Announcements

Homework 7 due today, due November 14.

Homework 6 due today.

No in class review.

Revisions this week will be review for midterm.

It will cover TAPL chapters 8-14 (except 12).

Midterm II is one week from Wednesday (November 16).

References

Another example

BoolArray = Ref (Nat ! Bool);
newarray = new_array ()
  : Unit .ref (
    n : Nat . false);
  : Unit ! BoolArray
lookup = 
  a : BoolArray .
    n : Nat . (! a) n;
  : BoolArray ! Nat ! Bool
update =
  a : BoolArray .
    m : Nat .
      v : Bool .
        let oldf = (! a) m
        in
          a :=
            n : Nat .
              if equal m n then v else oldf n;
          let newf = (! a) n
        in
          a :=
            n : Nat .
              if equal m n then v else newf n;
  : BoolArray ! Red (Her ! Boot) ;

leta = newarray ()
print (lookup a 3);
update a 3 true;
lookup a 3

7 November
Fall 2005
Software Foundations
CIS 500
Evaluation

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

(E-Deref)

\[
\begin{align*}
\frac{\pi \mid \lambda \leftarrow \pi \mid t_1}{\lambda = (\lambda)(\pi)}
\end{align*}
\]

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

(E-Ref)

\[
\begin{align*}
\frac{\pi \mid \lambda \leftarrow \pi \mid t_1}{\pi \mid \lambda \leftarrow \pi \mid t_1}
\end{align*}
\]

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

(E-RefV)

\[
\begin{align*}
\frac{\pi \mid \lambda \leftarrow \pi \mid t_1}{\pi \mid \lambda \leftarrow \pi \mid t_1}
\end{align*}
\]
Store Typings

Typing Locations

Q: What is the type of a location?
A: It depends on the store!
E.g., in the store $(l_1 \triangleright \text{unit}, l_2 \triangleright \text{unit})$, the term $l_2$ has type $\text{unit}$. But in the store $(l_1 \triangleright \text{unit}, l_2 \triangleright x : \text{unit})$, the term $l_2$ has type $\text{unit} \rightarrow \text{unit}$.

Typing Locations

Q: What is the type of a location?
A: It depends on the store!
E.g., in the store $(l_1 \triangleright \text{unit}, l_2 \triangleright \text{unit})$, the term $l_2$ has type $\text{unit}$. But in the store $(l_1 \triangleright \text{unit}, l_2 \triangleright x : \text{unit})$, the term $l_2$ has type $\text{unit} \rightarrow \text{unit}$.

Typing Locations

Q: What is the type of a location?
A: It depends on the store!
E.g., in the store $(l_1 \triangleright \text{unit}, l_2 \triangleright \text{unit})$, the term $l_2$ has type $\text{unit}$. But in the store $(l_1 \triangleright \text{unit}, l_2 \triangleright x : \text{unit})$, the term $l_2$ has type $\text{unit} \rightarrow \text{unit}$.
Problem

Now how big is the typing derivation for I_5?  

\[ I_5 \vdash x : \text{Nat} \]  
\[ I_5 \vdash \text{Nat} \]

But wait... it gets worse. Suppose:  

\[ I_5 \vdash x : \text{Nat} \]  
\[ I_5 \vdash \text{Nat} \]

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

\[ I_5 \vdash x : \text{Nat} \]  
\[ I_5 \vdash \text{Nat} \]
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. An example store typing would be

\[
\begin{align*}
&l_1 : \text{Ref T Nat} \\
&l_2 : \text{Ref T Nat} \\
&l_3 : \text{Ref T Nat} \\
&l_4 : \text{Ref T Nat} \\
&l_5 : \text{Ref T Nat} \\
\end{align*}
\]

A reasonable store typing would be

\[
\begin{align*}
&l_1 \mapsto \text{Nat} \\
&l_2 \mapsto \text{Nat} \\
&l_3 \mapsto \text{Nat} \\
&l_4 \mapsto \text{Nat} \\
&l_5 \mapsto \text{Nat} \\
\end{align*}
\]

