Recursion

In λ!, all programs terminate (Cf. Chapter 12.)

Example

```plaintext
iseven = fix ff;

iseven = fix ff;

iseven = fix ff;

iseven = fix ff;

iseven = fix ff;

iseven = fix ff;

iseven = fix ff;
```

But we can extend the system with a (typed) fixed-point operator...

Hence, untyped terms like \( \omega \) and \( \text{fix} \) are not typable.

Recursion in \( \lambda! \)
Amore convenient form

\[
\begin{align*}
\text{let rec } & \text{seven : Nat} = \text{let } x = \text{fix} (\forall x : \text{Nat}. \text{seven} (\text{pred} x)) \text{ in } \text{seven} 7 \\
\text{let } & \text{seven } = \text{let } x = \text{fix} (\forall x : \text{Nat}. \text{seven} (\text{pred} x)) \text{ in } \text{seven} 7 \\
\text{let } & \text{seven } = \text{let } x = \text{fix} (\forall x : \text{Nat}. \text{seven} (\text{pred} x)) \text{ in } \text{seven} 7
\end{align*}
\]

**New evaluation rules**

\[
\begin{align*}
\text{fix } t & \rightsquigarrow t_0 \\
\text{fix } t & \rightsquigarrow t_1 \quad \forall t_1. \text{fix}(\forall t_1. t_1) \equiv t_1 \\
\text{fix } t & \rightsquigarrow t_2 \quad \forall t_2. \text{fix}(\forall t_2. t_2) \equiv t_2
\end{align*}
\]

**New typing rules**

\[
\begin{align*}
\forall t_1. t_1 & : T_1 \\
\forall t_1. t_1 & : T_1 \quad \forall t_1. t_1 \equiv T_1 \\
\forall t_1. t_1 & : T_1 \quad \forall t_1. t_1 \equiv T_1 \\
\forall t_1. t_1 & : T_1 \quad \forall t_1. t_1 \equiv T_1
\end{align*}
\]
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 contrast to the features of immutable variables, which are referred to by a variable.

**Mutability**

In some languages (e.g., OCaml), these two features are kept separate. Variables are only for naming — the binding between a variable and its value is immutability, a new value may be assigned to a reference (dereferenced to obtain this value). At any given moment, a reference holds a value (and can be introspected); a new class of mutable values (called reference cells or references) can be introduced. A value of type RefT is a pointer to a cell holding a value of type T.

**Aliasing**

A value of type RefT 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. So we can change x by assigning to z:

```latex
code
basicExamples
\begin{equation}
\text{s} = \text{s} = \text{j}
\end{equation}
```

A value of type RefT 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. So we can change x by assigning to z:

```latex
\text{s} = \text{s} = \text{j}
\end{equation}
```

A value of type RefT 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. So we can change x by assigning to z:

```latex
\text{s} = \text{s} = \text{j}
\end{equation}
```

A value of type RefT 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. So we can change x by assigning to z:

```latex
\text{s} = \text{s} = \text{j}
\end{equation}
```

A value of type RefT 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. So we can change x by assigning to z:

```latex
\text{s} = \text{s} = \text{j}
\end{equation}
```

A value of type RefT 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. So we can change x by assigning to z:

```latex
\text{s} = \text{s} = \text{j}
\end{equation}
```

A value of type RefT 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. So we can change x by assigning to z:

```latex
\text{s} = \text{s} = \text{j}
\end{equation}
```

A value of type RefT 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. So we can change x by assigning to z:

```latex
\text{s} = \text{s} = \text{j}
\end{equation}
```

A value of type RefT 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. So we can change x by assigning to z:

```latex
\text{s} = \text{s} = \text{j}
\end{equation}
```

A value of type RefT 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. So we can change x by assigning to z:

```latex
\text{s} = \text{s} = \text{j}
\end{equation}
```

A value of type RefT 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. So we can change x by assigning to z:

```latex
\text{s} = \text{s} = \text{j}
\end{equation}
```

A value of type RefT 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. So we can change x by assigning to z:

```latex
\text{s} = \text{s} = \text{j}
\end{equation}
```
Reference cells are not the only language feature that introduces the possibility of aliasing. Arrays, communication channels, I/O devices (disks, etc.)...etc.

