Type Systems
Seminar 6

Mehmet Ali Arslan

Slides mostly “borrowed” from Martin Odersky

Plan

PREVIOUSLY: unit, sequencing, let, pairs, sums

TODAY:

1. recursion
2. state
3. ???

NEXT: exceptions?
NEXT: polymorphic (not so simple) typing

Recursion

Recursion in untyped λ-calculus

- We apply a *fixed-point combinator* to a *generator* function
- $\mathtt{fix} = \lambda \mathtt{f}.\, (\lambda \mathtt{x}.\, \mathtt{f}\, (\lambda \mathtt{y}.\, \mathtt{x}\, \mathtt{x}\, \mathtt{y}))\, (\lambda \mathtt{x}.\, \mathtt{f}\, (\lambda \mathtt{y}.\, \mathtt{x}\, \mathtt{x}\, \mathtt{y}))$

Example - Factorial

- $\mathtt{g} = \lambda \mathtt{fct}.\,(\lambda \mathtt{n}.\,\mathtt{if}\;\mathtt{realeq}\;\mathtt{n}\;\mathtt{c}_0\;\mathtt{then}\;\mathtt{c}_1$
  $\mathtt{else}\;(\mathtt{times}\;\mathtt{n}(\mathtt{fct}\;(\mathtt{prd}\;(\mathtt{prd}\;\mathtt{n})));$
- $\mathtt{factorial} = \mathtt{fix}\;\mathtt{g};$
- whiteboard...

Recursion in STLC

- the term $\lambda \mathtt{y.\,x\,x\,y}$ in *fix* can not be typed in STLC
- so we add it as a primitive

Example

```
ff = λie:Nat→Bool.
       λx:Nat.
         if iszero x then true
         else if iszero (pred x) then false
         else ie (pred (pred x));

iseven = fix ff;

iseven 7;
```

New syntactic forms

$\texttt{t} ::= \ldots$ *terms*

$\quad\quad \texttt{fix t}$ *fixed point of* $\texttt{t}$

New evaluation rules $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\begin{array}{c}\texttt{fix } (\lambda\texttt{x:T}_1\texttt{.t}_2) \\ \longrightarrow [\texttt{x} \mapsto (\texttt{fix } (\lambda\texttt{x:T}_1\texttt{.t}_2))]\texttt{t}_2\end{array} \qquad \text{(E-FixBeta)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{fix t}_1 \longrightarrow \texttt{fix t}_1'} \qquad \text{(E-Fix)}$$

New typing rules

$$\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \to \mathtt{T}_1}{\Gamma \vdash \mathtt{fix}\ \mathtt{t}_1 : \mathtt{T}_1} \qquad \text{(T-Fix)}$$

A more convenient form

$$\texttt{letrec x:}\texttt{T}_1\texttt{=}\texttt{t}_1\texttt{ in t}_2 \stackrel{\text{def}}{=} \texttt{let x = fix (}\lambda\texttt{x:}\texttt{T}_1\texttt{.}\texttt{t}_1\texttt{) in t}_2$$

```
letrec iseven : Nat→Bool =
  λx:Nat.
    if iszero x then true
    else if iszero (pred x) then false
    else iseven (pred (pred x))
in
  iseven 7;
```

References

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

We choose OCaml's style, which is easier to work with formally.

So a variable of type `T` in most languages (except OCaml) will correspond to a `Ref T` (actually, a `Ref(Option T)`) here.

Basic Examples

```
r = ref 5

!r

r := 7

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

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

Basic Examples

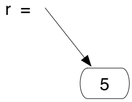
```
r = ref 5

!r

r := 7

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

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

i.e.,

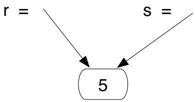
```
((((r:=succ(!r); r:=succ(!r)); r:=succ(!r));
  r:=succ(!r)); !r)
```

Aliasing

A value of type `Ref 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.



