

**State:  
Fluid Binding and References**

Spring 2009  
(with thanks to Robert Harper)

**Fluid Binding**

The language fragment  $\mathcal{L}\{\text{fluid}\}$  is described by the following grammar:

Category	Item	Abstract	Concrete
Expr	$e$	$::= \text{put}[a](e_1; e_2)$	$\text{put } a \text{ is } e_1 \text{ in } e_2$
		$ \text{get}[a]$	$\text{get } a$

Variable  $a$  ranges over some fixed set of **symbols**.

$\text{put } a \text{ is } e_1 \text{ in } e_2$  is a **fluid let**, binding symbol  $a$  to value of  $e_1$  for duration of evaluation of  $e_2$ .

Note that  $a$  is not bound in  $e_1$  or  $e_2$ ; it is simply a parameter to the  $\text{put } a \text{ is } e_1 \text{ in } e_2$  operation.

**Typing Rules for  $\mathcal{L}\{\text{fluid}\}$**

$$\frac{\frac{\Sigma \vdash a : \tau}{\Sigma \Gamma \vdash \text{get}[a] : \tau}}{\Sigma \vdash a : \tau_1 \quad \Sigma \Gamma \vdash e_1 : \tau_1 \quad \Sigma \Gamma \vdash e_2 : \tau_2} \quad \Sigma \Gamma \vdash \text{put}[a](e_1; e_2) : \tau_2$$

**Fluid Binding**

Dynamic scope has serious problems:

- with renaming of bound variables
- with type safety.

A better approach:

- Static binding of **variables** and
- Dynamic or **fluid** binding of **symbols**.

**Static Semantics for  $\mathcal{L}\{\text{fluid}\}$**

Static semantics for  $\mathcal{L}\{\text{fluid}\}$  are parametric in two sets of parameters and hypothetical in two forms of hypotheses:

- Set of symbols  $\mathcal{A}$  and symbol typing assumptions  $\Sigma$  of the form  $a : \tau$
- Set of variables  $\mathcal{X}$  and variable typing assumptions  $\Gamma$  of the form  $x : \tau$
- As usual, explicit mention of  $\mathcal{A}$  and  $\mathcal{X}$  is omitted

**Dynamic Semantics for  $\mathcal{L}\{\text{fluid}\}$**

**Shallow binding** keeps track of values associated with symbols in a stack-like fashion during execution.

Transition judgements have form  $e \mapsto_{\theta} e'$  where  $\theta$  maps symbols to closed values.

If  $a \in \text{dom}(\theta)$  may write  $\theta = \theta' \otimes \langle a : e \rangle$

If  $a \notin \text{dom}(\theta)$  may write  $\theta = \theta' \otimes \langle a : \bullet \rangle$

Will use  $\langle a : \_ \rangle$  to mean either  $\langle a : \bullet \rangle$  or  $\langle a : e \rangle$  for some  $e$ .

### Dynamic Semantics for $\mathcal{L}\{\text{fluid}\}$

$$\frac{}{\text{get}[a] \mapsto_{\theta \otimes (a:e)} e} \frac{e \text{ val}}{} \frac{e_1 \mapsto_{\theta} e'_1}{\text{put}[a](e_1; e_2) \mapsto_{\theta} \text{put}[a](e'_1; e_2)} \frac{e_1 \text{ val} \quad e_2 \mapsto_{\theta \otimes (a:e_1)} e'_2}{\text{put}[a](e_1; e_2) \mapsto_{\theta \otimes (a:\cdot)} \text{put}[a](e_1; e'_2)} \frac{e_1 \text{ val} \quad e_2 \text{ val}}{\text{put}[a](e_1; e_2) \mapsto_{\theta} e_2}$$

6

### Dynamic Semantics for $\mathcal{L}\{\text{fluid}\}$

No transition of the form  $\text{get}[a] \mapsto_{\theta \otimes (a:\bullet)} e$  for any  $e$ .