The difficulties of aliasing

The problem of aliasing has led some language designers simply to disallow it.

Example

```plaintext
Example

c = ref 0
incc = x: Unit.(c := succ(!c); !c)
dec = x: Unit.(c := pred(!c); !c)
dec = x: Unit.(c := succ(!c); !c)
0 = ref 0
```

The benefits of aliasing

There are good reasons why most languages do provide constructs involving aliasing:

...etc.

The possibility of aliasing introduces all sorts of useful forms of reasoning...etc.

The difficulties of aliasing

Reference cells are not the only language feature that introduces the possibility of aliasing...etc.
```
let newcounter = (_, Unit) in
let inc = (x: Unit) =>
  let c = ref 0 in
  let c = succ(!c) in
  !c
in
let dec = (x: Unit) =>
  let c = ref 0 in
  let c = pred(!c) in
  !c
in
let o = {i = inc, d = dec} in
```

### Syntax

- `t ::= t1 : T1`
- `t1 : T1` (reference creation)
- `!t1 : T1` (dereference)
- `t2 := t1 : Unit` (assignment)
- `t1 : RefT1` (plus other familiar types, in examples.)

### Typing Rules

- **T-Assign**
  \[
  \Gamma |- t1 : T1 \\
  \Gamma |- t2 : Unit \\
  \Gamma |- t1 := t2 : Unit
  \]
- **T-Deref**
  \[
  \Gamma |- t1 : RefT1 \\
  \Gamma |- !t1 : T1
  \]
- **T-Ref**
  \[
  \Gamma |- t1 : RefT1 \\
  \Gamma |- t1 : T1
  \]

### Another example

```
val BoolArray = Ref(Nat -> Bool); val newarray = (_, Unit).ref(n:Nat. false); val lookup = (a: BoolArray). (n:Nat. !a)n; val update = (a: BoolArray). (m:Nat. v:Bool). let oldf = !a in
  a := (n:Nat. if equal mn then v else oldf n);
```

### Let

```
let counter = 0 in
```

### Syntax

```
let newcounter = (_, Unit).
```
So what is a reference?

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

would behave the same.

\[
\begin{align*}
    \text{ref} & = s \\
    \text{r} & = \text{ref} \\
\end{align*}
\]

and

\[
\begin{align*}
    \text{ref} & = 0 \\
    \text{r} & = \text{ref} \\
\end{align*}
\]

Otherwise.

Crucial observation: evaluating ref 0 must do something.

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

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. In the big-step style, 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.

Does this mean we are going to allow programmers to write explicit locations to the store? No. This is just a modeling trick. We are enriching the "source language" to model some runtime structures, so that we can continue to formalize evaluation as a relation between source terms.

Aside: Does this mean all values are terms?...

Syntax of Terms

\[ \begin{align*}
& \text{Terms} \\
& \text{more location} \\
& \quad \text{assignment} \\
& \quad \text{dereference} \\
& \quad \text{reference creation} \\
& \quad \text{application} \\
& \quad \text{abstraction} \\
& \quad \text{variable} \\
& \quad \text{unit and constant values} \\
& \quad 1 \\
& \quad \text{t} \leftarrow t \\
& \quad t \\
& \quad \text{term} \\
& \quad \text{new t} \\
& \quad 0 \\
& \quad \text{f} \\
& \quad \text{var t} \\
& \quad \text{x: t} \leftarrow \text{x} \\
& \quad \text{unit} \\
& \quad \text{r} \\
& \end{align*} \]
An assignment to a variable inside $t_1$ until it becomes a value...

