Mechanized Proofs of Type Safety for a Family of λ-Calculi with References

Michalis A. Papakyriakou
Nikolaos S. Papaspyrou

National Technical University of Athens
School of Electrical and Computer Engineering
Software Engineering Laboratory

{mpapakyr, nickie}@softlab.ntua.gr

2nd Informal ACM SIGPLAN Workshop on Mechanizing Metatheory
Freiburg, Germany, October 4, 2007

Outline

Where did we start from?

- Shao, Trifonov, Saha and Papaspyrou,
  "A Type System for Certified Binaries",
  ACM TOPLAS, vol. 27, no. 1, pp. 1-45, 2005.

Where did we start from?

- Shao, Trifonov, Saha and Papaspyrou,
  "A Type System for Certified Binaries",
  ACM TOPLAS, vol. 27, no. 1, pp. 1-45, 2005.
- The big picture:
  - low-level code with verifiable specifications
  - one type language: variant of the CIC — Coq
  - several computation languages
  - CL depends on TL, but not vice-versa
  - TL defines the logic and the type system of the CL

Where did we start from?

- Shao, Trifonov, Saha and Papaspyrou,
  "A Type System for Certified Binaries",
  ACM TOPLAS, vol. 27, no. 1, pp. 1-45, 2005.
- The big picture:
  - low-level code with verifiable specifications
  - one type language: variant of the CIC — Coq
  - several computation languages
  - CL depends on TL, but not vice-versa
  - TL defines the logic and the type system of the CL
- What is missing from the CL:
  - references and destructive update
  - recursive data types
  - ...

Where did we start from?

- Shao, Trifonov, Saha and Papaspyrou,
  "A Type System for Certified Binaries",
  ACM TOPLAS, vol. 27, no. 1, pp. 1-45, 2005.
- The big picture:
  - low-level code with verifiable specifications
  - one type language: variant of the CIC — Coq
  - several computation languages
  - CL depends on TL, but not vice-versa
  - TL defines the logic and the type system of the CL
- What is missing from the CL:
  - references and destructive update
  - recursive data types
  - ...

References (i)

ML-style references

- Reference allocation

  $\texttt{let}\ r = \texttt{new}\ 7\ \texttt{in} \ldots$ $\qquad$ r : ref int

- Assignment

  $\ldots r := 42 \ldots$ $\qquad$ destructive update!

- Dereference

  $\ldots \texttt{print}\ (\texttt{deref}\ r);$ $\qquad$ prints 42

References *(i)*

ML-style references

- Reference allocation

  $\texttt{let}\ r = \texttt{new}\ 7\ \texttt{in} \ldots$ — r : ref int
- Assignment

  $\ldots r := 42 \ldots$ — destructive update!
- Dereference

  $\ldots \texttt{print}\ (\texttt{deref}\ r);$ — prints 42
- No reference deallocation!

  $\ldots \texttt{free}\ r$ — use garbage collection!

References (i)

ML-style references

- Reference allocation

  `let` $r =$ `new` 7 `in` ... r : ref int

- Assignment

  $\ldots r := 42 \ldots$ destructive update!

- Dereference

  ...`print` (`deref` r); prints 42

- No reference deallocation!

  ...`free` r use garbage collection!

or do we want reference deallocation?

References *(ii)*

But ML-style references are not enough in TSCB!

- We use singleton types for reasoning about computed values
- Suppose we start with an initial value

  $\texttt{let}\ r = \texttt{new}\ \overline{7}\ \texttt{in} \ldots$ $\qquad r : \mathsf{ref}\ (\mathsf{sint}\ \widehat{7})$

References (ii)

But ML-style references are not enough in TSCB!

- We use singleton types for reasoning about computed values
- Suppose we start with an initial value
  $\texttt{let}\ r = \texttt{new}\ \overline{7}\ \texttt{in} \ldots$ $\quad r : \mathsf{ref}\ (\mathsf{sint}\ \widehat{7})$
- and then we want to change it
  $\ldots r := \overline{42} \ldots$ destructive update!

References *(ii)*

But ML-style references are not enough in TSCB!

- We use singleton types for reasoning about computed values
- Suppose we start with an initial value
  $\texttt{let}\ r = \texttt{new}\ \overline{7}\ \texttt{in} \ldots$ $\qquad r : \mathsf{ref}\ (\mathsf{sint}\ \widehat{7})$
- and then we want to change it
  $\ldots r := \overline{42} \ldots$ $\qquad$ destructive update!
- Type error!
  $r : \mathsf{ref}\ (\mathsf{sint}\ \widehat{7})$ $\qquad \overline{42} : \mathsf{sint}\ \widehat{42}$ $\qquad \mathsf{sint}\ \widehat{7} \neq \mathsf{sint}\ \widehat{42}$

References (ii)

But ML-style references are not enough in TSCB!

