CS411 Notes 10 – Types III

A. Demers

2 Mar 2001

We continue the definition begun in the previous notes by presenting evaluation rules.

1 Evaluation Rules

1.1 Preliminaries

As for previous language fragments, the judgements of our evaluation rules will have the form

$$\langle(\pi \vdash e : tau), \phi, \sigma\rangle \quad \rightarrow \quad \langle v, \sigma'\rangle$$

The expression to be evaluated is a well-typed program expression – that is, a derivation according to the typing rules of the previous Notes.

The environment ϕ is a finite set of bindings

$$\phi \quad = \quad \{\ldots\ x_i \sim b_i\ \ldots\}$$

where b_i is a "bindable value," to be defined later.

The stores σ and σ' are likewise finite sets

$$\sigma \quad = \quad \{\ldots\ x_i \sim v_i\ \ldots\}$$

where v_i is a "(storable) value," also to be defined later.

Clearly we'll want to enforce some consistency properties among the type assignments, expressions, environments and stores in our judgements. First, every free name of e should be defined in π and ϕ:

$$\mathbf{dom}(\pi) = \mathbf{dom}(\phi) \supseteq FV(e)$$

In addition, the types of names in the type assignment and environment should agree:

$$(((x : \tau) \in \pi) \wedge ((x \sim b) \in \phi)) \quad \Rightarrow \quad (\pi \vdash b : \tau)$$

The store should be defined on all addresses that appear explicitly in e, and no location in the store should become undefined:

$$\mathbf{dom}(\sigma') \ \supseteq \ \mathbf{dom}(\sigma) \ \supseteq \ A(e)$$

where $A(e)$ is the set of all explicit address constants a^τ that appear anywhere in e (an inductive definition of $A(e)$ is left as an exercise). Finally, the store itself should be properly typed:

$$((a^\tau \sim v) \in \sigma) \quad \Rightarrow \quad (\pi \vdash v : \tau)$$

That is, if a^τ is a type τ address, then the value stored there should have type τ.

Of course, all this must be considered rather informal, as it relies on an intuitive notion of what it means for a value to "have a type." In particular, we haven't yet said what values are, and how judgements like

$$\pi \vdash b : \tau \qquad \text{and} \qquad \pi \vdash v : \tau$$

should be interpreted for (bindable and storable) values.

1.2 What is a value?

The question "what is a value," was easy to answer when computing over a finite set of simple types. But now we are working with an infinite set of types, with a lot of interesting structure.

Intuitively, we want a value to be a "finished" expression – one on which we cannot perform any further computation. It should not depend on the environment, because it exists in the store independent of any particular environment; and it should not depend on other values in the store, since we would like the freedom to update individual storage locations independently.

Clearly, if an expression has free names, its evaluation might depend on the environment; So something like "17" should be a value, while something like "x" should not. Similarly, if an expression looks up the value at an address (uses the

$\uparrow$ operator), its evaluation might depend on the store; so something like "a^τ" (which represents an address in the store) might be a value, but something like "$a^\tau \uparrow$ (representing the contents of the store at address a^τ) should not be a value.

These goals are easy to achieve for our basic types – it is clear that a (string, bool or int) constant should be usable as a value, and in fact that's what we've been doing all along.

It is also fairly clear how to make values of a product type – they are just tuples composed (inductively) of values of the component types.

It may be less obvious what the values of a function type should be. Intuitively, a function is a rule for computing a (result value and final store) from an (argument value and initial store). Given our goal that a value should not depend on the environment, we shall insist that a function value be closed (i.e. have no free variables). But that is the only constraint we shall impose. Since any closed **lambda** expression can be interpreted as a rule for computing values, we will allow any closed **lambda** expression to be considered a value.

With these goals in mind, we can define storable values inductively as

$$
\begin{array}{l}
v \in \mathbf{V} \subset \mathbf{Exp} \qquad \text{(storable values)} \\
v \ ::= \ n \mid t \mid s \mid a^\tau \\
\qquad \mid \langle \ \ldots \ x_i \sim v_i \ \ldots \ \rangle \\
\qquad \mid (\textbf{lambda } x : \tau \textbf{ dot } e) \ \text{ where } \ FV(v) = \emptyset
\end{array}
$$

That is, the storable values are constants, tuples built up out of storable values, and closed **lambda** expressions.

For now, we shall let the bindable values be the same as the storable ones:

$$b \in \mathbf{B} \ = \ \mathbf{V} \qquad \text{(bindable values)}$$

Later we shall expand the set of bindable values.

The evaluation rules below maintain the invariant that the environment (resp. store) contain only bindable (resp. storable) values. Such values are closed, independent of the current environment. This property will enable us to achieve static scope behavior.

1.3 Eager Static Rules

Here are eager-evaluation, static scope rules.

