CIS 500

Software Foundations

Fall 2003

27 October

Administrivia

- Changes to recitation schedule:
  - The advanced recitations on Wednesday 6:00 – 7:30 and Thursday 6:30 – 8:00 are cancelled. The Wednesday 3:30 – 5:00 advanced recitation continues as before.
  - Two new review recitations are now available at [...]
  - You are welcome to attend any recitation(s) you find convenient. (If we have capacity problems, we will figure something out later.)
    The complete list of recitation times and meeting places is available on the course web page.

References (continued)

Evaluation

What is the value of the expression `ref 0`?

Evaluation

What is the value of the expression `ref 0`?

Crucial observation: evaluating `ref 0` must do something.

Otherwise,

```
r = ref 0
s = ref 0
```

and

```
r = ref 0
s = r
```

would behave the same.

Evaluation

What is the value of the expression `ref 0`?

Crucial observation: evaluating `ref 0` must do something.

Otherwise,

```
r = ref 0
s = ref 0
```

and

```
r = ref 0
s = r
```

would behave the same.

Specifically, evaluating `ref 0` should allocate some storage and yield a reference (or pointer) to that storage.

Evaluation

What is the value of the expression `ref 0`?

Crucial observation: evaluating `ref 0` must do something.

Otherwise,

```
r = ref 0
s = ref 0
```

and

```
r = ref 0
s = r
```

would behave the same.

Specifically, evaluating `ref 0` should allocate some storage and yield a reference (or pointer) to that storage.

So what is a reference?

The Store

A reference names a location in the store (also known as the heap or just the memory).

What is the store?

The Store

A reference names a location in the store (also known as the heap or just the memory).

What is the store?

- Concretely: An array of 8-bit bytes, indexed by 32-bit integers.

The Store

A reference names a location in the store (also known as the heap or just the memory).

What is the store?

- Concretely: An array of 8-bit bytes, indexed by 32-bit integers.
- More abstractly: an array of values

The Store

A reference names a location in the store (also known as the heap or just the memory).

What is the store?

- Concretely: An array of 8-bit bytes, indexed by 32-bit integers.
- More abstractly: an array of values
- Even more abstractly: a partial function from locations to values.

Locations

Syntax of values:

v	::=	values
	unit	unit constant
	λx:T.t	abstraction value
	l	store location

... and since all values are terms...

Syntax of Terms

t	::=	terms
	unit	unit constant
	x	variable
	λx:T.t	abstraction
	t t	application
	ref t	reference creation
	!t	dereference
	t:=t	assignment
	l	store location

Aside

Does this mean we are going to allow programmers to write explicit locations in their programs??

No: This is just a modeling trick. We are enriching the "source language" to include some run-time structures, so that we can continue to formalize evaluation as a relation between source terms.

Aside: If we formalize evaluation in the big-step style, then we can add locations to the set of values (results of evaluation) without adding them to the set of terms.

Evaluation

The result of evaluating a term now depends on the store in which it is evaluated. Moreover, the result of evaluating a term is not just a value — we must also keep track of the changes that get made to the store.

I.e., the evaluation relation should now map a term and a store to a reduced term and a new store.

$$t \mid \mu \longrightarrow t' \mid \mu'$$

We use the metavariable μ to range over stores.

Evaluation

An assignment $t_1 := t_2$ first evaluates t_1 and t_2 until they become values...

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1 := t_2 \mid \mu \longrightarrow t_1' := t_2 \mid \mu'} \qquad \text{(E-Assign1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1 := t_2 \mid \mu \longrightarrow v_1 := t_2' \mid \mu'} \qquad \text{(E-Assign2)}$$

... and then returns $\mathtt{unit}$ and updates the store:

$$l := v_2 \mid \mu \longrightarrow \mathtt{unit} \mid [l \mapsto v_2]\mu \qquad \text{(E-Assign)}$$

A term of the form $\texttt{ref}\ t_1$ first evaluates inside t_1 until it becomes a value...

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{\texttt{ref}\ t_1 \mid \mu \longrightarrow \texttt{ref}\ t_1' \mid \mu'} \qquad \text{(E-Ref)}$$

... and then chooses (allocates) a fresh location l, augments the store with a binding from l to v_1, and returns l:

$$\frac{l \notin \mathrm{dom}(\mu)}{\texttt{ref}\ v_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto v_1)} \qquad \text{(E-RefV)}$$

A term $!t_1$ first evaluates in t_1 until it becomes a value...

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{!t_1 \mid \mu \longrightarrow !t_1' \mid \mu'} \qquad \text{(E-DEREF)}$$

... and then looks up this value (which must be a location, if the original term was well typed) and returns its contents in the current store:

$$\frac{\mu(l) = v}{!l \mid \mu \longrightarrow v \mid \mu} \qquad \text{(E-DEREFLOC)}$$

Evaluation rules for function abstraction and application are augmented with stores, but don't do anything with them directly.

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1\ t_2 \mid \mu \longrightarrow t_1'\ t_2 \mid \mu'} \qquad \text{(E-App1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1\ t_2 \mid \mu \longrightarrow v_1\ t_2' \mid \mu'} \qquad \text{(E-App2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \mid \mu \longrightarrow [x \mapsto v_2]t_{12} \mid \mu \qquad \text{(E-AppAbs)}$$

Aside: garbage collection

Note that we are not modeling garbage collection — the store just grows without bound.

