Lecture Notes on Aggregate Data Structures

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 8
September 23, 2004

In this lecture we discuss various language extensions which make MinML a more realistic language without changing its basic character. In the second part of the lecture we also consider an environment-based semantics in which we avoid explicit application of substitution to give the semantics a more realistic character and discuss some common mistakes in language definition.

Products. Introducing products just means adding pairs and a unit element to the language [Ch. 19.1]. We could also directly add n-ary products, but we will instead discuss records later when we talk about object-oriented programming. MinML is a call-by-value language. For consistency with the basic choice, the pair constructor also evaluates its arguments—otherwise we would be dealing with *lazy pairs*.[1] In addition to the `pair` constructor, we can extract the first and second component of a pair.[2]

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{pair}(e_1, e_2) : \mathtt{cross}(\tau_1, \tau_2)}$$

$$\frac{\Gamma \vdash e : \mathtt{cross}(\tau_1, \tau_2)}{\Gamma \vdash \mathtt{fst}(e) : \tau_1} \qquad \frac{\Gamma \vdash e : \mathtt{cross}(\tau_1, \tau_2)}{\Gamma \vdash \mathtt{snd}(e) : \tau_2}$$

We often adopt a more mathematical notation according to the table at the end of these notes. However, it is important to remember that the

[1]See Assignment 3

[2]An alternative treatment is given in [Ch. 19.1], where the destructor provides access to both components of a pair simultaneously.

mathematical shorthand is just that: it is just a different way to shorten higher-order abstract syntax or make it easier to read.

A pair is a value if both components are values. If not, we can use the search rules to reduce, using a left-to-right order. Finally, the reduction rules extract the corresponding component of a pair.

$$\frac{e_1 \ \mathsf{value} \quad e_2 \ \mathsf{value}}{\mathtt{pair}(e_1, e_2) \ \mathsf{value}}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{pair}(e_1, e_2) \mapsto \mathtt{pair}(e_1', e_2)} \quad \frac{v_1 \ \mathsf{value} \quad e_2 \mapsto e_2'}{\mathtt{pair}(v_1, e_2) \mapsto \mathtt{pair}(v_1, e_2')}$$

$$\frac{e \mapsto e'}{\mathtt{fst}(e) \mapsto \mathtt{fst}(e')} \quad \frac{e \mapsto e'}{\mathtt{snd}(e) \mapsto \mathtt{snd}(e')}$$

$$\frac{v_1 \ \mathsf{value} \quad v_2 \ \mathsf{value}}{\mathtt{fst}(\mathtt{pair}(v_1, v_2)) \mapsto v_1} \quad \frac{v_1 \ \mathsf{value} \quad v_2 \ \mathsf{value}}{\mathtt{snd}(\mathtt{pair}(v_1, v_2)) \mapsto v_2}$$

Since it is at the core of the progress property, we make the value inversion property explicit.

> If $\cdot \vdash v : \mathtt{cross}(\tau_1, \tau_2)$ and v value then $v = \mathtt{pair}(v_1, v_2)$ for some v_1 value and v_2 value.

Unit Type. For the unit type we only have a constructor but no destructor, since there are no components to extract.[3]

$$\frac{}{\Gamma \vdash \mathtt{unitel} : \mathtt{unit}}$$

The unit types does not yield any new search or reduction rules, only a new value. At first it may not seem very useful, but we will see an application when we add references to the language.

$$\frac{}{\mathtt{unitel} \ \mathsf{value}}$$

The value inversion property is also simple.

> If $\cdot \vdash v : \mathtt{unit}$ then $v = \mathtt{unitel}$.

[3] A so-called `check` construct is possible but not necessary; see [Ch. 19.1].

Sums. Unions, as one might know them from the C programming language, are inherently not type safe. They can be abused in order to access the underlying representations of data structures and intentionally violate any kind of abstraction that might be provided by the language. Consider, for example, the following snippet from C.

```
union {
   float f;
   int   i;
} unsafe;

unsafe.f = 5.67e-5;
printf("%d", unsafe.i);
```

Here we set the member of the union as a floating point number and then print the underlying bit pattern as if it represented an integer. Of course, much more egregious examples can be imagined here.