- We use singleton types for reasoning about computed values
- Suppose we start with an initial value
  $\texttt{let}\ r = \texttt{new}\ \overline{7}\ \texttt{in} \ldots$ $\qquad r : \mathsf{ref}\ (\mathsf{sint}\ \widehat{7})$
- and then we want to change it
  $\ldots r := \overline{42} \ldots$ $\qquad$ destructive update!
- Type error!
  $r : \mathsf{ref}\ (\mathsf{sint}\ \widehat{7})$ $\qquad \overline{42} : \mathsf{sint}\ \widehat{42}$ $\qquad \mathsf{sint}\ \widehat{7} \neq \mathsf{sint}\ \widehat{42}$
- Strong update: the type changes!

References (ii)

But ML-style references are not enough in TSCB!

- We use singleton types for reasoning about computed values
- Suppose we start with an initial value
  $\texttt{let}\ r = \texttt{new}\ \overline{7}\ \texttt{in} \ldots$ $\qquad r : \mathsf{ref}\ (\mathsf{sint}\ \widehat{7})$
- and then we want to change it
  $\ldots r := \overline{42} \ldots$ $\qquad$ destructive update!
- Type error!
  $r : \mathsf{ref}\ (\mathsf{sint}\ \widehat{7})$ $\qquad \overline{42} : \mathsf{sint}\ \widehat{42}$ $\qquad \mathsf{sint}\ \widehat{7} \neq \mathsf{sint}\ \widehat{42}$
- Strong update: the type changes!
- Can use weak update $\qquad r : \mathsf{ref}\ (\exists n : \mathbb{Z}.\ \mathsf{sint}\ n)$
  but then we cannot reason about r's value

References (iii)

How do we deal with this problem?

- Before changing the type of a reference, make sure that nobody else knows about it!

References *(iii)*

How do we deal with this problem?

- Before changing the type of a reference, make sure that nobody else knows about it!
- Linear (substructural) type systems

References (iii)

How do we deal with this problem?

- Before changing the type of a reference, make sure that nobody else knows about it!
- Linear (substructural) type systems
- a linear reference $r : {}^{\mathsf{L}}\mathsf{ref}\ \tau$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{new}\ e : {}^{\mathsf{L}}\mathsf{ref}\ \tau}\ \textit{(new)}$$

References (iii)

How do we deal with this problem?

- Before changing the type of a reference, make sure that nobody else knows about it!
- Linear (substructural) type systems
- a linear reference $r : {}^{\mathsf{L}}\mathsf{ref}\ \tau$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{new}\ e : {}^{\mathsf{L}}\mathsf{ref}\ \tau}\ \textit{(new)}$$

- an unrestricted reference $r : {}^{\mathsf{U}}\mathsf{ref}\ \tau$

$$\frac{\Gamma \vdash e : {}^{\mathsf{U}}\mathsf{ref}\ \tau}{\Gamma \vdash \texttt{deref}\ e : \tau}\ \textit{(deref)}$$

References *(iv)*

What do we get with a linear type system?

- Weak update — type is preserved

$$\frac{\Gamma \vdash e_1 : {}^{\mathsf{U}}\mathsf{ref}\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \mathsf{unit}} \ \textit{(weak)}$$

References (iv)

What do we get with a linear type system?

- Weak update — type is preserved

$$\frac{\Gamma \vdash e_1 : {}^{\mathsf{U}}\mathsf{ref}\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \mathsf{unit}} \ (\textit{weak})$$

- Strong update — type changes

$$\frac{\Gamma \vdash e_1 : {}^{\mathsf{L}}\mathsf{ref}\ \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 := e_2 : {}^{\mathsf{L}}\mathsf{ref}\ \tau'} \ (\textit{strong})$$

References (iv)

What do we get with a linear type system?

- Weak update — type is preserved

$$\frac{\Gamma \vdash e_1 : {}^{\mathsf{U}}\mathsf{ref}\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \mathsf{unit}}\ \textit{(weak)}$$

- Strong update — type changes

$$\frac{\Gamma \vdash e_1 : {}^{\mathsf{L}}\mathsf{ref}\ \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 := e_2 : {}^{\mathsf{L}}\mathsf{ref}\ \tau'}\ \textit{(strong)}$$

- Deallocation

$$\frac{\Gamma \vdash e : {}^{\mathsf{L}}\mathsf{ref}\ \tau}{\Gamma \vdash \mathtt{free}\ e : \mathsf{unit}}\ \textit{(free)}$$

References (iv)

What do we get with a linear type system?