So we can change `r` by assigning to `s`:

```
(s:=6; !r)
```

Aliasing all around us

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

- object references
- explicit pointers in C
- arrays
- communication channels
- I/O devices (disks, etc.)

Example

```
c = ref 0
incc = λx:Unit. (c := succ (!c); !c)
decc = λx:Unit. (c := pred (!c); !c)
incc unit
decc unit
o = {i = incc, d = decc}
```

Syntax

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

... plus other familiar types, in examples.

Typing Rules

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

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

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

Final 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`?

Evaluation

What is the *value* of the expression `ref 0`?

Crucial observation: evaluating `ref 0` must *do* something.

Otherwise,

```
r = ref 0
s = ref 0
```

and

```
r = ref 0
s = r
```

would behave the same.

Evaluation

What is the *value* of the expression `ref 0`?
Crucial observation: evaluating `ref 0` must *do* something.
Otherwise,

```
r = ref 0
s = ref 0
```

and

```
r = ref 0
s = r
```

would behave the same.
Specifically, evaluating `ref 0` should *allocate some storage* and yield a *reference* (or *pointer*) to that storage.

Evaluation

What is the *value* of the expression `ref 0`?
Crucial observation: evaluating `ref 0` must *do* something.
Otherwise,

```
r = ref 0
s = ref 0
```

and

```
r = ref 0
s = r
```

would 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?

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.

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*

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*.

Locations

Syntax of values:

$\mathtt{v}$ ::=		*values*
	$\mathtt{unit}$	*unit constant*
	$\lambda\mathtt{x{:}T.t}$	*abstraction value*
	l	*store location*

... and since all values are terms...

Syntax of Terms

`t ::=`		*terms*
	`unit`	*unit constant*
	`x`	*variable*
	λ`x: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 run-time 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 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.
I.e., the evaluation relation should now map a term and a store to a reduced term and a new store.

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

We use the metavariable μ to range over stores.

Evaluation

An assignment $\mathtt{t}_1\mathtt{:=}\mathtt{t}_2$ first evaluates $\mathtt{t}_1$ and $\mathtt{t}_2$ 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 $\mathtt{unit}$ and updates the store:

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

A term of the form $\texttt{ref t}_1$ first evaluates inside $\texttt{t}_1$ until it becomes a value...

$$\frac{\texttt{t}_1 \mid \mu \longrightarrow \texttt{t}_1' \mid \mu'}{\texttt{ref t}_1 \mid \mu \longrightarrow \texttt{ref t}_1' \mid \mu'} \qquad \text{(E-Ref)}$$

... and then chooses (allocates) a fresh location l, augments the store with a binding from l to $\texttt{v}_1$, and returns l:

$$\frac{l \notin dom(\mu)}{\texttt{ref v}_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto \texttt{v}_1)} \qquad \text{(E-RefV)}$$

A term $\texttt{!t}_1$ first evaluates in $\texttt{t}_1$ until it becomes a value...

$$\frac{\texttt{t}_1 \mid \mu \longrightarrow \texttt{t}_1' \mid \mu'}{\texttt{!t}_1 \mid \mu \longrightarrow \texttt{!t}_1' \mid \mu'} \qquad \text{(E-Deref)}$$

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

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

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

$$\frac{\mathtt{t}_1 |\ \mu \longrightarrow \mathtt{t}_1' |\ \mu'}{\mathtt{t}_1\ \ \mathtt{t}_2 |\ \mu \longrightarrow \mathtt{t}_1'\ \ \mathtt{t}_2 |\ \mu'} \qquad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 |\ \mu \longrightarrow \mathtt{t}_2' |\ \mu'}{\mathtt{v}_1\ \ \mathtt{t}_2 |\ \mu \longrightarrow \mathtt{v}_1\ \ \mathtt{t}_2' |\ \mu'} \qquad \text{(E-App2)}$$

$$(\lambda\mathtt{x}{:}\mathtt{T}_{11}.\mathtt{t}_{12})\ \ \mathtt{v}_2 |\ \mu \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} |\ \mu \qquad \text{(E-AppAbs)}$$

Aside: garbage collection

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

Aside: pointer arithmetic

We can't do any!

