

Programming Languages, CPSC 604—Slides 11
References and Store

Jaakko Järvi

March 12, 2009

Outline

- 1 References
- 2 Evaluation
- 3 Typing
- 4 Safety properties
- 5 Subtyping and references

Introduction

- This far the entire execution state of our abstract machine has been represented in the term that is being evaluated
- This is not very realistic, practical languages have all kinds of **computational effects**, and the evaluation depends on the state of a memory
- Next we look at how to model **memory** abstractly, and reason about it
- We look at simply-typed lambda calculus with **unit** and **references**.

Operations on references

- Basic operations on mutable references:
 - Allocation `r = ref 17`
 - Dereferencing `!r`
 - Assignment `r := 99`
- Many languages do not differentiate between references and immutable “variables” (C++, Java)
- Others do (ML, Oz, Scheme)
- We too make the use of references explicit in our syntax

Typing terms involving references

- A reference `cell` containing a value of type T has type `Ref T`.
- Assume $r : \text{Ref } T$, then $!r : T$.
- Assignment to a reference does not return anything in our language (unlike in, say, C). Assume $t : T$, then
`r := t : unit`

Side effects

- Now evaluating terms can have side-effects
- With that, sequencing makes more sense
`r := if (!r) then false else true; !r`
- Due to assignment having type `unit`, we can sequence many assignments:
`r := pred(!r); r := pred(!r); r := pred(!r); !r`

Aliasing and shared state

- What are the effects of evaluating the following terms:

$a = \{\text{ref } 0, \text{ref } 0\}$

$c = (\lambda x.\{x, x\})(\text{ref } 0)$

Aliasing and shared state

- What are the effects of evaluating the following terms:

$a = \{\text{ref } 0, \text{ref } 0\}$

$c = (\lambda x.\{x, x\})(\text{ref } 0)$

- The effect of the last term depends on the evaluation order

Aliasing and shared state

- What are the effects of evaluating the following terms:

$a = \{\text{ref } 0, \text{ref } 0\}$

$c = (\lambda x.\{x, x\})(\text{ref } 0)$

- The effect of the last term depends on the evaluation order
- What about the effect of this?

$(r := 1; r := !s)$

Aliasing and shared state

- What are the effects of evaluating the following terms:

`a = {ref 0, ref 0}`

`c = (λ x.{x, x})(ref 0)}`

- The effect of the last term depends on the evaluation order
- What about the effect of this?

`(r := 1; r := !s)`

- Consider the case where `r = s`

Simple objects

```
cell = ref 0;  
inc = λ x:unit.(cell := succ (!cell); !cell);  
dec = λ x:unit.(cell := pred (!cell); !cell);
```

```
counter_obj = {increment = inc,  
               decrement = dec};
```

```
counter_obj.decrement unit;  
counter_obj.increment unit;
```

- `cell` is shared state (state of an object)

Garbage collection

- Notice that we have no operation for deallocating cells
- We assume garbage collection
- Type safety without GC (or really with manual deallocation) is difficult to achieve
 - Intractable in languages like C, C++

Outline

- 1 References
- 2 Evaluation**
- 3 Typing
- 4 Safety properties
- 5 Subtyping and references

Typing references

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \text{ref } t : \text{Ref } T}$$

$$\frac{\Gamma \vdash t : \text{Ref } T}{\Gamma \vdash !t : T}$$

$$\frac{\Gamma \vdash t_1 : \text{Ref } T_1 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 := t_2 : \text{unit}}$$

Store

- Evaluating the allocation operation has a side effect
 - This side effect must somehow be observable — it is not observable in the term
 - The effect is a change in the memory, or **store**.
- ⇒ Evaluation relation must thus now also describe the the changes in the store
- We can abstractly represent store as an array of **values**, and a set of **locations** in the store.
 - A reference is then a location
 - The store is a **partial function from locations to values**
 - Why partial?
 - The metavariable μ ranges over stores

