Chapter 13: Reference

Why reference

Typing

Evaluation

Store Typings

Safety

Notes

References

Computational Effects

Also known as *side effects*.

A *function* or *expression* is said to have a **side effect** if, in addition to returning a value, it also *modifies some state* or has an *observable interaction with* calling functions or the outside world.

- modify a *global variable* or *static variable*, modify *one of its arguments*,
- *raise an exception,*
- *write* data to *a display* or file, *read* data, or
- call other side-effecting functions.

In the presence of side effects, a program's behavior may depend on *history*; i.e., the *order of evaluation* matters.

Computational Effects

Side effects are the *most common way* that a program *interacts with the outside world* (people, file systems, other computers on networks).

The degree to which side effects are used depends on the *programming paradigm*.

- *Imperative programming* is known for *its frequent utilization* of side effects.
- In *functional programming*, side effects are rarely used. Functional languages like *Standard ML*, *Scheme* and *Scala* do not restrict side effects, but it is customary for programmers to avoid them. The functional language *Haskell* expresses side effects such as I/O and other stateful computations using *monadic* actions.

Mutability

So far, what we have discussed does not yet include *computational effects* (i.e., *side effects*) .

In particular, whenever we defined function, we *never changed variables* *or* *data*. Rather, we always computed *new data*.

- E.g., the operations to *insert an item* into the data structure *didn't effect the old copy* of the data structure. Instead, we *always built a new data structure* with the item appropriately inserted.
- For the most part, programming in a functional style (i.e., *without side effects*) is a "good thing" because it's *easier to reason locally about the behavior* of the program.

Mutability

In most programming languages, *variables are mutable* — i.e., a variable provides both

- ***a name*** that refers to a previously calculated value, and
- ***the possibility of overwriting this value*** with another (which will be referred to by the same name)

In some languages (e.g., OCaml), these features are separate:

- variables are only for naming — the binding between a variable and its value is immutable
- introduce a new class of mutable values (called *reference cells* or *references*)
  - at any given moment, a reference *holds a value* (and can be dereferenced to obtain this value)
  - *a new value* may be assigned to a reference

Mutability

Writing values into memory locations is the **fundamental mechanism** of imperative languages such as C/C++.

– mutable structures are required to implement many *efficient algorithms*.

– they are also very convenient to represent the *current state of a state machine*.

Basic Examples

```
#let r = ref 5
val r : int ref = {contents  = 5}

# r:= !r +2
# !r
-: int = 7

(r:=succ(!r); !r)
(r:=succ(!r); r:=succ(!r); r:=succ(!r);  r:=succ(!r); !r)

 i.e.,
  ((((r:=succ(!r); r:=succ(!r)); r:=succ(!r));  :=succ(!r)); !r)
```

Basic Examples

```
# let flag = ref true;;
-val flag: bool ref = {contents = true}

# if !flag then 1 else 2;;
-: int = 1
```

Reference

Basic operations

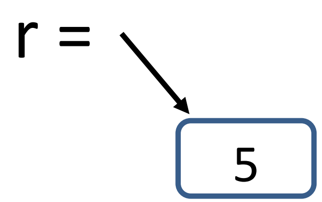
- allocation ref (operator)
- dereferencing !
- assignment :=

Is there any difference between the expressions of ?

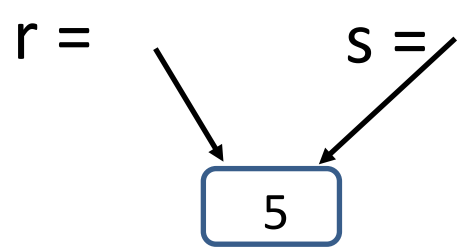
```
5 + 8;
r: =7;
(r:=succ(!r); !r)
```

Aliasing

A value of type $\mathrm{ref}\ \mathrm{T}$ is a *pointer* to a cell holding a value of type T.