Constants

$$\frac{}{\langle n, \phi, \sigma\rangle \rightarrow \langle n, \sigma\rangle} \tag{E10.1}$$

$$\frac{}{\langle t, \phi, \sigma\rangle \rightarrow \langle t, \sigma\rangle} \tag{E10.2}$$

$$\frac{}{\langle s, \phi, \sigma\rangle \rightarrow \langle s, \sigma\rangle} \tag{E10.3}$$

$$\frac{}{\langle a^\tau, \phi, \sigma\rangle \rightarrow \langle a^\tau, \sigma\rangle} \tag{E10.4}$$

Constants are elements of V and reduce to themselves.

Operators These rules mimic our earlier systems.

$$\frac{\langle e_1, \phi, \sigma\rangle \rightarrow \langle v_1, \sigma'\rangle}{\langle \psi e_1, \phi, \sigma\rangle \rightarrow \langle v, \sigma'\rangle} \qquad \text{where } v = \psi v_1 \tag{E10.5}$$

$$\frac{\begin{array}{c}\langle e_1, \phi, \sigma\rangle \rightarrow \langle v_1, \sigma_1\rangle \\ \langle e_2, \phi, \sigma_1\rangle \rightarrow \langle v_2, \sigma 2\rangle\end{array}}{\langle e_1 \theta e_2, \phi, \sigma\rangle \rightarrow \langle v, \sigma_2\rangle} \qquad \text{where } v = v_1 \theta v_2 \tag{E10.6}$$

Control Structures Again, there are no surprises here.

$$\frac{\begin{array}{c}\langle e_1, \phi, \sigma\rangle \rightarrow \langle v_1, \sigma_1\rangle \\ \langle e_2, \phi, \sigma_1\rangle \rightarrow \langle v_2, \sigma_2\rangle\end{array}}{\langle (e_1; e_2), \phi, \sigma\rangle \rightarrow \langle v_2, \sigma_2\rangle} \tag{E10.7}$$

$$\frac{\begin{array}{c}\langle e_1, \phi, \sigma\rangle \rightarrow \langle \textbf{true}, \sigma_1\rangle \\ \langle e_2, \phi, \sigma_1\rangle \rightarrow \langle v_2, \sigma'\rangle\end{array}}{\langle (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3), \phi, \sigma\rangle \rightarrow \langle v_2, \sigma'\rangle} \tag{E10.8}$$

$$\frac{\begin{array}{c}\langle e_1, \phi, \sigma\rangle \rightarrow \langle \textbf{false}, \sigma_1\rangle \\ \langle e_3, \phi, \sigma_1\rangle \rightarrow \langle v_3, \sigma'\rangle\end{array}}{\langle (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3), \phi, \sigma\rangle \rightarrow \langle v_3, \sigma'\rangle} \tag{E10.9}$$

$$\frac{\langle e_1, \phi, \sigma \rightarrow \langle\ \textbf{false}, \sigma'\rangle}{\langle (\textbf{while } e_1 \textbf{ do } e_2), \phi, \sigma\rangle \rightarrow \langle\ \textbf{false}, \sigma'\rangle} \tag{E10.10}$$

$$\frac{\begin{array}{c}\langle e_1, \phi, \sigma \rightarrow \langle\ \textbf{true}, \sigma_1\rangle \\ \langle e_2, \phi, \sigma_1\rangle \rightarrow \langle v, \sigma_2\rangle \\ \langle (\textbf{while } e_1 \textbf{ do } e_2), \phi, \sigma_2\rangle \rightarrow \langle\ \textbf{false}, \sigma'\rangle\end{array}}{\langle (\textbf{while } e_1 \textbf{ do } e_2), \phi, \sigma\rangle \rightarrow \langle\ \textbf{false}, \sigma'\rangle} \tag{E10.11}$$

Assignable Variables These rules are interesting for the treatment of address valued constant expressions and explicit memory allocation.

$$\frac{\langle e_1, \phi, \sigma\rangle \rightarrow \langle v_1, \sigma_1\rangle}{\langle \mathbf{newvar}(e_1), \phi, \sigma\rangle \ \rightarrow \ \langle a^\tau, \sigma'\rangle} \tag{E10.12}$$

with the rather complex condition

$$a^\tau = \mathbf{next}(\tau, \sigma_1), \qquad \sigma' = \sigma_1 \oplus \{a^\tau \sim v_1\}$$

That is, if a^τ is the least uninitialized address of type τ in σ, executing **newvar** will allocate and initialize it.

$$\frac{\begin{array}{c}\langle e_1, \phi, \sigma\rangle \rightarrow \langle a^\tau, \sigma_1\rangle \\ \langle e_2, \phi, \sigma_1\rangle \rightarrow \langle v_2, \sigma_2\rangle\end{array}}{\langle e_1 \leftarrow e_2, \phi, \sigma\rangle \rightarrow \langle v_2, \sigma \oplus \{a^\tau \sim v_2\}\rangle} \tag{E10.13}$$

An assignment could refer to an "unallocated" address. This would update the store in a type-safe (though perhaps not program-methodologically-correct) fashion.

$$\frac{\langle e_1, \phi, \sigma\rangle \ \rightarrow \ \langle a^\tau, \sigma_1\rangle}{\langle e_1 \uparrow, \phi, \sigma\rangle \ \rightarrow \ \langle v, \sigma_1\rangle} \qquad \text{where } a^\tau \sim v \in \sigma_1 \tag{E10.14}$$

This rule does not apply unless the location a^τ is defined in σ. In effect, a program attempting to read an uninitialized variable does not reduce; it gets "stuck."