In a type-safe language we replace unions by disjoint sums. In the implementation, the members of a disjoint sum type are tagged with their origin so we can safely distinguish the cases. In order for every expression to have a unique type, we also need to index the corresponding injection operator with their target type. We avoid this complication here, postponing the issue of how to perform type-checking to a future lecture.

$$\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathtt{inl}(e_1) : \mathtt{sum}(\tau_1, \tau_2)} \quad \frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{inr}(e_2) : \mathtt{sum}(\tau_1, \tau_2)}$$

$$\frac{\Gamma \vdash e : \mathtt{sum}(\tau_1, \tau_2) \quad \Gamma, x_1{:}\tau_1 \vdash e_1 : \sigma \quad \Gamma, x_2{:}\tau_2 \vdash e_2 : \sigma}{\Gamma \vdash \mathtt{case}(e, x_1.e_1, x_2.e_2) : \sigma}$$

Note that we require both branches of a `case`-expression to have the same type σ, just as for a conditional, because we cannot be sure at type-checking time which branch will be taken at run time.

$$\frac{e_1 \ \mathsf{value}}{\mathtt{inl}(e_1) \ \mathsf{value}} \qquad \frac{e_2 \ \mathsf{value}}{\mathtt{inr}(e_2) \ \mathsf{value}}$$

$$\frac{e \mapsto e'}{\mathtt{case}(e, x_1.e_1, x_2.e_2) \mapsto \mathtt{case}(e', x_1.e_1, x_2.e_2)}$$

$$\frac{v_1 \ \mathsf{value}}{\mathtt{case}(\mathtt{inl}(v_1), x_1.e_1, x_2.e_2) \mapsto \{v_1/x_1\}e_1}$$

$$\frac{v_2 \ \mathsf{value}}{\mathtt{case}(\mathtt{inr}(v_2), x_1.e_1, x_2.e_2) \mapsto \{v_2/x_2\}e_2}$$

We also state the value inversion property.

> If $\cdot \vdash v : \mathtt{sum}(\tau_1, \tau_2)$ then either $v = \mathtt{inl}(v_1)$ with v_1 value or $v = \mathtt{inr}(v_2)$ with v_2 value.

Void type. The empty type `void` can be thought of as a zero-ary sum. It has no values, and can only be given to expressions that do not terminate. For example,

$$\frac{\Gamma, x{:}\mathtt{void} \vdash x : \mathtt{void}}{\Gamma \vdash \mathtt{rec}(\mathtt{void}, x.x) : \mathtt{void}}$$

The value inversion property here just expresses that there are no values of void type.

> If $\cdot \vdash v : \mathtt{void}$ then we have a contradiction.

In this lecture we did not explicitly revisit the cases in the proof of the preservation and progress theorem, but the cases follow exactly the previously established patterns.

Environment-Based Semantics. So far, most of our semantic specifications rely on *substitution* as a primitive operation. From the point of view of implementation, this is impractical, because a program would be copied many times. So we seek an alternative semantics in which substitutions are not carried out explicitly, but an association between variables and their values is maintained. Such a data structure is called an *environment*. Care has to be taken to ensure that the intended meaning of the program (as given by the specification with substitution) is not changed. We have already discussed such a semantics in Lecture 4 for arithmetic expressions.

Because we are in a call-by-value language, environments η bind variables to values.

$$\text{Environments} \quad \eta \quad ::= \quad \cdot \mid \eta, x{=}v$$

The basic intuition regarding typing is that if $\Gamma \vdash e : \tau$, then e should be evaluated in an environment which supplies bindings of appropriate type for all the variables declared in Γ. We therefere formalize this as a judgment, writing $\eta : \Gamma$ if the bindings of variables to values in η match the context Γ. We make the general assumption that a variable x is bound only once in an environment, which corresponds to the assumption that a variable x is declared only once in a context. If necessary, we can rename bound variables in order to maintain this invariant.

$$\frac{}{\cdot : \cdot} \qquad\qquad \frac{\eta : \Gamma \quad \cdot \vdash v : \tau \quad v \ \mathsf{value}}{(\eta, x{=}v) : (\Gamma, x{:}\tau)}$$

Note that the values v bound in an environment are closed, that is, they contain no free variables. This means that expressions are evaluated in an environment, but the resulting values must be closed. This creates a difficulty when we come to the evaluation of function expressions. Relaxing this restriction, however, causes even more serious problems.[4]

We start with integers and let-bindings, the latter of which is an explicit motivation for the introduction of environments. Here we assume some primitive operators $\mathtt{o}$ (such as `plus` and `times`) and their mathematical counterparts f_o. For simplicity, we just write binary operators here.

$$\frac{x{\Downarrow}v \in \eta}{\eta \vdash x \Downarrow v}\ e.var \qquad\qquad \frac{}{\eta \vdash \mathtt{num}(k) \Downarrow \mathtt{num}(k)}\ e.num$$

$$\frac{\eta \vdash e_1 \Downarrow \mathtt{num}(k_1) \quad \eta \vdash e_2 \Downarrow \mathtt{num}(k_2) \quad (f_o(k_1, k_2) = k)}{\eta \vdash \mathtt{o}(e_1, e_2) \Downarrow \mathtt{num}(k)}\ e.o$$

$$\frac{\eta \vdash e_1 \Downarrow v_1 \quad \eta, x{\Downarrow}v_1 \vdash e_2 \Downarrow v_2}{\eta \vdash \mathtt{let}(e_1, x.e_2) \Downarrow v_2}\ e.let \quad (x \text{ not declared in } \eta)$$

Next we come to functions. Before we state these rules let's explicitly state the preservation property we expect to hold at the end. Note that there is no progress property, because it cannot be formulated very easily on the big-step semantics.

