Expressions} & e & ::= & \texttt{num}(n) \mid o(e_1, \ldots, e_n) \\
 & & \mid & \texttt{true} \mid \texttt{false} \mid \texttt{if}(e, e_1, e_2) \\
 & & \mid & \texttt{let}(e_1, x.e_2) \\
 & & \mid & \texttt{fn}(\tau, x.e) \mid \texttt{apply}(e_1, e_2) \\
 & & \mid & \texttt{rec}(\tau, x.e) \\
 & & \mid & x
\end{array}
$$

Our typing judgment that sorts out the well-formed expressions has the form $\Gamma \vdash e : \tau$, where a context Γ has the form $\cdot, x_1{:}\tau_1, \ldots, x_n{:}\tau_n$. It is a hypothetical judgment as explained in the previous lecture. Our assumption that all variables x_i declared in a context must be distinct is still in force, which means that the rule

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

is unambiguous since there can be at most one declaration for x in Γ.

We have already discussed arithmetic expressions; booleans constitute a similar basic type. Unlike languages such as C, integers and booleans are strictly separate types, avoiding some common confusions and errors. Below are the typing rules related to booleans.

$$\frac{\Gamma \vdash e_1 : \mathtt{int} \quad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash \mathtt{equals}(e_1, e_2) : \mathtt{bool}}\ \textit{EqualsTyp}$$

$$\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}}\ \textit{TrueTyp} \qquad\qquad \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}}\ \textit{FalseTyp}$$

$$\frac{\Gamma \vdash e : \mathtt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{if}(e, e_1, e_2) : \tau}\ \textit{IfTyp}$$

Perhaps the only noteworthy point here is that the two branches of a conditional must have the same type. This is because we cannot know at type-checking time which branch will be taken at run-time. We are therefore conservative, asserting only that the result of the conditional will definitely have type τ if each branch has type τ. Later in this class, we will see a type system that can more accurately analyze conditionals so that, for example, `if true then 1 else false` could be given a type (which is impossible here).

A more important extension from our first language of arithmetic expressions is the addition of functions. In mathematics we are used to describe functions in the form $f(x) = e$, for example $f(x) = x^2 + 1$. In a functional language we want a notation for the function f itself. The abstract (mathematical) notation for this concept is λ-abstraction, written $f = \lambda x.e$. The above example would be written as $f = \lambda x.x^2 + 1$.

In the concrete syntax of MinML we express $\lambda x{:}\tau.e$ as `fn x:t => e`; in our abstract syntax it is written as $\mathtt{fn}(\tau, x.e)$. This is an illustration of the unfortunate situation that we generally have to deal with at least three ways of expressing the same concepts. One is the mathematical notation, one is the concrete syntax, and one is the abstract syntax. In research papers, one mostly uses mathematical notation or pseudo-concrete syntax that really stands for abstract syntax but is easier to read. Inevitably, we will also start sliding between levels of discourse which is acceptable as long as we always know what we *really* mean.

Returning to functions, the typing rules are rather straightforward.

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

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

Keep in mind that in the rule *FnTyp*, the variable x must not already be declared in Γ. We can always rename x in $\mathtt{fn}(\tau, x.e)$ to satisfy this condition, because we treat abstract syntax as α-equivalence classes, that is, modulo variable renaming.

Functions defined with the language given so far are rather limited. For example, there is no way to define the exponential function from multiplication and addition, because there is no way to express recursion implicit in the definition

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

We address this problem in the next lecture when we introduce the concept of recursion.

Below is a summary of the typing rules for the language. We show only the case of one operator—the others are analogous.

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \mathit{VarTyp} \qquad \frac{}{\Gamma \vdash \mathtt{num}(n) : \mathtt{int}}\ \mathit{NumTyp}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{int} \quad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash \mathtt{equals}(e_1, e_2) : \mathtt{bool}}\ \mathit{EqualsTyp}$$

$$\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}}\ \mathit{TrueTyp} \qquad \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}}\ \mathit{FalseTyp}$$

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let}(e_1, x.e_2) : \tau_2}\ \mathit{LetTyp}$$

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

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

We specify the operational semantics as a *natural semantics* also called a *big-step semantics*. As the semantics for our small language of expressions, it relates an expression to its final value (if it has one), but it does not directly specify each step of evaluation. We use substitution instead of environments for simplicity, so the judgment has the form $e \Downarrow v$ where we assume that $\cdot \vdash e : \tau$. We also define the judgment e value which expresses that e is a value (written v).