- Weak update — type is preserved

$$\frac{\Gamma \vdash e_1 : {}^{\mathsf{U}}\mathsf{ref}\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \mathsf{unit}}\ \textit{(weak)}$$

- Strong update — type changes

$$\frac{\Gamma \vdash e_1 : {}^{\mathsf{L}}\mathsf{ref}\ \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 := e_2 : {}^{\mathsf{L}}\mathsf{ref}\ \tau'}\ \textit{(strong)}$$

- Deallocation

$$\frac{\Gamma \vdash e : {}^{\mathsf{L}}\mathsf{ref}\ \tau}{\Gamma \vdash \mathtt{free}\ e : \mathsf{unit}}\ \textit{(free)}$$

- But how do we convert a ${}^{\mathsf{L}}\mathsf{ref}\ \tau$ to a ${}^{\mathsf{U}}\mathsf{ref}\ \tau$?

Linear references (i)

The let! construct $\texttt{let!}\ (x)\ y = e_1\ \texttt{in}\ e_2$

- Temporarily converts a ${}^{\mathsf{L}}\mathsf{ref}\ \tau$ to a ${}^{\mathsf{U}}\mathsf{ref}\ \tau$

Linear references *(i)*

The let! construct $\texttt{let!}\ (x)\ y = e_1\ \texttt{in}\ e_2$

- Temporarily converts a ${}^{\mathsf{L}}\mathsf{ref}\ \tau$ to a ${}^{\mathsf{U}}\mathsf{ref}\ \tau$
- Example

$\texttt{let}\ r = \texttt{new}\ 6\ \texttt{in}$ $\qquad r : {}^{\mathsf{L}}\mathsf{ref\ int}$

Linear references (i)

The let! construct $\texttt{let!}\ (x)\ y = e_1\ \texttt{in}\ e_2$

- Temporarily converts a ${}^{\mathsf{L}}\mathsf{ref}\ \tau$ to a ${}^{\mathsf{U}}\mathsf{ref}\ \tau$
- Example

$\texttt{let}\ r = \texttt{new}\ 6\ \texttt{in}$ $\qquad r : {}^{\mathsf{L}}\mathsf{ref\ int}$

$\texttt{let!}\ (r)$ $\qquad r : {}^{\mathsf{U}}\mathsf{ref\ int}$

Linear references *(i)*

The let! construct $\texttt{let!}\ (x)\ y = e_1\ \texttt{in}\ e_2$

- Temporarily converts a ${}^{\mathsf{L}}\mathsf{ref}\ \tau$ to a ${}^{\mathsf{U}}\mathsf{ref}\ \tau$
- Example

$$\texttt{let}\ r = \texttt{new}\ 6\ \texttt{in} \qquad r : {}^{\mathsf{L}}\mathsf{ref\ int}$$

$$\texttt{let!}\ (r) \qquad r : {}^{\mathsf{U}}\mathsf{ref\ int}$$

$$y = (\texttt{let}\ a = \texttt{deref}\ r\ \texttt{in}$$

$$r := \texttt{deref}\ r + 1;$$

$$a * \texttt{deref}\ r)$$

Linear references (i)

The let! construct $\texttt{let!}\ (x)\ y = e_1\ \texttt{in}\ e_2$

- Temporarily converts a ${}^{\mathsf{L}}\mathsf{ref}\ \tau$ to a ${}^{\mathsf{U}}\mathsf{ref}\ \tau$
- Example

$\texttt{let}\ r = \texttt{new}\ 6\ \texttt{in}$ $\qquad r : {}^{\mathsf{L}}\mathsf{ref\ int}$

$\texttt{let!}\ (r)$ $\qquad r : {}^{\mathsf{U}}\mathsf{ref\ int}$

$\quad y = (\texttt{let}\ a = \texttt{deref}\ r\ \texttt{in}$

$\qquad\quad r := \texttt{deref}\ r + 1;$

$\qquad\quad a * \texttt{deref}\ r)$

$\texttt{in}$ $\qquad r : {}^{\mathsf{L}}\mathsf{ref\ int}$

Linear references *(i)*

The let! construct $\texttt{let!}\ (x)\ y = e_1\ \texttt{in}\ e_2$

- Temporarily converts a $^{\mathsf{L}}\mathsf{ref}\ \tau$ to a $^{\mathsf{U}}\mathsf{ref}\ \tau$
- Example

```
let r = new 6 in                      r : ^L ref int
let! (r)                              r : ^U ref int
  y = (let a = deref r in
         r := deref r + 1;
         a * deref r)
in                                    r : ^L ref int
  free r;
  print y
```

Linear references (ii)

How do we know let! is only temporary?

- The ${}^{\mathsf{U}}\mathsf{ref}\ \tau$ must not escape the scope of `let!`

Linear references (ii)

How do we know let! is only temporary?

- The $^{\mathsf{U}}\mathsf{ref}\ \tau$ must not escape the scope of `let!`

$\texttt{let}\ r = \texttt{new}\ 6\ \texttt{in}$ $\qquad r : {}^{\mathsf{L}}\mathsf{ref\ int}$

Linear references *(ii)*

How do we know let! is only temporary?