From locations to types, these intended types can be collected into a store typing — a partial function. A reason store typing in the store is always used to hold values of the same type.
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. Applying the "current store typing" we can observe the type of \( v \) and extend the current store typing.

\[
\text{if } t \vdash \mathcal{L} \text{ and } t \not\vdash \mathcal{L} \text{ then } \exists_1 \vdash \pi \rightarrow \pi, \pi \vdash \mathcal{L} \rightarrow \mathcal{L}.
\]

So, when a new location is created during evaluation, we can use an empty store typing. When we first typecheck a program, there will be no explicit locations, so: Where do these store typings come from?
A store $\sigma$ is said to be well-typed with respect to a typing context $\Gamma$ and a store $\sigma$ if

$$\Gamma \vdash \sigma : \mathbb{X}$$

for some $\mathbb{X} : \mathbb{T}$ and $\Gamma \vdash \tau : \mathbb{X}$.

For some $\mathbb{X} : \mathbb{T}$ and $\Gamma \vdash \tau : \mathbb{X}$.

Substitution for stores:

If $\Gamma \vdash \sigma : \mathbb{X}$ and $\Gamma \vdash \tau : \mathbb{Y}$ then

$$\Gamma \vdash \sigma \tau : \mathbb{Y}$$

Weakening for stores:

If $\Gamma \vdash \sigma : \mathbb{X}$ and $\Gamma \vdash \tau : \mathbb{Z}$ then

$$\Gamma \vdash \sigma \tau : \mathbb{Z}$$

New lemmas for preservation:

Substitution for stores:

If $\Gamma \vdash \sigma : \mathbb{X}$ and $\Gamma \vdash \tau : \mathbb{Y}$ then

$$\Gamma \vdash \sigma \tau : \mathbb{Y}$$

Weakening for stores:

If $\Gamma \vdash \sigma : \mathbb{X}$ and $\Gamma \vdash \tau : \mathbb{Z}$ then

$$\Gamma \vdash \sigma \tau : \mathbb{Z}$$

A store $\sigma$ is said to be well-typed with respect to a typing context $\Gamma$ and
New lemmas for preservation

Substitution for stores:

If \( j \not\models t \) and \( (l) = T \) and \( j \not\models v \), then

\[ j \not\models [l \not\models v] \]

Weakening for stores:

If \( j \not\models t : T \) and \( 0 \), then

\[ j \not\models t : T \]

Safety

Suppose that \( j \not\models t : T \)

then either

1. \( t \) is a value, or else
2. for any store \( f' \) such that \( j \not\models f' \), there is some \( t \) and store \( f'' \) with

\[ t \not\models f' \] \( J \not\models t \) and \( f'' \not\models \]

Suppose that \( J \not\models t \) \( F \) \( J \not\models t \) then either

1. \( t \) is a value, or else
2. for any store \( f' \) such that \( J \not\models f' \), there is some \( t \) and store \( f'' \) with

\[ t \not\models f' \]

Why isn't \( \mathcal{F} \) required to be empty?

\[ \mathcal{F} \]

Excluding for stores:

If \( J \not\models t : T \) and \( \not\models t \) and \( f' \not\models \]

Suppose that \( J \not\models t \) \( f' \not\models \)

1. \( t \) is a value, or else
2. for any store \( f' \) such that \( J \not\models f' \), there is some \( t \) and store \( f'' \) with

\[ t \not\models f' \]

Why isn't \( \mathcal{F} \) required to be empty?

\[ \mathcal{F} \]

Substitution for stores:

If \( J \not\models t \) \( \not\models t \) and \( f' \not\models \]

Suppose that \( J \not\models t \) \( f' \not\models \)

1. \( t \) is a value, or else
2. for any store \( f' \) such that \( J \not\models f' \), there is some \( t \) and store \( f'' \) with

\[ t \not\models f' \]