If this value is "*copied*" by assigning it to another variable, the cell pointed to is not copied. (*r* and *s* are *aliases*)



So we can change r by assigning to s:

(s:=10; !r)

Aliasing all around us

Reference cells are *not the only language feature* that introduces the possibility of aliasing

- arrays
- communication channels
- I/O devices (disks, etc.)

The difficulties of aliasing

The possibility of aliasing *invalidates* all sorts of useful forms of *reasoning about programs*, both *by programmers*...

e.g., function

$$\lambda r{:}\,Ref\ Nat.\,\lambda s{:}\,Ref\ Nat.\,(r \coloneqq 2;\ s \coloneqq 3;\ !r)$$

always returns 2 unless r and s are aliases.

... and *by compilers*:

Code motion out of loops, *common sub-expression elimination*, *allocation of variables to registers*, and *detection of uninitialized variables* all depend upon the compiler knowing which objects a load or a store operation could reference.

High-performance compilers ***spend significant energy*** on *alias analysis* to try to establish when different variables cannot possibly refer to the same storage.

The benefits of aliasing

The *problems of aliasing* have led some language designers simply to disallow it (e.g., Haskell).

However, there are good reasons why most languages do provide constructs involving aliasing:

- efficiency (e.g., arrays)
- "action at a distance" (e.g., symbol tables)
- shared resources (e.g., locks) in concurrent systems
- ……

Example

$c = ref\ 0$

$\text{incc} = \lambda x{:}\, Unit.\,(c := succ(!\,c);\,!\,c)$

$\text{decc} = \lambda x{:}\, Unit.\,(c := pred(!\,c);\,!\,c)$

$\text{incc}\ unit$

$decc\ unit$

$\text{o} = \{\text{i} = incc, d = decc\}$

$let\ newcounter = o$

$\lambda_{.Unit}.$

$\text{let}\ c\ =\ ref\ 0\ \text{in}$

$\text{let incc} = \lambda x{:}\, Unit.\,(c := succ(!\,c);\,!\,c)\ \text{in}$

$\text{let decc} = \lambda x{:}\, Unit.\,(c := pred(!\,c);\,!\,c)$

$\text{let o} = \{i = incc, d = decc\}\ \text{in}$

o

How to enrich the language with the new mechanism？

Syntax

`t ::=`		*terms*
	`unit`	*unit constant*
	`x`	*variable*
	`λx:T.t`	*abstraction*
	`t t`	*application*
	`ref t`	*reference creation*
	`!t`	*dereference*
	`t:=t`	*assignment*

... plus other familiar types, in examples.

Typing rules

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \mathtt{ref}\ \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1} \qquad \text{(T-Ref)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1}{\Gamma \vdash \,!\mathtt{t}_1 : \mathtt{T}_1} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_1}{\Gamma \vdash \mathtt{t}_1 \mathtt{:=} \mathtt{t}_2 : \mathtt{Unit}} \qquad \text{(T-Assign)}$$

type system

a set of rules that assigns a property called type to the various “constructs “ of a computer program, such as *variables*, *expressions*, *functions* or *module*

Example

```
NatArray = Ref (Nat→Nat);

newarray = λ_:Unit. ref (λn:Nat.0);
         : Unit → NatArray

lookup = λa:NatArray. λn:Nat. (!a) n;
       : NatArray → Nat → Nat

update = λa:NatArray. λm:Nat. λv:Nat.
           let oldf = !a in
           a := (λn:Nat. if equal m n then v else oldf n);
       : NatArray → Nat → Nat → Unit
```

Evaluation

What is the value of the expression ref 0 ?

Crucial observation: evaluating ref 0 must *do* something ?

Is

r = ref 0

s = ref 0

and

r = ref 0

s = r

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**?

- *Concretely*: an array of *8-bit bytes*, indexed by 32/64-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 *value*s are *term*s ...

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 *runtime 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 (with references)

- *depends on the store* in which it is evaluated.
- *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 as well as a store* to *a reduced term* and *a new store*.

$$t \mid \mu \rightarrow t' \mid \mu'$$

To use the metavariable μ to *range over stores*.