Need judgements for entering and propagating stuck states that arise from reference to a symbol that is not bound to a value:

$$\frac{}{\text{get}[a] \text{ unbound}_{\theta \otimes (a:\bullet)}} \frac{e_1 \text{ unbound}_{\theta}}{\text{put}[a](e_1; e_2) \text{ unbound}_{\theta}} \frac{e_1 \text{ val} \quad e_2 \text{ unbound}_{\theta \otimes (a:e_1)}}{\text{put}[a](e_1; e_2) \text{ unbound}_{\theta}}$$

7

### Type Safety for $\mathcal{L}\{\text{fluid}\}$

Auxiliary judgement  $\theta : \Sigma$  forces symbol bindings to agree with hypotheses:

$$\frac{\frac{\frac{\theta : \emptyset}{\Sigma \vdash e : \tau} \quad \theta : \Sigma}{\theta \otimes (a:e) : \Sigma, a : \tau}}{\theta : \Sigma} \frac{}{\theta \otimes (a:\bullet) : \Sigma, a : \tau}$$

8

### Type Safety for $\mathcal{L}\{\text{fluid}\}$

#### Theorem 1 (Preservation)

If  $e \mapsto_{\theta} e'$ , where  $\theta : \Sigma$  and  $\Sigma \vdash e : \tau$ , then  $\Sigma \vdash e' : \tau$ .

Proof is by induction on evaluation.

#### Theorem 2 (Progress)

If  $\Sigma \vdash e : \tau$  and  $\theta : \Sigma$  then either  $e \text{ val}$  or  $e \text{ unbound}_{\theta}$  or there exists  $e'$  such that  $e \mapsto_{\theta} e'$ .

The proof is by induction on typing,

9

### Symbol Generation

The language  $\mathcal{L}\{\text{fluid new}\}$ , which enriches  $\mathcal{L}\{\text{fluid}\}$  with ability to generate new symbols during execution, is described by the following grammar:

Category	Item	Abstract	Concrete
Type	$\tau$	$::= \text{sym}(\sigma; \tau)$	$\langle \sigma \rangle \tau$
Expr	$e$	$::= \text{new}[\sigma](a; e)$   $\text{gen}(e)$	$\nu(a : \sigma.e)$ $\text{gen}(e)$

Type  $\text{sym}(\sigma; \tau)$  represents expressions of type  $\tau$  that require a symbol of type  $\sigma$  for their execution.

10

### Typing Rules for $\mathcal{L}\{\text{fluid new}\}$

$$\frac{\Sigma, a : \sigma \Gamma \vdash e : \tau}{\Sigma \Gamma \vdash \text{new}[\sigma](a; e) : \text{sym}(\sigma; \tau)} \frac{\Sigma \Gamma \vdash e : \text{sym}(\sigma; \tau)}{\Sigma \Gamma \vdash \text{gen}(e) : \tau}$$

11

### Dynamic Semantics for $\mathcal{L}\{\text{fluid new}\}$

$$\frac{\frac{\text{new}[\sigma](a; e) \text{ val}}{e \mapsto_{\theta} e'}}{\text{gen}(e) \mapsto_{\theta} \text{gen}(e')}}{\frac{a \text{ fresh}}{\text{gen}(\text{new}[\sigma](a; e)) \mapsto_{\theta} e}}$$

Condition  $a \text{ fresh}$  can always be met by renaming the symbol  $a$  before applying the rule.

12

### Dynamic Semantics for $\mathcal{L}\{\text{fluid new}\}$

To make the freshness condition more precise, define transitions for an **abstract machine** with states  $e @ \nu$ , where

- $\nu$  is a finite set of symbols.
- $e$  is an expression involving at most the symbols in  $\nu$ .

The set  $\nu$  represents the **active** symbols, so a **fresh** symbol is one not in  $\nu$ .

13

### Dynamic Semantics for $\mathcal{L}\{\text{fluid new}\}$

Transition judgement  $e @ \nu \mapsto_{\theta} e' @ \nu'$  defined for states  $e @ \nu$  such that  $\text{dom}(\theta) \subseteq \nu$ , so mapping  $\theta$  governs only active symbols.

Initial states have the form  $e @ \emptyset$ , so  $e$  must type check with no assumptions made about types of symbols.

Final states have the form  $e @ \nu$ , where  $e$  is a value.