Evaluation relation

- \rightarrow is now a relation between store-term pairs: $(t, \mu) \rightarrow (t', \mu')$
- Effect to our current evaluation rules:

$$\begin{array}{c}
 \text{E-AppAbs} \\
 (\lambda x : T. t) \ v | \mu \rightarrow [v/x]t | \mu
 \end{array}
 \qquad
 \begin{array}{c}
 \text{E-App-Fun} \\
 \frac{t_1 | \mu \rightarrow t'_1 | \mu'}{t_1 \ t_2 | \mu \rightarrow t'_1 \ t_2 | \mu'}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{E-App-Arg} \\
 \frac{t_2 | \mu \rightarrow t'_2 | \mu'}{v \ t_2 | \mu \rightarrow v \ t_2 | \mu'}
 \end{array}$$

- We syntactically represent the store as:
 $(l_1 \mapsto v_1, \dots, l_n \mapsto v_n)$
- $[l \mapsto v]\mu$ gives a store equal to μ , except that l maps to v

New internal form

- The result of allocating memory from the store is a label
- We thus allow new **internal** syntax

$v ::= \dots$

/

label

- It is often convenient to have a slightly different internal language than the one that the user's input must conform to
 - The intermediate language may be more general (some safety checks and restrictions occur only at the surface language)
 - Less cases to handle: trivial syntactic differences are translated away

Evaluation rules for manipulating references

E-Dereference

$$\frac{t|\mu \rightarrow t'|\mu'}{!t|\mu \rightarrow !t'|\mu'}$$

E-DereferenceLocation

$$\frac{\mu(l) = v}{!l|\mu \rightarrow v|\mu}$$

E-AssignLeft

$$\frac{t_1|\mu \rightarrow t'_1|\mu'}{t_1 := t_2|\mu \rightarrow t'_1 := t_2|\mu'}$$

E-AssignRight

$$\frac{t_2|\mu \rightarrow t'_2|\mu'}{v := t_2|\mu \rightarrow v := t'_2|\mu'}$$

E-Assign

$$l := v|\mu \rightarrow \text{unit}([l \mapsto v]|\mu)$$

E-Reference

$$\frac{t|\mu \rightarrow t'|\mu'}{\text{ref } t|\mu \rightarrow \text{ref } t'|\mu'}$$

E-ReferenceValue

$$\frac{l \notin \text{dom}(\mu)}{\text{ref } v \rightarrow l|(\mu, l \rightarrow v)}$$

Outline

- 1 References
- 2 Evaluation
- 3 Typing**
- 4 Safety properties
- 5 Subtyping and references

Store typing

- In principle, we might infer the typing of reference cells from the type stored in the cell

$$\frac{\Gamma \vdash \mu(l) : T}{\Gamma \vdash l : \text{Ref } T}$$

- This is slow, and may not work (think of cycles):

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

- However, even though contents of a cell can change, its type cannot
- ⇒ We use a distinct **store typing**, a mapping from locations to types.
- Σ ranges over store typings

Typing rules for references again

$$\frac{\text{T-Reference} \quad \Gamma | \Sigma \vdash t : T}{\Gamma | \Sigma \vdash \text{ref } t : \text{Ref } T}$$

$$\frac{\text{T-Dereference} \quad \Gamma | \Sigma \vdash t : \text{Ref } T}{\Gamma | \Sigma \vdash !t : T}$$

$$\frac{\text{T-Assignment} \quad \Gamma | \Sigma \vdash t_1 : \text{Ref } T_1 \quad \Gamma | \Sigma \vdash t_2 : T_1}{\Gamma | \Sigma \vdash t_1 := t_2 : \text{unit}}$$

- Rule that performs a lookup to store typing.

$$\frac{\text{T-Location} \quad \Sigma(l) = T}{\Gamma | \Sigma \vdash l : \text{Ref } T}$$

- T-Variable, T-Abstraction, T-Application and T-Unit at home

Outline

- 1 References
- 2 Evaluation
- 3 Typing
- 4 Safety properties**
- 5 Subtyping and references

Type safety

- Proving progress is not interesting. Store must be worked in into to the proofs, and cases for the new typing rules must be added.
- Preservation is trickier:

Theorem (Preservation (INCORRECT))

If $\Gamma|\Sigma \vdash t : T$ and $t|\mu \rightarrow t'|\mu'$, then $\Gamma|\Sigma \vdash t' : T$

Well-typedness of store

Definition (Well-typed store)

A store μ is **well-typed** with respect to typing context Γ and store typing Σ , if $dom(\mu) = dom(\Sigma)$ and $\Gamma|\Sigma \vdash \mu(l) : \Sigma(l)$ for each $l \in dom(\mu)$. The well-typedness relation is written as $\Gamma|\Sigma \vdash \mu$.