Evaluation

An assignment $t_1 \coloneqq t_2$ first evaluates t_1 and t_2 *until they become values* ...

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{t}_1 \mathtt{:=} \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_1' \mathtt{:=} \mathtt{t}_2 \mid \mu'} \qquad \text{(E-Assign1)}$$

$$\frac{\mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_2' \mid \mu'}{\mathtt{v}_1 \mathtt{:=} \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{v}_1 \mathtt{:=} \mathtt{t}_2' \mid \mu'} \qquad \text{(E-Assign2)}$$

... and then returns unit and updates the store:

$$l \mathtt{:=} \mathtt{v}_2 \mid \mu \longrightarrow \mathtt{unit} \mid [l \mapsto \mathtt{v}_2]\mu \qquad \text{(E-Assign)}$$

Evaluation

A term of *the form* $\text{ref } t_1$

1. first *evaluates* inside t_1 *until it becomes a value* ...

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

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

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

Evaluation

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

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

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

Evaluation

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

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

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

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

Aside

Garbage Collection

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

It may not be problematic for most *theoretical purposes*, whereas it is clear that for *practical purposes* some form of *deallocation* of unused storage must be provided.

Pointer Arithmetic

```
p++;
```

We can’t do any!

Store Typing

Typing Locations

Question: What is the *type of a location*?

Answer: Depends on the ***contents*** of the store!

For example,
in the store $(l_1 \longmapsto \text{unit}, l_2 \longmapsto \text{unit})$, the term $!\,l_2$ is evaluated to unit, having type Unit.

But in the store $(l_1 \longmapsto \text{unit}, l_2 \longmapsto \lambda x{:}\,\text{Unit}.\,x)$, the term $!\,l_2$ has type $\text{Unit} \rightarrow \text{Unit}$.

Typing Locations — first try

Roughly, to find the type of a location l, first *look up* the current contents of l in the store, and calculate the type T_1 of the contents:

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

More precisely, to make the type of a term depend on the store (keeping a consistent state), we should change the *typing relation* from *three-place* to :

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

i.e., typing is now a *four-place relation* (about *contexts*, *stores*, *terms*, and *types*), though ***the store is a part of the context*** ……

Problems #1

However, this rule is not *completely satisfactory*, and is *rather inefficient*.

- First of all, it can make *typing derivations very large* (if a location *appears* ***many times*** in a term) !
- e.g., if

$$\mu = (l_1 \mapsto \lambda x{:}\,\mathrm{Nat}.\ 999,$$
$$l_2 \mapsto \lambda x{:}\,\mathrm{Nat}.\ (!\,l_1)\ x,$$
$$l_3 \mapsto \lambda x{:}\,\mathrm{Nat}.\ (!\,l_2)\ x,$$
$$l_4 \mapsto \lambda x{:}\,\mathrm{Nat}.\ (!\,l_3)\ x,$$
$$l_5 \mapsto \lambda x{:}\,\mathrm{Nat}.\ (!\,l_4)\ x),$$

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

Problems #2

But wait... it *gets worse* if the store contains a *cycle*. Suppose

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

how big is the typing derivation for $!\,l_2$?

Calculating a type for l_2 requires finding the type of l_1, which in turn involves l_2.

Why?

What leads to the problems?

Our typing rule for locations requires us to *recalculate the type of a location every time it's* mentioned in a term, which should not be necessary.

In fact, once a location is first created, *the type of the initial value* is **known**, and *the type will be kept* even if the values can be changed.

Store Typing

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.*

Store Typing

E.g., for

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

A reasonable *store typing* would be

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

Store Typing

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.

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

i.e., *typing* is now a *four-place relation on* contexts, *store typings*, terms, and types.

Proviso: the typing rules accurately predict the results of evaluation *only if* the *concrete store* used during evaluation actually *conforms to* the store typing.

Final typing rules

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

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

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

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

Store Typing

Question: Where do *these store typings* come from?

Answer: When we first typecheck a program, there will be no explicit locations, so we can use *an empty store typing*, since the locations arise only in terms that are *the intermediate results* of evaluation.