[4] This is known in the Lisp community as the *upward funarg problem*.

Preservation. If $\Gamma \vdash e : \tau$ and $\eta : \Gamma$ and $\eta \vdash e \Downarrow v$ then $\cdot \vdash v : \tau$.

Note in particular here the formal expression of the intuition above that the output v must be closed. The following rule

$$\frac{}{\eta \vdash \mathtt{fn}(\tau, x.e) \Downarrow \mathtt{fn}(\tau, x.e)}\ e.fn?$$

would be incorrect because $\mathtt{fn}(\tau, x.e)$ can refer to variables defined in η which would "leak" into the output value, violating the closedness condition of the preservation theorem. Instead we need to create a so-called *closure* which pairs up a function with its environment, representing a new form of value. We write

$$\langle\langle \eta; \mathtt{fn}(\tau, x.e) \rangle\rangle$$

for the closure of $\mathtt{fn}(\tau, x.e)$ over the environment η.

There are no evaluation rules for closures (they are values), and the typing rules have to "guess" a context that matches the environment. Note that we always type values in the empty environment.

$$\frac{}{\langle\langle \eta; \mathtt{fn}(\tau, x.e) \rangle\rangle\ \mathsf{value}} \qquad \frac{\eta : \Gamma \quad \Gamma \vdash \mathtt{fn}(\tau, x.e) : \tau'}{\cdot \vdash \langle\langle \eta; \mathtt{fn}(\tau, x.e) \rangle\rangle : \tau'}$$

Note that function expressions like $\mathtt{fn}(\tau, x.e)$ are no longer values–only function closures are values. We now modify the incorrect rule by building a closure instead and write down the right evaluation rule for function application.

$$\frac{}{\eta \vdash \mathtt{fn}(\tau, x.e) \Downarrow \langle\langle \eta; \mathtt{fn}(\tau, x.e) \rangle\rangle}\ e.fn$$

$$\frac{\eta \vdash e_1 \Downarrow \langle\langle \eta'; \mathtt{fn}(\tau_2, x.e_1') \rangle\rangle \quad \eta \vdash e_2 \Downarrow v_2 \quad \eta', x{=}v_2 \vdash e_1' \Downarrow v}{\eta \vdash \mathtt{apply}(e_1, e_2) \Downarrow v}\ e.app$$

Note that every aspect of this rule is critical: evaluation of e_1 returns a closure instead of a function expression, and the body of the function is evaluated in environment found in the closure extended by the binding for x. By our general convention about variables, x may not already be declared in the environment η' so the new one is well-formed. This can always be achieved by the tacit renaming of the bound variable so it differens from the variables in η'.

One interesting problems that arises in this context is the treatment of recursion. There are a number of ways to avoid explicit substitutions, such as creating recursive environments, or allowing bindings of variables to unevaluated expressions, to be evaluated when they are looked up. The solution taken in Standard ML is to syntactically restrict recursion to functional expressions and declare the resulting functions $\texttt{rec}(\texttt{arrow}(\tau_1, \tau_2), f.\texttt{fn}(\tau_1, x.e))$ to be values. This is the solution taken in the notes [Ch. 9].

Higher-Order Abstract Syntax	Concrete Syntax	Mathematical Syntax
$\texttt{arrow}(\tau_1, \tau_2)$	τ_1 `->` τ_2	$\tau_1 \rightarrow \tau_2$
$\texttt{cross}(\tau_1, \tau_2)$	τ_1`*`τ_2	$\tau_1 \times \tau_2$
`unit`	`unit`	1
$\texttt{sum}(\tau_1, \tau_2)$	τ_1`+`τ_2	$\tau_1 + \tau_2$
`void`	`void`	0
$\texttt{pair}(e_1, e_2)$	`(`e_1`,`e_2`)`	$\langle e_1, e_2 \rangle$
$\texttt{fst}(e)$	`#1` e	$\pi_1\, e$
$\texttt{snd}(e)$	`#2` e	$\pi_2\, e$
`unitel`	`()`	$\langle\,\rangle$
$\texttt{inl}(e_1)$	$\texttt{inl}(e_1)$	$\mathsf{inl}_{\tau_1+\tau_2}(e_1)$
$\texttt{inr}(e_2)$	$\texttt{inr}(e_2)$	$\mathsf{inr}_{\tau_1+\tau_2}(e_2)$
$\texttt{case}(e, x_1.e_1, x_2.e_2)$	`case` e `of` $\texttt{inl}(x_1)$ `=>` e_1 `|` $\texttt{inr}(x_2)$ `=>` e_2 `esac`	$\mathsf{case}(e, x_1.e_1, x_2.e_2)$
$\texttt{abort}(e)$	$\texttt{abort}(e)$	$\mathsf{abort}_\tau(e)$