Theorem (Preservation)

If $\Gamma|\Sigma \vdash t : T$, $\Gamma|\Sigma \vdash \mu$, and $t|\mu \rightarrow t'|\mu'$, then for some $\Sigma' \supseteq \Sigma$, $\Gamma|\Sigma' \vdash t' : T$ and $\Gamma|\Sigma' \vdash \mu'$,

Proof.

Homework... □

Theorem (Progress (for completeness sake))

If $\emptyset|\Sigma \vdash t : T$, then either t is a value or for any well-typed store μ (i.e., $\emptyset|\Sigma \vdash \mu$), $t|\mu \rightarrow t'|\mu'$ for some t' and μ' .

Outline

- 1 References
- 2 Evaluation
- 3 Typing
- 4 Safety properties
- 5 Subtyping and references**

Subtyping rule for references

$$\frac{S <: T \quad T <: S}{\text{Ref } S <: \text{Ref } T}$$

- Why do we need?

$$\frac{S <: T}{\text{Ref } S <: \text{Ref } T}$$

- Why do we need?

$$\frac{T <: S}{\text{Ref } S <: \text{Ref } T}$$

- Why not?

$$\frac{S \equiv T}{\text{Ref } S <: \text{Ref } T}$$

Subtyping rule for references

$$\frac{S <: T \quad T <: S}{\text{Ref } S <: \text{Ref } T}$$

- Why do we need?

$$\frac{S <: T}{\text{Ref } S <: \text{Ref } T}$$

- Consider $v : \text{Ref } T$. Now $!v$ can be used in a context that expects T . If really $v : \text{Ref } S$, it is not safe unless $S <: T$.

- Why do we need?

$$\frac{T <: S}{\text{Ref } S <: \text{Ref } T}$$

- Why not?

$$\frac{S \equiv T}{\text{Ref } S <: \text{Ref } T}$$

Subtyping rule for references

$$\frac{S <: T \quad T <: S}{\text{Ref } S <: \text{Ref } T}$$

- Why do we need?

$$\frac{S <: T}{\text{Ref } S <: \text{Ref } T}$$

- Consider $v : \text{Ref } T$. Now $!v$ can be used in a context that expects T . If really $v : \text{Ref } S$, it is not safe unless $S <: T$.

- Why do we need?

$$\frac{T <: S}{\text{Ref } S <: \text{Ref } T}$$

- If $v : \text{Ref } T$ and $t : T$, then $v := t$ is ok. If really $v : \text{Ref } S$, then $!v : S$, which is only safe if also $t : S$, which is the case if $T <: S$.

- Why not?

$$\frac{S \equiv T}{\text{Ref } S <: \text{Ref } T}$$

Subtyping rule for references

$$\frac{S <: T \quad T <: S}{\text{Ref } S <: \text{Ref } T}$$

- Why do we need?

$$\frac{S <: T}{\text{Ref } S <: \text{Ref } T}$$

- Why do we need?

$$\frac{T <: S}{\text{Ref } S <: \text{Ref } T}$$

- Why not?

$$\frac{S \equiv T}{\text{Ref } S <: \text{Ref } T}$$

- Consider $v : \text{Ref } T$. Now $!v$ can be used in a context that expects T . If really $v : \text{Ref } S$, it is not safe unless $S <: T$.

- If $v : \text{Ref } T$ and $t : T$, then $v := t$ is ok. If really $v : \text{Ref } S$, then $!v : S$, which is only safe if also $t : S$, which is the case if $T <: S$.

- We still get some flexibility: records with different permutation of fields are mutually subtypes of each other

Same with arrays

- Now it's clear why the Java array covariance rule is problematic

$$\frac{S <: T}{\text{Array } S <: \text{Array } T}$$

- or in Java syntax:

$$\frac{S <: T}{S[] <: T[]}$$