So, when a new location is created during evaluation,

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

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

Store Typing

As evaluation proceeds and *new locations are created*, *the store typing is extended* by looking at the type of the initial values being placed in newly allocated cells.

Σ only records the *association*

between

already-allocated storage cells and

their types.

Safety

Coherence between

the statics and the dynamics

Well-formed programs are well-behaved

Preservation

the steps of evaluation

preserve

typing

Preservation

How to express the statement of preservation?

First attempt: just add *stores* and *store typings* in the appropriate places.

Theorem(?): if $\Gamma \mid \Sigma \vdash t : T$ and $t|\mu \longrightarrow t'|\mu'$, then $\Gamma \mid \Sigma \vdash t' : T$

Right??

Wrong!

Why wrong?

Because Σ and μ here are not constrained to have anything to do with each other!

Exercise: Construct an example that breaks this statement of preservation

Preservation

Definition: A store μ is said to be *well typed* with respect to a typing context Γ and a store typing Σ, written $\Gamma \mid \Sigma \vdash \mu$, if $dom(\mu) = dom(\Sigma)$ and $\Gamma \mid \Sigma \vdash \mu(l){:}\ \Sigma(l)$ for every $\mathrm{l} \in dom(\mu)$.

Theorem (?) : if

$$\Gamma \mid \Sigma \vdash \mathrm{t}: \mathrm{T}$$
$$\mathrm{t} \mid \mu \longrightarrow \mathrm{t}' \mid \mu'$$
$$\Gamma \mid \Sigma \vdash \mu$$

then $\Gamma \mid \Sigma \vdash \mathrm{t}': \mathrm{T}$

Right this time?

Still wrong !

Why? Where?

Preservation

Creation of a *new reference cell* ...

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

... *breaks the correspondence* between the store typing and the store.

Since *the store can grow during evaluation*:

Creation of a new reference cell yields a store with a *larger domain* than the initial one, making the conclusion *incorrect*: if μ' includes a binding for ***a fresh location*** l, then l cann't be in the domain of Σ, and it will not be the case that $\boldsymbol{t'}$ ***is typable under*** $\boldsymbol{\Sigma}$.

Preservation

Theorem: if

$$\Gamma \mid \Sigma \vdash t : T$$
$$\Gamma \mid \Sigma \vdash \mu$$
$$t \mid \mu \longrightarrow t' \mid \mu'$$

then, for *some* $\Sigma' \supseteq \Sigma$,

$$\Gamma \mid \Sigma' \vdash t' : T$$
$$\Gamma \mid \Sigma' \vdash \mu'.$$

A correct version.

What is Σ' ?

Proof: Easy extension of the preservation proof for $\lambda_{\rightarrow}$.

Progress

well-typed expressions are either

values or can be

further evaluated

Progress

Theorem:

Suppose t is a closed, well-typed term

(i.e., $\Gamma | \Sigma \vdash \mathrm{t}: T$ for some T and Σ)

then either t is a *value* or else, for any store μ such that $\Gamma | \Sigma \vdash \mu$, there is some term t' and store μ' with

$$\mathrm{t} \mid \mu \longrightarrow \mathrm{t}' \mid \mu'$$

Safety

- preservation and progress together constitute the proof of safety
  - progress theorem ensures that well-typed expressions don't get stuck in an ill-defined state, and
  - preservation theorem ensures that if a step is a taken the result remains well-typed (*with the same type*).
- These two parts ensure the *statics and dynamics* are coherent, and that no ill-defined states can ever be encountered while evaluating a well-typed expression

In summary …

Syntax

We added to $\lambda_{\rightarrow}$ (with Unit) syntactic forms for *creating*, *dereferencing*, and *assigning* reference cells, plus a new type constructor Ref.