...and then looks up this value (which must be a location) in the current store:

\[ l = v \]

\[ (E-DerefLoc) \]

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

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

...and then returns this value (which must be a location) in the current store:

\[ l = v \]

\[ (E-DerefLoc) \]

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

...and then returns this value (which must be a location) in the current store:

\[ l = v \]

\[ (E-DerefLoc) \]

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

...and then returns this value (which must be a location) in the current store:

\[ l = v \]

\[ (E-DerefLoc) \]

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

...and then returns this value (which must be a location) in the current store:

\[ l = v \]

\[ (E-DerefLoc) \]

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

...and then returns this value (which must be a location) in the current store:

\[ l = v \]

\[ (E-DerefLoc) \]

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

...and then returns this value (which must be a location) in the current store:

\[ l = v \]

\[ (E-DerefLoc) \]

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

...and then returns this value (which must be a location) in the current store:

\[ l = v \]

\[ (E-DerefLoc) \]

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

...and then returns this value (which must be a location) in the current store:

\[ l = v \]

\[ (E-DerefLoc) \]

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

...and then returns this value (which must be a location) in the current store:

\[ l = v \]

\[ (E-DerefLoc) \]
Aside: garbage collection

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

Aside: pointer arithmetic

Q: What is the type of a location?

Typing Locations

A: It depends on the store!

E.g., in the store (l17!unit; l27!unit), the term !l2 has type Unit.

But in the store (l17!unit; l27!x:Unit), the term !l2 has type Unit!Unit.
Q: What is the type of a location?
A: It depends on the store!

E.g., in the store \((l_1 \in \text{unit}; l_2 \in \text{unit})\), the \(l_1\) has type Unit.

But in the store \((l_1 \in \text{unit}; l_2 \in \text{unit}; t \in \text{unit})\), the \(l_1\) has type Unit.

A: It depends on the store.

Q: Where is the type of a location?


Problem!

But wait... it gets worse. Suppose

Now how big is the typing derivation for

Observation: The typing rules we have chosen for references guarantee that a

Store Typings

A reasonable store typing would be

Now, suppose we are given a store typing describing the store in which we

Store Typings

from locations to types.

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

Observation: The typing rules we have chosen for references guarantee that a

Problem

But wait... It gets worse. Suppose
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, we can observe the type of \( v_1 \) and extend the "current store typing"

\[
\text{E-RefV} \quad \text{(E-RefV)}
\]

appropriately.

\[
\text{refv_1} \ l \ (v_1 \rightarrow \ \{\text{dom} \not\in 1\})
\]

we can observe the type of \( v_1 \) and extend the "current store typing"

\[
\text{T-Assign} \quad \text{(T-Assign)}
\]

\[
\text{assign} : \; t_1 \rightarrow t_2 \mid L
\]

\[
\text{T-Deref} \quad \text{(T-Deref)}
\]

\[
\text{deref} : \; t_1 \rightarrow t_2 \mid L
\]

\[
\text{T-Ref} \quad \text{(T-Ref)}
\]

\[
\text{ref} : \; t_1 \rightarrow t_2 \mid L
\]

\[
\text{T-Ref} \quad \text{(T-Ref)}
\]

\[
\text{ref} : \; t_1 \rightarrow t_2 \mid L
\]

\[
\text{T-Ref} \quad \text{(T-Ref)}
\]

\[
\text{ref} : \; t_1 \rightarrow t_2 \mid L
\]

\[
\text{T-Ref} \quad \text{(T-Ref)}
\]

\[
\text{ref} : \; t_1 \rightarrow t_2 \mid L
\]

\[
\text{T-Ref} \quad \text{(T-Ref)}
\]

\[
\text{ref} : \; t_1 \rightarrow t_2 \mid L
\]

\[
\text{T-Ref} \quad \text{(T-Ref)}
\]

\[
\text{ref} : \; t_1 \rightarrow t_2 \mid L
\]

\[
\text{T-Ref} \quad \text{(T-Ref)}
\]

\[
\text{ref} : \; t_1 \rightarrow t_2 \mid L
\]