- The $^{\mathsf{U}}\mathsf{ref}\ \tau$ must not escape the scope of `let!`

$\texttt{let}\ r = \texttt{new}\ 6\ \texttt{in}$ $\qquad r : {}^{\mathsf{L}}\mathsf{ref\ int}$

$\texttt{let!}\ (r)$ $\qquad r : {}^{\mathsf{U}}\mathsf{ref\ int}$

Linear references *(ii)*

How do we know let! is only temporary?

- The ${}^{\mathsf{U}}\mathsf{ref}\ \tau$ must not escape the scope of `let!`

$$\texttt{let}\ r = \texttt{new}\ 6\ \texttt{in} \qquad r : {}^{\mathsf{L}}\mathsf{ref\ int}$$

$$\texttt{let!}\ (r) \qquad r : {}^{\mathsf{U}}\mathsf{ref\ int}$$

$$f = \lambda u{:}\mathsf{unit}.\,\texttt{deref}\ r \qquad f : \mathsf{unit} \longrightarrow \mathsf{int}$$

Linear references *(ii)*

How do we know let! is only temporary?

- The ${}^{\mathsf{U}}\mathsf{ref}\ \tau$ must not escape the scope of `let!`

$\texttt{let}\ r = \texttt{new}\ 6\ \texttt{in}$	$r : {}^{\mathsf{L}}\mathsf{ref\ int}$
$\texttt{let!}\ (r)$	$r : {}^{\mathsf{U}}\mathsf{ref\ int}$
$\quad f = \lambda u{:}\mathsf{unit}.\ \texttt{deref}\ r$	$f : \mathsf{unit} \longrightarrow \mathsf{int}$
$\texttt{in}$	$r : {}^{\mathsf{L}}\mathsf{ref\ int}$

Linear references (ii)

How do we know let! is only temporary?

- The $^{\mathsf{U}}\mathsf{ref}\ \tau$ must not escape the scope of `let!`

$\mathtt{let}\ r = \mathtt{new}\ 6\ \mathtt{in}$	$r : {}^{\mathsf{L}}\mathsf{ref\ int}$
$\mathtt{let!}\ (r)$	$r : {}^{\mathsf{U}}\mathsf{ref\ int}$
$\quad f = \lambda u{:}\mathtt{unit}.\ \mathtt{deref}\ r$	$f : \mathsf{unit} \longrightarrow \mathsf{int}$
$\mathtt{in}$	$r : {}^{\mathsf{L}}\mathsf{ref\ int}$
$\quad \mathtt{free}\ r$	

Linear references *(ii)*

How do we know let! is only temporary?

- The ${}^{\mathsf{U}}\mathsf{ref}\ \tau$ must not escape the scope of `let!`

`let` $r =$ `new` 6 `in`	$r : {}^{\mathsf{L}}\mathsf{ref\ int}$
`let!` (r)	$r : {}^{\mathsf{U}}\mathsf{ref\ int}$
$f = \lambda u{:}\mathsf{unit}.\,$`deref` r	$f : \mathsf{unit} \longrightarrow \mathsf{int}$
`in`	$r : {}^{\mathsf{L}}\mathsf{ref\ int}$
`free` $r;\ \ f\ ()$	

Linear references (ii)

How do we know let! is only temporary?

- The ${}^{\mathsf{U}}\mathsf{ref}\ \tau$ must not escape the scope of `let!`

$$
\begin{array}{ll}
\texttt{let}\ r = \texttt{new}\ 6\ \texttt{in} & r : {}^{\mathsf{L}}\mathsf{ref\ int} \\
\texttt{let!}\ (r) & r : {}^{\mathsf{U}}\mathsf{ref\ int} \\
\quad f = \lambda u{:}\texttt{unit}.\ \texttt{deref}\ r & f : \mathsf{unit} \longrightarrow \mathsf{int} \\
\texttt{in} & r : {}^{\mathsf{L}}\mathsf{ref\ int} \\
\quad \texttt{free}\ r;\ \ f\ () &
\end{array}
$$

- The unrestricted r may escape by being
  - returned as (part of) the value computed by `let!`
  - used in function closures
  - assigned to other references
  - ...

Linear references (iii)

How do we know let! is only temporary?

- Several solutions proposed
  - hyperstrict evaluation (Wadler, 1990)
  - observer types (Odersky, 1992)
  - adoption and focus (Fähndrich & DeLine, 2002)
  - ...

Linear references (iii)

How do we know let! is only temporary?

- Several solutions proposed
  - hyperstrict evaluation (Wadler, 1990)
  - observer types (Odersky, 1992)
  - adoption and focus (Fähndrich & DeLine, 2002)
  - ...
  - yet another (wannabe) solution (MP & NP, 2007+)

Linear references (iii)

How do we know let! is only temporary?