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

Evaluation

Evaluation becomes *a* *four-place* relation: $t \mid \mu \longrightarrow t' \mid \mu'$

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

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

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

Typing

Typing becomes *a three-place* relation: $\Gamma \mid \Sigma \vdash \mathtt{t} : \mathtt{T}$

$$\frac{\Sigma(l) = \mathtt{T}_1}{\Gamma \mid \Sigma \vdash l : \mathtt{Ref}\ \ \mathtt{T}_1} \qquad \text{(T-Loc)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \mid \Sigma \vdash \mathtt{ref}\ \ \mathtt{t}_1 : \mathtt{Ref}\ \ \mathtt{T}_1} \qquad \text{(T-Ref)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \ \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash \mathtt{!t}_1 : \mathtt{T}_{11}} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \ \mathtt{T}_{11} \qquad \Gamma \mid \Sigma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash \mathtt{t}_1\mathtt{:=}\mathtt{t}_2 : \mathtt{Unit}} \qquad \text{(T-Assign)}$$

Preservation

Theorem: if

$$\Gamma \mid \Sigma \vdash t : T$$

$$\Gamma \mid \Sigma \vdash \mu$$

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

then, for some $\Sigma' \supseteq \Sigma$,

$$\Gamma \mid \Sigma' \vdash t' : T$$

$$\Gamma \mid \Sigma' \vdash \mu'.$$

Progress

Theorem: Suppose t is a *closed, well-typed* term (that is, $\emptyset \mid \Sigma \vdash t : T$ for some T and Σ). Then either t is a value or else, for any store μ such that $\emptyset \mid \Sigma \vdash \mu$, there is some term t' and store μ' with $t \mid \mu \longrightarrow t' \mid \mu'$.

Others …

Arrays

Fix-sized vectors of values. All of the values must have the *same type*, and the fields in the array can be accessed and modified.

e.g., arrays can be created with in Ocaml

$[|e_1; \ldots ; e_n|]$

```
# let a = [|1;3;5;7;9|];;
val a : int array = [|1;3;5;7;9|]
#a;;
-: int array = [|1;3;5;7;9|]
```

Recursion via references

Indeed, we can define *arbitrary recursive functions* using references

1. Allocate a ref cell and initialize it with a *dummy function* of the appropriate type:

$$\text{fact}_{ref} \;=\; \text{ref}\,(\lambda\text{n: Nat. }0)$$

2. Define *the body of the function* we are interested in, using *the contents of the reference cell* for making recursive calls:

$$\text{fact}_{body} =$$
$$\lambda\text{n: Nat.}$$
$$\text{if iszero n then 1 else times n }((!\,\text{fact}_{ref})(\text{pred n}))$$

3. "Backpatch" by storing the real body into the reference cell:

$$\text{fact}_{ref} := \text{fact}_{body}$$

4. Extract the contents of the reference cell and use it as desired:

$$\text{fact} \;=\; !\,\text{fact}_{ref}$$
$$\text{fact } 5$$

Homework☺

- Read chapter 13
- Read and chew over the codes of *fullref*.

- HW: 13.3.1 and 13.5.2

- Preview chapter 14

Non-termination via references

There are *well-typed terms* in this system that are not strongly normalizing. For example:

$$t1 = \lambda r{:}\,\mathrm{Ref}\,(\mathrm{Unit} \to \mathrm{Unit}).$$
$$(r := (\lambda x{:}\,\mathrm{Unit}.\,(!\,r)x);$$
$$(!\,r)\ \mathrm{unit});$$
$$t2 = \mathrm{ref}\,(\lambda x{:}\,\mathrm{Unit}.\,x);$$

Applying t1 to t2 yields a (well-typed) divergent term.

Nontermination via references

There are well-typed terms in this system that are not strongly normalizing. For example:

$$t1 = \lambda r{:}\,\mathrm{Ref}\,(\mathrm{Unit} \to \mathrm{Unit}).$$
$$(r := (\lambda x{:}\,\mathrm{Unit}.\,(!\,r)x);$$
$$(!\,r)\ \mathrm{unit});$$
$$t2 = \mathrm{ref}\,(\lambda x{:}\,\mathrm{Unit}.\,x);$$

Applying t1 to t2 yields a (well-typed) divergent term.