Let Bindings The treatment of **let** blocks is eager.

$$\frac{\begin{array}{c}\cdots \quad \langle e_i, \phi, \sigma_{i-1}\rangle \rightarrow \langle v_i, \sigma_i\rangle \quad \cdots \\ \langle e_0, (\phi \oplus \{\ldots \ x_i \sim v_i \ \ldots\}), \sigma_n\rangle \rightarrow \langle v, \sigma'\rangle\end{array}}{\langle \mathbf{let} \ \ldots \ x_i \sim e_i \ \ldots \ \mathbf{in} \ e_0), \phi, \sigma\rangle \rightarrow \langle v, \sigma'\rangle} \tag{E10.15}$$

The bound expressions e_i are fully (eagerly) evaluated, and the results entered into the environment for the hypothesis.

$$\frac{}{\langle x, \phi, \sigma\rangle \rightarrow \langle v, \sigma\rangle} \qquad \text{where}(x \sim v) \in \phi \tag{E10.16}$$

Since **let** bindings are done eagerly, looking up the value of a name involves no computation, hence an empty hypothesis set.

Products Cartesian products (aka records) are new in this language fragment, but should not hold any surprises. Their evaluation is eager.

$$\frac{\cdots \qquad \langle e_i, \phi, \sigma_{i-1}\rangle \to \langle v_i, \sigma_i\rangle \qquad \cdots}{\langle(\langle \ldots\ x_i \sim e_i\ \ldots\rangle), \phi, \sigma\rangle \to \langle(\langle \ldots\ x_i \sim v_i\ \ldots\rangle), \sigma_n\rangle} \tag{E10.17}$$

$$\frac{\langle e, \phi, \sigma\rangle \to \langle(\langle \ldots\ x_i \sim v_i\ \ldots\rangle), \sigma'\rangle}{\langle e.x_i, \phi, \sigma\rangle \to \langle v_i, \sigma'\rangle} \tag{E10.18}$$

Functions Evaluating a **lambda** expression – that is, producing an equivalent closed **lambda** expression or function value – is the most subtle part of these rules.

$$\frac{}{\langle(\textbf{lambda}\ x : \tau\ \textbf{dot}\ e), \phi, \sigma\rangle \to \langle(\textbf{lambda}\ x : \tau\ \textbf{dot}\ e), \phi, \sigma\rangle} \quad \text{if } FV(e) \subseteq \{x\} \tag{E10.19}$$

A closed **lambda** expression is already a function value; it evaluates to itself.

$$\frac{\langle(\textbf{lambda}\ x : \tau\ \textbf{dot}\ (\textbf{let}\ y \sim v\ \textbf{in}\ e)), \phi, \sigma\rangle \to \langle v', \sigma'\rangle}{\langle(\textbf{lambda}\ x : \tau\ \textbf{dot}\ e), \phi, \sigma \to \langle v', \sigma'\rangle} \quad \begin{array}{c}\text{where } y \text{ is the least element of } FV(e) - \{x\} \\ \text{and } (y \sim v) \in \phi\end{array} \tag{E10.20}$$

To evaluate a **lambda** expression that is not closed, we generate a subgoal in which the function body is enclosed in a **let** block that binds one of the function's free variables to it value from the current environment. Since the value from the environment is closed, the new function body has one fewer free variables than the original one had. This eventually leads to a subgoal in which the **lambda** expression to be evaluated has no free variables at all, so the previous rule applies.

$$\frac{\begin{array}{c}\langle e_1, \phi, \sigma\rangle \to \langle(\textbf{lambda}\ x : \tau\ \textbf{dot}\ e_3), \sigma_1\rangle \\ \langle e_2, \phi, \sigma_1\rangle \to \langle v_2, \sigma_2\rangle \\ \langle e_3, (\phi \oplus \{x \sim v_2\}), \sigma_2\rangle \to \langle v, \sigma'\rangle\end{array}}{\langle e_1(e_2), \phi, \sigma\rangle \to \langle v, \sigma'\rangle} \tag{E10.21}$$

To evaluate an application, first reduce the function part to a function value – a closed **lambda** expression. Then reduce the argument to its value v_2. Finally, evaluate the function body using the one-element environment $\{x \sim v_2\}$. Since the function value was closed, the function body can have at most the parameter name x free, so its evaluation in this environment should succeed.