- Several solutions proposed
  - hyperstrict evaluation (Wadler, 1990)
  - observer types (Odersky, 1992)
  - adoption and focus (Fähndrich & DeLine, 2002)
  - ...
  - yet another (wannabe) solution (MP & NP, 2007+)
- Our work plan:
  - define a λ-calculus with references, polymorphism, linear types, let!
  - mechanically prove its type safety
  - extend it to fit in the TSCB framework

What is this talk about?

- A family of languages

What is this talk about?

- A family of languages

$\lambda^{\rightarrow}$

What is this talk about?

- A family of languages

$$\lambda^{\rightarrow} \xrightarrow{\text{references}} \lambda^{\text{ref}}$$

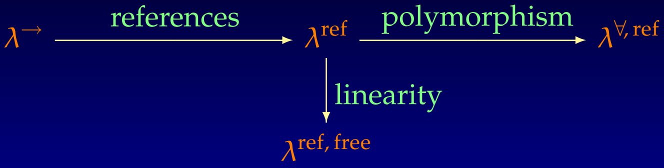
What is this talk about?

- A family of languages

$$\lambda^{\rightarrow} \xrightarrow{\quad\text{references}\quad} \lambda^{\text{ref}} \xrightarrow{\quad\text{polymorphism}\quad} \lambda^{\forall,\,\text{ref}}$$

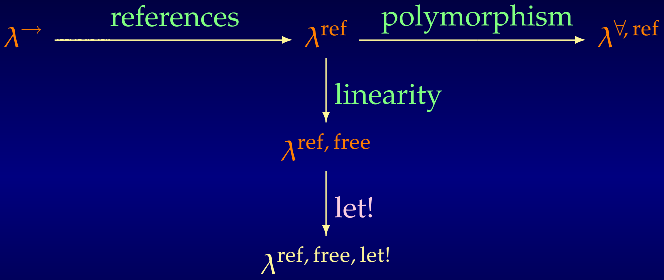
What is this talk about?

- A family of languages



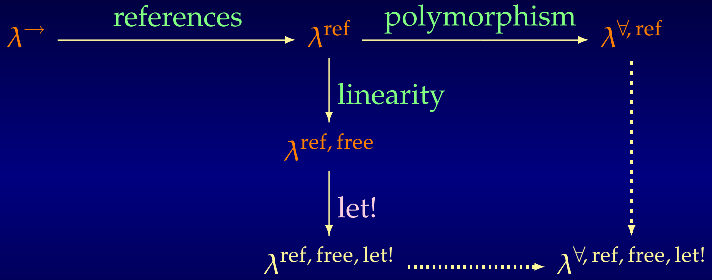
What is this talk about?

- A family of languages



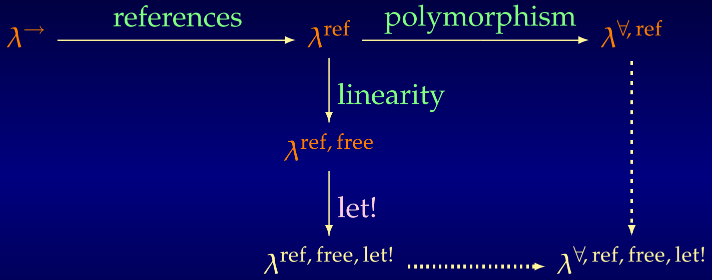
What is this talk about?

- A family of languages



What is this talk about?

- A family of languages



- The goal
  Proof of type safety (progress + preservation)
- The tools
  Isabelle/HOL, ISAR style, locally nameless

The basics: safety proof for $\lambda^{\rightarrow}$ *(i)*

- Abstract syntax

$$\tau \;::=\; b \;\mid\; \tau \rightarrow \tau$$

$$e \;::=\; x \;\mid\; \lambda x{:}\tau.\, e \;\mid\; e_1\, e_2$$

The basics: safety proof for $\lambda^{\rightarrow}$ *(i)*

- Abstract syntax

  $$\tau \;::=\; b \mid \tau \rightarrow \tau$$

  $$e \;::=\; x \mid \lambda x{:}\tau.\, e \mid e_1\, e_2$$

- Environments are sets of pairs $(x \triangleright \tau)$:
  finite and consistent

The basics: safety proof for $\lambda^{\rightarrow}$ *(ii)*

- Typing in locally nameless $\Gamma \vdash \mathtt{e} : \tau$

```
inductive Typing intros
  t_var: ⟦ Γ ⊢ OK; (x▷τ)∈Γ ⟧ ⟹ Γ ⊢ TmFreeVar x : τ
  t_app: ⟦ Γ ⊢ e1 : τ1→τ2; Γ ⊢ e2 : τ1 ⟧ ⟹
            Γ ⊢ e1·e2 : τ2
```