14

### Dynamic Semantics for $\mathcal{L}\{\text{fluid new}\}$

$$\frac{\frac{\frac{a \in \nu}{\text{get}[a] @ \nu \mapsto_{\theta @ \langle a; e \rangle} e @ \nu}}{e_1 @ \nu \mapsto_{\theta} e'_1 @ \nu'}}{\text{put}[a](e_1; e_2) @ \nu \mapsto_{\theta} \text{put}[a](e'_1; e_2) @ \nu'}}{\frac{e_1 \text{ val} \quad e_2 @ \nu \mapsto_{\theta @ \langle a; e_1 \rangle} e'_2 @ \nu'}{\text{put}[a](e_1; e_2) @ \nu \mapsto_{\theta @ \langle a; \cdot \rangle} \text{put}[a](e_1; e'_2) @ \nu'}}}{\frac{e_1 \text{ val} \quad e_2 \text{ val}}{\text{put}[a](e_1; e_2) @ \nu \mapsto_{\theta} e_2 @ \nu}}{\frac{e @ \nu \mapsto_{\theta} e' @ \nu'}{\text{gen}(e) @ \nu \mapsto_{\theta} \text{gen}(e') @ \nu'}} \quad \frac{a \notin \nu}{\text{gen}(\text{new}[\sigma](a; e)) @ \nu \mapsto_{\theta} e @ \nu \cup \{a\}}}$$

15

### Persistent and Ephemeral Data Structures

Data structures in conventional imperative languages are **ephemeral**.

- Insertion into a linked list mutates the list. The old version is lost.
- Pushing onto a stack modifies the stack pointer and writes on the underlying memory. Popping writes the stack pointer.

It is difficult to avoid ephemeral data structures in these languages.

16

### Persistent and Ephemeral Data Structures

Data structures in functional languages are **persistent**.

- Inserting an element into a list yields a new list. The old version is still available.
- Stacks can be implemented so that pushing yields a new stack, leaving the old stack still available.

It is possible to support **both** persistent **and** ephemeral data structures, distinguishing a **value** of a type from a **mutable cell** holding a value of the type that may change over time.

17

## References

Reference cells are the fundamental ephemeral structure. The language fragment  $\mathcal{L}\{\text{ref}\}$  of mutable cells is described by the following grammar:

Category	Item	Abstract	Concrete
Type	$\tau$	$::= \text{ref}(\tau)$	$\tau \text{ ref}$
Expr	$e$	$::= \text{loc}[l]$ $\quad   \text{ref}(e)$ $\quad   \text{get}(e)$ $\quad   \text{set}(e_1; e_2)$	$l$ $\text{ref}(e)$ $!e$ $e_1 \leftarrow e_2$

Mutable cells are handled by **reference**, and represented by **locations**, which are **names** or **abstract addresses**, for the cells.

18

## Static Semantics for References

Static semantics for references are parametric in two sets of parameters and hypothetical in two forms of hypotheses:

- Set of locations  $\mathcal{L}$  and location typing assumptions  $\Lambda$  of the form  $l_i : \tau_i$
- Set of variables  $\mathcal{X}$  and variable typing assumptions  $\Gamma$  of the form  $x_i : \tau_i$
- As usual, explicit mention of  $\mathcal{L}$  and  $\mathcal{X}$  is omitted

19

## Typing Rules for References

$$\frac{}{\Lambda, l : \tau \Gamma \vdash \text{loc}[l] : \text{ref}(\tau)}$$

$$\frac{\Lambda \Gamma \vdash e : \tau}{\Lambda \Gamma \vdash \text{ref}(e) : \text{ref}(\tau)}$$

$$\frac{\Lambda \Gamma \vdash e : \text{ref}(\tau)}{\Lambda \Gamma \vdash \text{get}(e) : \tau}$$

$$\frac{\Lambda \Gamma \vdash e_1 : \text{ref}(\tau_2) \quad \Lambda \Gamma \vdash e_2 : \tau_2}{\Lambda \Gamma \vdash \text{set}(e_1; e_2) : \tau_2}$$

20

## Dynamic Semantics for References

**Abstract machine** with states  $e @ \mu$ , where

- $\mu$  is a memory with domain  $\text{dom}(\mu)$ .
- $e$  is an closed expression which may refer to locations in  $\text{dom}(\mu)$ .