Store Typings

Typing Locations

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

Typing Locations

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

A: It depends on the store!

E.g., in the store $(l_1 \mapsto \texttt{unit}, l_2 \mapsto \texttt{unit})$, the term $\texttt{!}l_2$ has type $\texttt{Unit}$.

But in the store $(l_1 \mapsto \texttt{unit}, l_2 \mapsto \lambda\texttt{x:Unit.x})$, the term $\texttt{!}l_2$ has type $\texttt{Unit}\to\texttt{Unit}$.

Typing Locations — first try

Roughly:

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

Typing Locations — first try

Roughly:

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

More precisely:

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

I.e., typing is now a *four*-place relation (between contexts, *stores*, terms, and types).

Problem

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

E.g., if

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

then how big is the typing derivation for $\texttt{!}l_5$?

Problem!

But wait... it gets worse. Suppose

$$\begin{aligned}(\mu = l_1 &\mapsto \lambda\texttt{x:Nat. }\texttt{!}l_2\ \texttt{x},\\ l_2 &\mapsto \lambda\texttt{x:Nat. }\texttt{!}l_1\ \texttt{x}),\end{aligned}$$

Now how big is the typing derivation for $\texttt{!}l_2$?

Store Typings

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.

E.g., for

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

A reasonable store typing would be

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

Now, suppose we are given a store typing Σ describing the store μ in which we intend to evaluate some term $\mathtt{t}$. Then we can use Σ to look up the types of locations in $\mathtt{t}$ instead of calculating them from the values in μ.

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

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

Final typing rules

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

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

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

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

Q: Where do these store typings come from?

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,

$$\frac{l \notin dom(\mu)}{\texttt{ref}\ \ \texttt{v}_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto \texttt{v}_1)} \qquad \text{(E-RefV)}$$

we can extend the "current store typing" with the type of $\texttt{v}_1$.

Safety

Preservation

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

Theorem (?): If $\Gamma \mid \Sigma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{t} \mid \mu \longrightarrow \mathtt{t}' \mid \mu'$, then $\Gamma \mid \Sigma \vdash \mathtt{t}' : \mathtt{T}$.

Preservation

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

> *Theorem (?):* If $\Gamma \mid \Sigma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{t} \mid \mu \longrightarrow \mathtt{t}' \mid \mu'$, then $\Gamma \mid \Sigma \vdash \mathtt{t}' : \mathtt{T}$. Wrong!

Why is this wrong?

Preservation

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

Theorem (?): If $\Gamma \mid \Sigma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{t} \mid \mu \longrightarrow \mathtt{t}' \mid \mu'$, then $\Gamma \mid \Sigma \vdash \mathtt{t}' : \mathtt{T}$. Wrong!

Why is this 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

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 $l \in dom(\mu)$.

Preservation

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 $l \in dom(\mu)$.

Next attempt:

Theorem (?): If

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

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

Preservation

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 $l \in dom(\mu)$.

Next attempt:

Theorem (?): If

$\Gamma \mid \Sigma \vdash \mathtt{t} : \mathtt{T}$

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

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

then $\Gamma \mid \Sigma \vdash \mathtt{t}' : \mathtt{T}$. Still wrong!

What's wrong now?

Preservation

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 $l \in dom(\mu)$.

Next attempt:

Theorem (?): If

$\Gamma \mid \Sigma \vdash \mathtt{t} : \mathtt{T}$

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

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

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

Still wrong!

Creation of a new reference cell...

$$\frac{l \notin dom(\mu)}{\mathtt{ref}\ \mathtt{v}_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto \mathtt{v}_1)} \qquad \text{(E-RefV)}$$

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

Preservation (correct version)

Theorem: If

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

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

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

Preservation (correct version)

Theorem: If

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

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

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

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

Progress

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