The basics: safety proof for $\lambda^{\rightarrow}$ *(ii)*

- Typing in locally nameless $\Gamma \vdash e : \tau$

```
inductive Typing intros
  t_var: ⟦ Γ ⊢ OK; (x▷τ)∈Γ ⟧ ⟹ Γ ⊢ TmFreeVar x : τ
  t_app: ⟦ Γ ⊢ e1 : τ1→τ2; Γ ⊢ e2 : τ1 ⟧ ⟹
             Γ ⊢ e1·e2 : τ2
  t_abs: ⟦ Γ ⊢ OK;
           ∀x. ¬ x free in e ∧ ¬ Γ defines x ⟶
                 Γ,(x:τ1) ⊢ freshen_tm x e : τ2 ⟧ ⟹
             Γ ⊢ λ[τ1]. e : τ1→τ2
```

The basics: safety proof for $\lambda^{\rightarrow}$ (ii)

- Typing in locally nameless $\Gamma \vdash e : \tau$

```
inductive Typing intros
  t_var: ⟦ Γ ⊢ OK; (x▷τ)∈Γ ⟧ ⟹ Γ ⊢ TmFreeVar x : τ
  t_app: ⟦ Γ ⊢ e1 : τ1→τ2; Γ ⊢ e2 : τ1 ⟧ ⟹
           Γ ⊢ e1·e2 : τ2
  t_abs: ⟦ Γ ⊢ OK; finite L;
           ∀x. ¬ x free in e ∧ ¬ Γ defines x ∧ x∉L ⟶
               Γ,(x:τ1) ⊢ freshen_tm x e : τ2 ⟧ ⟹
           Γ ⊢ λ[τ1]. e : τ1→τ2
```

Adding references: λ^{ref} *(i)*

- Abstract syntax

$$\tau \ ::= \ b \mid \tau \to \tau \mid \mathsf{ref}\ \tau$$

$$\begin{aligned} e \ ::= \ & c \mid x \mid \lambda x{:}\tau.\,e \mid e_1\,e_2 \\ \mid \ & \texttt{new}\ e \mid e_1 := e_2 \mid \texttt{deref}\ e \mid \texttt{loc}\ \ell \end{aligned}$$

Adding references: λ^{ref} *(i)*

- Abstract syntax

$$\tau ::= b \mid \tau \to \tau \mid \text{ref}\ \tau$$

$$\begin{aligned} e ::= {} & c \mid x \mid \lambda x{:}\tau.\, e \mid e_1\, e_2 \\ \mid {} & \texttt{new}\ e \mid e_1 := e_2 \mid \texttt{deref}\ e \mid \texttt{loc}\ \ell \end{aligned}$$

- Values

$$v ::= c \mid \lambda x{:}\tau.\, e \mid \texttt{loc}\ \ell$$

Adding references: λ^{ref} *(i)*

- Abstract syntax

$$\tau \ ::= \ b \mid \tau \to \tau \mid \text{ref}\ \tau$$

$$e \ ::= \ c \mid x \mid \lambda x{:}\tau.\,e \mid e_1\, e_2$$
$$\mid \ \texttt{new}\ e \mid e_1 := e_2 \mid \texttt{deref}\ e \mid \texttt{loc}\ \ell$$

- Values

$$v \ ::= \ c \mid \lambda x{:}\tau.\,e \mid \texttt{loc}\ \ell$$

- Stores are sets of pairs $(\ell \mapsto v)$

Adding references: λ^{ref} *(i)*

- Abstract syntax

  $\tau \ ::= \ b \mid \tau \to \tau \mid \mathsf{ref}\ \tau$

  $e \ ::= \ c \mid x \mid \lambda x{:}\tau.\, e \mid e_1\, e_2$

  $\quad \mid \ \mathtt{new}\ e \mid e_1 := e_2 \mid \mathtt{deref}\ e \mid \mathtt{loc}\ \ell$

- Values

  $v \ ::= \ c \mid \lambda x{:}\tau.\, e \mid \mathtt{loc}\ \ell$

- Stores are sets of pairs $(\ell \mapsto v)$
- It simplifies things to take $\ell \equiv x$

Adding references: λ^{ref} *(i)*

- Abstract syntax

$$\tau ::= b \mid \tau \rightarrow \tau \mid \text{ref}\ \tau$$

$$e ::= c \mid x \mid \lambda x{:}\tau.\, e \mid e_1\, e_2$$
$$\mid \texttt{new}\ e \mid e_1 := e_2 \mid \texttt{deref}\ e \mid \texttt{loc}\ \ell$$

- Values

$$v ::= c \mid \lambda x{:}\tau.\, e \mid \texttt{loc}\ \ell$$

- Stores are sets of pairs $(\ell \mapsto v)$
- It simplifies things to take $\ell \equiv x$
- Typing still uses one environment $\Gamma \vdash e : \tau$