A **memory** is a finite function  $\mu : \text{Loc} \rightarrow \text{Val}$  such that

- $\mu(l)$  has no free variables;
- the locations in  $\mu(l)$  are in  $\text{dom}(\mu)$ .

21

## Dynamic Semantics for References

Memory locations act like **bound variables** of the machine state!

- The “names” of locations don't matter, and can be changed at will.
- In a state  $e @ \mu$  the locations in  $\text{dom}(\mu)$  are **bound** simultaneously in  $M$  and in  $e$ .

We regard as equivalent any two states that differ only in the names of locations.

22

## Dynamic Semantics for References

We write  $\emptyset$  for the empty memory and  $\mu \otimes \langle l : e \rangle$  for the result of adding new location  $l$  with contents  $e$  to memory  $\mu$ .

Initial states have the form  $e @ \emptyset$ , where no locations occur in  $e$ .

Final states have the form  $e @ \mu$ , where  $e$  is a value.

Locations are values:  $\overline{\text{loc}[l]} \text{ val}$

23

## Dynamic Semantics for References

$$\frac{e @ \mu \mapsto e' @ \mu'}{\text{ref}(e) @ \mu \mapsto \text{ref}(e') @ \mu'} \quad \frac{e \text{ val}}{\text{ref}(e) @ \mu \mapsto \text{loc}[l] @ \mu \otimes \langle l : e \rangle}$$

$$\frac{e @ \mu \mapsto e' @ \mu'}{\text{get}(e) @ \mu \mapsto \text{get}(e') @ \mu'} \quad \frac{e \text{ val}}{\text{get}(\text{loc}[l]) @ \mu \otimes \langle l : e \rangle \mapsto e @ \mu \otimes \langle l : e \rangle}$$

$$\frac{e_1 @ \mu \mapsto e'_1 @ \mu'}{\text{set}(e_1; e_2) @ \mu \mapsto \text{set}(e'_1; e_2) @ \mu'}$$

$$\frac{e_1 \text{ val} \quad e_2 @ \mu \mapsto e'_2 @ \mu'}{\text{set}(e_1; e_2) @ \mu \mapsto \text{set}(e_1; e'_2) @ \mu'}$$

$$\frac{e \text{ val}}{\text{set}(\text{loc}[l]; e) @ \mu \otimes \langle l : e' \rangle \mapsto e @ \mu \otimes \langle l : e \rangle}$$

24

## Type Safety for References

Two judgements:

- $e @ \mu \text{ ok}$  = machine state  $e @ \mu$  is well-formed.
- $\mu : \Lambda$  = memory  $\mu$  has location typing  $\Lambda$ .

Well-formedness of machine states:

$$\frac{\Lambda \vdash e : \tau \quad \Lambda \vdash \mu : \Lambda}{e @ \mu \text{ ok}}$$

25

## Memory Typing

The definition of memory typing is given by the following rules:

$$\frac{}{\Lambda \vdash \emptyset : \emptyset} \quad \frac{\Lambda \vdash e : \tau \quad \Lambda \vdash \mu' : \Lambda'}{\Lambda \vdash \mu' \otimes \langle l : e \rangle : \Lambda', l : \tau}$$

**Important:** typing for  $\mu(l)$  uses **all** of  $\Lambda!$

- Allows self-reference:  $\mu(l) = \text{fn } x \Rightarrow (!l)(x)$ .
- Allows cyclic references:  $\mu(l) = \text{fn } x \Rightarrow (!l')(x)$  and  $\mu(l') = \text{fn } x \Rightarrow (!l)(x)$ .

26

## Memory Typing

Suppose that  $\mu(l) = \text{fn } x \Rightarrow (!l)(x)$  and that  $\Lambda(l) = \text{int} \rightarrow \text{int}$ .

To check that  $\mu : \Lambda$ , we must check

$$\Lambda \vdash \text{fn } x \Rightarrow (!l)(x) : \text{int} \rightarrow \text{int}$$

This is easily verified, since we are **assuming** that  $l$  has type  $\text{ref}(\text{int} \rightarrow \text{int})$ .