Integers This is quite simple and as for arithmetic expressions. First, only numbers are values.

$$\frac{}{\texttt{num}(k)\ \mathsf{value}}$$

Then the rules for evaluations; we only show the rules for the primitive equality operator.

$$\frac{}{\texttt{num}(k) \Downarrow \texttt{num}(k)}$$

$$\frac{e_1 \Downarrow \texttt{num}(k_1) \quad e_2 \Downarrow \texttt{num}(k_2) \quad (k_1 = k_2)}{\texttt{equals}(e_1, e_2) \Downarrow \texttt{true}}$$

$$\frac{e_1 \Downarrow \texttt{num}(k_1) \quad e_2 \Downarrow \texttt{num}(k_2) \quad (k_1 \neq k_2)}{\texttt{equals}(e_1, e_2) \Downarrow \texttt{false}}$$

Booleans First, `true` and `false` are values.

$$\frac{}{\texttt{true}\ \mathsf{value}} \qquad \frac{}{\texttt{false}\ \mathsf{value}}$$

Then, the decision on which branch of a conditional to evaluate is based on the return value of the condition.

$$\frac{}{\texttt{true} \Downarrow \texttt{true}} \qquad \frac{}{\texttt{false} \Downarrow \texttt{false}}$$

$$\frac{e \Downarrow \texttt{true} \quad e_1 \Downarrow v_1}{\texttt{if}(e, e_1, e_2) \Downarrow v_1} \qquad \frac{e \Downarrow \texttt{false} \quad e_2 \Downarrow v_2}{\texttt{if}(e, e_1, e_2) \Downarrow v_2}$$

Definitions This remains unchanged from the arithmetic expression language.

$$\frac{e_1 \Downarrow v_1 \quad \{v_1/x\}e_2 \Downarrow v_2}{\texttt{let}(e_1, x.e_2) \Downarrow v_2}$$

Functions It is often claimed that functions are "first-class", but this is not quite true, since we cannot observe the structure of functions in the same way we can observe booleans or integers. Therefore, there is no need to evaluate the body of a function, and in fact we could not since it is not closed and we would get stuck when encountering the function parameter. So, any function by itself is a value.

$$\frac{}{\texttt{fn}(\tau, x.e)\ \mathsf{value}}$$

The second question is if we have to evaluate the argument in a function call before performing the call. Both answers are sensible. In languages like C, Java, or ML, function arguments are evaluated. This also corresponds to mathematical practice. For example, a function from integers to integers takes integers as arguments, not expressions. Of course, we may perform reasoning to deduce equations involving functions, but this is quite distinct from computation. In other languages like Haskell, function arguments are not evaluated. We will discuss this possibility and its applications in more detail later in this course. Given that we evaluate arguments, we call our language *call-by-value* and define it by the following rules.

$$\frac{}{\texttt{fn}(\tau, x.e) \Downarrow \texttt{fn}(\tau, x.e)}$$

$$\frac{e_1 \Downarrow \texttt{fn}(\tau_2, x.e_1') \quad e_2 \Downarrow v_2 \quad \{v_2/x\}e_1' \Downarrow v}{\texttt{apply}(e_1, e_2) \Downarrow v}$$

Which theorems regarding the operational semantics make sense in this setting? First, we can state that evaluation, if it terminates, should always result in a value. Second, we can state that evaluation preserves the type of the expression all the way to its value. Finally, we want to claim that the language is deterministic, that is, the value of an expression (if it exists) is uniquely determined.

1. (Evaluation) If $\cdot \vdash e : \tau$ and $e \Downarrow v$ then $v\ \mathsf{value}$.
2. (Preservation) If $\cdot \vdash e : \tau$ and $e \Downarrow v$, then $\cdot \vdash v : \tau$
3. (Determinism) If $\cdot \vdash e : \tau$ and $e \Downarrow v'$ and $e \Downarrow v''$ then $v' = v''$.

We will return to the problem of proving these and similar theorems in the next lecture. Note that this does not exhaust the possibilities of possible theorems, and there are many other properties which may be of interest for specific purposes in the implementation or use of a language.

We conclude the lecture with some discussion on how these inference rules may be viewed as specifications of algorithms. Interestingly, the evaluation judgment can be viewed as an algorithm for evaluating an expression, and the typing judgment can be viewed as an algorithm for type-checking. However, not every judgment can be interpreted in this way, so we must take some care to ensure this kind of reading is meaningful. The kind of reasoning we apply here is also the kind of reasoning required to turn the judgments and rules into functional implementations (say, using ML or Haskell as an implementation language). This sort of analysis is routine for programming language researchers, but it is rarely made explicit.