Adding references: λ^{ref} *(ii)*

- It further simplifies things to use variables as the real values

Adding references: λ^{ref} *(ii)*

- It further simplifies things to use variables as the real values
- Semantics with "temporaries" S;e$\hookrightarrow$S';e'

```
inductive Eval intros
  e_val: ⟦ ¬ S defines z; S ⊨ Store; value v ⟧ ⟹
           S;v ↪ S,(z↦v);TmFreeVar z
```

Adding references: λ^{ref} *(ii)*

- It further simplifies things to use variables as the real values
- Semantics with "temporaries" $\quad$ `S;e⟶S';e'`

```
inductive Eval intros
  e_val: ⟦ ¬ S defines z; S ⊨ Store; value v ⟧ ⟹
            S;v ↪ S,(z↦v);TmFreeVar z
  e_beta: ⟦ S;z⇓λ[τ].e1 ⟧ ⟹
       S;(TmFreeVar z)·(TmFreeVar y) ↪ S;freshen_tm y e1
```

Adding references: λ^{ref} (ii)

- It further simplifies things to use variables as the real values
- Semantics with "temporaries" $\quad$ `S;e↪S';e'`

```
inductive Eval intros
  e_val: ⟦ ¬ S defines z; S ⊨ Store; value v ⟧ ⟹
           S;v ↪ S,(z↦v);TmFreeVar z
  e_beta: ⟦ S;z⇓λ[τ].e1 ⟧ ⟹
       S;(TmFreeVar z)·(TmFreeVar y) ↪ S;freshen_tm y e1
```

- Store typing $\quad \models S : \Gamma$

Adding references: λ^{ref} (ii)

- It further simplifies things to use variables as the real values
- Semantics with "temporaries" S;e↪S';e'

```
inductive Eval intros
  e_val: ⟦ ¬ S defines z; S ⊨ Store; value v ⟧ ⟹
            S;v ↪ S,(z↦v);TmFreeVar z
  e_beta: ⟦ S;z⇓λ[τ].e1 ⟧ ⟹
      S;(TmFreeVar z)·(TmFreeVar y) ↪ S;freshen_tm y e1
```

- Store typing ⊨ S : Γ
- In preservation, S and Γ expand
  - temporaries are added — computed values
  - locations are added — allocated objects

Adding polymorphism: $\lambda^{\forall,\,\mathrm{ref}}$ *(i)*

- Abstract syntax

$$\tau \;::=\; b \mid \tau \to \tau \mid \mathsf{ref}\,\tau \mid \alpha \mid \forall\alpha.\,\tau$$

$$e \;::=\; c \mid x \mid \lambda x{:}\tau.\,e \mid e_1\,e_2 \mid \Lambda\alpha.\,e \mid e\,[\tau]$$
$$\mid\; \mathtt{new}\;e \mid e_1 := e_2 \mid \mathtt{deref}\;e \mid \mathtt{loc}\;\ell$$

$$v \;::=\; c \mid \lambda x{:}\tau.\,e \mid \Lambda\alpha.\,v \mid \mathtt{loc}\;\ell$$

Adding polymorphism: $\lambda^{\forall,\,\mathrm{ref}}$ *(i)*

- Abstract syntax

$$\tau \;::=\; b \mid \tau \to \tau \mid \mathsf{ref}\,\tau \mid \alpha \mid \forall\alpha.\,\tau$$

$$\begin{aligned} e \;::=\;& c \mid x \mid \lambda x{:}\tau.\,e \mid e_1\,e_2 \mid \Lambda\alpha.\,e \mid e\,[\tau] \\ \mid\;& \mathtt{new}\,e \mid e_1 := e_2 \mid \mathtt{deref}\,e \mid \mathtt{loc}\,\ell \end{aligned}$$

$$v \;::=\; c \mid \lambda x{:}\tau.\,e \mid \Lambda\alpha.\,v \mid \mathtt{loc}\,\ell$$

- Substitution of types and terms
- Two substitution lemmata

Adding polymorphism: $\lambda^{\forall,\mathrm{ref}}$ *(ii)*

- In the type substitution lemma, at some point in the case `e = Λ[*]. e_b`, we must show

$$\tau_1\{i+1 \mapsto x\}\{0 \mapsto \tau_2\{i \mapsto x\}\} = \tau\{i \mapsto x\}$$

$$\Longrightarrow \quad \tau_1\{i+1 \mapsto \tau'\}\{0 \mapsto \tau_2\{i \mapsto \tau'\}\} = \tau\{i \mapsto \tau'\}$$

  (provided all mentioned types are closed and x is not free in τ, τ_1, τ_2)

Adding polymorphism: $\lambda^{\forall,\mathrm{ref}}$ (ii)