27

## Backpatching

```
(* loop forever when called *)
fun diverge (x:int):int = diverge x
(* allocate a reference cell *)
val fc : (int->int) ref = ref (diverge)
(* define a function that "recurs" through fc *)
fun f 0 = 1 | f n = n * ((!fc)(n-1))
(* tie the knot *)
val _ = fc := f
(* now call f *)
val n : int = f 5
```

28

## Preservation Theorem

### Theorem 3 (Preservation)

If  $e @ \mu \text{ ok}$  and  $e @ \mu \mapsto e' @ \mu'$ , then  $e' @ \mu' \text{ ok}$ .

Doesn't state relation between pre- and post-typing of memory!

- Memory grows monotonically.
- Type never changes once allocated.

29

### Preservation for References

We prove instead this stronger form:

#### Lemma 4

If  $\Lambda \vdash e : \tau$ ,  $\Lambda \vdash \mu : \Lambda$ , and  $e @ \mu \mapsto e' @ \mu'$ , then there exists  $\Lambda' \supseteq \Lambda$  such that  $\Lambda' \vdash e' : \tau$  and  $\Lambda' \vdash \mu' : \Lambda'$ .

Proof is by induction on evaluation.

30

### Proof of Preservation

Suppose that  $e = \mathbf{ref}(v)$  so that  $\mu' = \mu @ (l : v)$ , where  $l \notin \text{dom}(\mu)$ , and  $e' = l$ .

Suppose that  $\mu : \Lambda$  and  $\Lambda \vdash e : \mathbf{ref}(\sigma)$ .

Choose  $\Lambda' := \Lambda, l : \sigma$ . Note that  $l \notin \text{dom}(\Lambda)$  and that  $\Lambda' \supseteq \Lambda$ .

Finally, note that  $\Lambda' \vdash \mu' : \Lambda'$  and  $\Lambda' \vdash e' : \mathbf{ref}(\sigma)$ .

31

### Progress for References

#### Theorem 5 (Progress)

If  $e @ \mu$  ok then either  $e$  is a value or there exists  $e' @ \mu'$  such that  $e @ \mu \mapsto e' @ \mu'$ .

The proof is by induction on typing

32

### Reference Variance

What is the appropriate variance principle for reference types?

- $\sigma \mathbf{ref} <: \tau \mathbf{ref}$  if ... ?

How can a value of type  $\tau \mathbf{ref}$  be used?

- Fetch its contents, use as a value of type  $\tau$ .
- Replace its contents with a value of type  $\tau$ .

33

### Reference Variance

If  $r$  has type  $\sigma \mathbf{ref}$ , then its contents are of type  $\sigma$ .

- Contents have type  $\tau$  only if  $\sigma <: \tau$ .
- **Covariance!**

Suppose we wish to **store** a value of type  $\tau$  in  $r$ .

- Valid only if  $\tau <: \sigma$ .
- **Contravariance!**

34

### Reference Variance

Suppose that  $r$  has type  $\mathbf{int ref}$ .

If reference types were covariant,

- Then  $r$  would have type  $\mathbf{float ref}$  too.
- So we could store  $\pi$  into  $r$ .
- But the contents of  $r$  must be an integer!

35

### Reference Variance

Suppose that  $r$  has type `float ref`.

If reference types were contravariant,

- Then  $r$  would have type `int ref` too.
- Storing an integer is OK, since every integer is a float.
- But the contents of  $r$  might be  $\pi$ !

36

### Reference Variance

Either way we lose type safety.

Conclusion:  $\frac{\sigma <: \tau \quad \tau <: \sigma}{\sigma \text{ ref} <: \tau \text{ ref}}$  so reference types are **invariant**!

- Type of a reference cell does not change by subtyping!

Similar conclusions apply to

- **Mutable record** types whose fields are assignable (e.g., `struct`'s).
- **Mutable arrays** whose elements may be assigned.

37

### Reference Variance

Java gets this wrong!

- Java states that arrays are **covariant**.
- But arrays are **assignable**!

This is **incorrect**! How is this possible?

- Answer: by imposing a severe run-time penalty.

38