We begin with the evaluation judgment. We would like to read the rules for evaluation as an algorithm for computing the value of an expression. So we commit to saying that in the judgment $e \Downarrow v$, e is the input (given) and v is the output (to be computed). Now we analyze each rule to see if we can see how to compute v given e.

Integers For integers, the analysis is entirely straightforward.

$$\frac{}{\mathtt{num}(k) \Downarrow \mathtt{num}(k)}$$

Given the input $\texttt{num}(k)$ we can indeed compute the (identical) output $\texttt{num}(k)$.

$$\frac{e_1 \Downarrow \texttt{num}(k_1) \quad e_2 \Downarrow \texttt{num}(k_2) \quad (k_1 = k_2)}{\texttt{equals}(e_1, e_2) \Downarrow \texttt{true}}$$

$$\frac{e_1 \Downarrow \texttt{num}(k_1) \quad e_2 \Downarrow \texttt{num}(k_2) \quad (k_1 \neq k_2)}{\texttt{equals}(e_1, e_2) \Downarrow \texttt{false}}$$

Given the input $\texttt{equals}(e_1, e_2)$ we know both e_1 and e_2. Since e_1 is known, by induction hypothesis[1] we can compute k_1. From the second premise we can obtain k_2. Then we can compare these values and return either `true` or `false`, depending on which rule applies.

We skip Booleans and definitions, and go right to the most complicated case of functions.

Functions Function expression evaluate to themselves, so if we know the input we can return the output.

$$\frac{}{\texttt{fn}(\tau, x.e) \Downarrow \texttt{fn}(\tau, x.e)}$$

$$\frac{e_1 \Downarrow \texttt{fn}(\tau_2, x.e_1') \quad e_2 \Downarrow v_2 \quad \{v_2/x\}e_1' \Downarrow v}{\texttt{apply}(e_1, e_2) \Downarrow v}$$

For function application, the reasoning is more complex.

We are given $\texttt{apply}(e_1, e_2)$.
Hence we know e_1 and e_2.
By i.h. we know $\texttt{fn}(\tau_2, x.e_1')$.
By i.h. we know v_2.
We therefore can calculate $\{v_2/x\}e_1'$
By i.h. we can compute v
Therefore we can return v

For the typing judgment, we can perform a similar analysis. But first we have to decide what are the inputs, and what are the outputs of the judgment $\Gamma \vdash e : \tau$. We might try[2] to use both Γ, e, and τ as inputs and decide

[1] This reasoning could be formally set up as an induction, showing that if e is given then v can be computed (assuming it exists at all). Even though we do not formalize this, we still refer to the "induction hypothesis" when analyzing the premises.

[2] suggested in lecture by a student

if the judgments holds or not (that is, either succeed or fail). Unfortunately, this does not work for function application $\mathtt{apply}(e_1, e_2)$: we cannot determine the type of the argument e_2.

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

We assume that Γ, $\mathtt{apply}(e_1, e_2)$ and τ are known. But we cannot apply the induction hypothesis to the first premise, because τ_2 is unknown. Similarly, we cannot apply the induction hypothesis to the second premise, since τ_2 is unknown. We are therefore stuck, which means that we cannot easily interpret the typing rules for checking a given expression against a given type.

Fortuntely, we can assume Γ and e as inputs and generate τ as output, or fail (if the expression is not well-typed). In that case we analyze the rule as follows.

Γ, $\mathtt{apply}(e_1, e_2)$ are given.
Therefore, e_1 and e_2 are known.
By i.h. a τ_1 such that $\Gamma \vdash e_1 : \tau_1$ can be computed (or we fail).
By i.h. τ_2 can be computed from the second premise (or we fail).
Now we check if $\tau_1 = \mathtt{arrow}(\tau_2, \tau)$ for some τ.
If no, we fail.
If yes, we return τ.

Finally, we consider functional abstraction.

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

Γ, $\mathtt{fn}(\tau_1, x.e)$ are given.
Hence τ_1, x, and e are known.
By i.h. τ_2 can be computed (or we may fail).
If we succeed, we can construct $\mathtt{arrow}(\tau_2, \tau_2)$.

This reasoning required that the type τ_1 be present in the expression, otherwise we could not apply the induction hypothesis. This is precisely the reason why τ_1 in in fact required in the syntax. ML does not require this type, because it performs a much more complicated analysis of expressions called *type inference*. Briefly, does not compute exact types but creates placeholders and generates a potentially large set of equational constraints between types and placeholders which must be satisfied for the expression to be well-typed. It then solves these constraints by an algorithm that resembles Gaussian elimination for solving linear arithmetic equalities. We will come back to this process in a later lecture.