Aside: pointer arithmetic

We can't do any!

Store Typings

Typing Locations

Q: What is the type of a location?

Typing Locations

Q: What is the type of a location?

A: It depends on the store!

E.g., in the store $(l_1 \mapsto \mathtt{unit}, l_2 \mapsto \mathtt{unit})$, the term $!l_2$ has type $\mathtt{Unit}$.

But in the store $(l_1 \mapsto \mathtt{unit}, l_2 \mapsto \lambda x{:}\mathtt{Unit}.x)$, the term $!l_2$ has type $\mathtt{Unit}\to\mathtt{Unit}$.

Typing Locations — first try

Roughly:

$$\frac{\Gamma \vdash \mu(l) : T_1}{\Gamma \vdash l : \texttt{Ref}\ T_1}$$

Typing Locations — first try

Roughly:

$$\frac{\Gamma \vdash \mu(l) : T_1}{\Gamma \vdash l : \mathtt{Ref}\ T_1}$$

More precisely:

$$\frac{\Gamma \mid \mu \vdash \mu(l) : T_1}{\Gamma \mid \mu \vdash l : \mathtt{Ref}\ T_1}$$

I.e., typing is now a four-place relation (between contexts, stores, terms, and types).

Problem

However, this rule is not completely satisfactory. For one thing, it can make typing derivations very large!

E.g., if

$$
\begin{aligned}
(\mu = {} & l_1 \mapsto \lambda x{:}\mathtt{Nat}.\ 999, \\
& l_2 \mapsto \lambda x{:}\mathtt{Nat}.\ !l_1\ (!l_1\ x), \\
& l_3 \mapsto \lambda x{:}\mathtt{Nat}.\ !l_2\ (!l_2\ x), \\
& l_4 \mapsto \lambda x{:}\mathtt{Nat}.\ !l_3\ (!l_3\ x), \\
& l_5 \mapsto \lambda x{:}\mathtt{Nat}.\ !l_4\ (!l_4\ x)),
\end{aligned}
$$

then how big is the typing derivation for $!l_5$?

Problem!

But wait... it gets worse. Suppose

$$(\mu = l_1 \mapsto \lambda x{:}\mathtt{Nat}.\ !l_2\ x,$$
$$l_2 \mapsto \lambda x{:}\mathtt{Nat}.\ !l_1\ x),$$

Now how big is the typing derivation for $!l_2$?

Store Typings

Observation: The typing rules we have chosen for references guarantee that a given location in the store is always used to hold values of the same type.

These intended types can be collected into a store typing — a partial function from locations to types.

E.g., for

$$\begin{aligned}\mu = (&l_1 \mapsto \lambda x{:}\mathtt{Nat}.\ 999,\\ &l_2 \mapsto \lambda x{:}\mathtt{Nat}.\ !l_1\ (!l_1\ x),\\ &l_3 \mapsto \lambda x{:}\mathtt{Nat}.\ !l_2\ (!l_2\ x),\\ &l_4 \mapsto \lambda x{:}\mathtt{Nat}.\ !l_3\ (!l_3\ x),\\ &l_5 \mapsto \lambda x{:}\mathtt{Nat}.\ !l_4\ (!l_4\ x)),\end{aligned}$$

A reasonable store typing would be

$$\begin{aligned}\Sigma = (&l_1 \mapsto \mathtt{Nat}\to\mathtt{Nat},\\ &l_2 \mapsto \mathtt{Nat}\to\mathtt{Nat},\\ &l_3 \mapsto \mathtt{Nat}\to\mathtt{Nat},\\ &l_4 \mapsto \mathtt{Nat}\to\mathtt{Nat},\\ &l_5 \mapsto \mathtt{Nat}\to\mathtt{Nat})\end{aligned}$$

Now, suppose we are given a store typing Σ describing the store μ in which we intend to evaluate some term t. Then we can use Σ to look up the types of locations in t instead of calculating them from the values in μ.

$$\frac{\Sigma(l) = T_1}{\Gamma \mid \Sigma \vdash l : \text{Ref } T_1} \qquad \text{(T-Loc)}$$

I.e., typing is now a four-place relation between between contexts, store typings, terms, and types.

Final typing rules

$$\frac{\Sigma(l) = T_1}{\Gamma \mid \Sigma \vdash l : \mathtt{Ref}\ T_1} \qquad \text{(T-Loc)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : T_1}{\Gamma \mid \Sigma \vdash \mathtt{ref}\ t_1 : \mathtt{Ref}\ T_1} \qquad \text{(T-Ref)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \mathtt{Ref}\ T_{11}}{\Gamma \mid \Sigma \vdash !t_1 : T_{11}} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \mathtt{Ref}\ T_{11} \qquad \Gamma \mid \Sigma \vdash t_2 : T_{11}}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \mathtt{Unit}} \qquad \text{(T-Assign)}$$

Q: Where do these store typings come from?

Q: Where do these store typings come from?

A: When we first typecheck a program, there will be no explicit locations, so we can use an empty store typing.

So, when a new location is created during evaluation,

$$\frac{l \notin \mathrm{dom}(\mu)}{\mathtt{ref}\ v_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto v_1)} \qquad \text{(E-REFV)}$$

we can observe the type of v_1 and extend the "current store typing" appropriately.

Safety

[on board]