- In the type substitution lemma, at some point in the case `e = Λ[*]. e_b`, we must show

$$\tau_1\{i+1 \mapsto x\}\{0 \mapsto \tau_2\{i \mapsto x\}\} = \tau\{i \mapsto x\}$$

$$\Longrightarrow \quad \tau_1\{i+1 \mapsto \tau'\}\{0 \mapsto \tau_2\{i \mapsto \tau'\}\} = \tau\{i \mapsto \tau'\}$$

  (provided all mentioned types are closed and x is not free in τ, τ_1, τ_2)
- Easier to generalize: substitution functions
  - meta-level functions representing contexts
  - mapping closed terms to terms

Adding polymorphism: $\lambda^{\forall,\mathrm{ref}}$ *(ii)*

- In the type substitution lemma, at some point in the case `e = Λ[*]. e_b`, we must show

$$f(x) = \tau\{i \mapsto x\}$$
$$\Longrightarrow \quad f(\tau') = \tau\{i \mapsto \tau'\}$$

  (provided all mentioned types are closed and x is not free in τ, τ_1, τ_2)
- Easier to generalize: substitution functions
  - meta-level functions representing contexts
  - mapping closed terms to terms

Adding polymorphism: $\lambda^{\forall,\,\mathrm{ref}}$ *(ii)*

- In the type substitution lemma, at some point in the case `e = Λ[*]. e_b`, we must show

  $$f(x) = g(x)$$
  $$\implies f(\tau') = g(\tau')$$

  (provided all mentioned types are closed and x is not free in τ, τ_1, τ_2)
- Easier to generalize: substitution functions
  - meta-level functions representing contexts
  - mapping closed terms to terms

Adding linearity: $\lambda^{\text{ref, free}}$ *(i)*

- Abstract syntax

$$
\begin{array}{llll}
q & ::= & \mathsf{L} \mid \mathsf{U} & \text{qualifiers} \\
\varphi & ::= & b \mid \tau \rightarrow \tau \mid \mathsf{ref}\ \tau & \text{pretypes} \\
\tau & ::= & {}^{q}\varphi & \text{types} \\
e & ::= & {}^{q}c \mid x \mid {}^{q}\lambda x\!:\!\tau.\, e \mid e_1\, e_2 & \\
 & \mid & \mathtt{new}\ e \mid e_1 := e_2 \mid \mathtt{deref}\ e \mid {}^{q}\mathtt{loc}\ \ell & \\
 & \mid & \mathtt{free}\ e \mid e_1 :=: e_2 &
\end{array}
$$

Adding linearity: $\lambda^{\text{ref, free}}$ *(i)*

- Abstract syntax

$$
\begin{array}{lcll}
q & ::= & \mathsf{L} \mid \mathsf{U} & \text{qualifiers} \\
\varphi & ::= & b \mid \tau \to \tau \mid \mathsf{ref}\,\tau & \text{pretypes} \\
\tau & ::= & {}^{q}\varphi & \text{types} \\
e & ::= & {}^{q}c \mid x \mid {}^{q}\lambda x\!:\!\tau.\,e \mid e_1\,e_2 & \\
 & \mid & \mathtt{new}\,e \mid e_1 := e_2 \mid \mathtt{deref}\,e \mid {}^{q}\mathtt{loc}\,\ell & \\
 & \mid & \mathtt{free}\,e \mid e_1 :=: e_2 &
\end{array}
$$

- Two additional constructs
  - Explicit deallocation
  - Swapping: assignment without losing the previous contents

Adding linearity: $\lambda^{\text{ref, free}}$ *(ii)*

- Easier to separate temporaries from locations

Adding linearity: $\lambda^{\text{ref, free}}$ *(ii)*

- Easier to separate temporaries from locations
- Stores are sets of pairs $(x \mapsto v)$ — temporaries

Adding linearity: $\lambda^{\text{ref, free}}$ *(ii)*

- Easier to separate temporaries from locations
- Stores are sets of pairs $(x \mapsto v)$ — temporaries
- Memories are sets of pairs $(\ell \mapsto x)$ — locations

Adding linearity: $\lambda^{\text{ref, free}}$ *(ii)*

- Easier to separate temporaries from locations
- Stores are sets of pairs $(x \mapsto v)$ — temporaries
- Memories are sets of pairs $(\ell \mapsto x)$ — locations
- Looking up the store: linear values are removed

Adding linearity: $\lambda^{\text{ref, free}}$ *(ii)*

- Easier to separate temporaries from locations
- Stores are sets of pairs $(x \mapsto v)$ — temporaries
- Memories are sets of pairs $(\ell \mapsto x)$ — locations
- Looking up the store: linear values are removed
- Compatible type environments $\Gamma 1 \sim \Gamma 2$

Adding linearity: $\lambda^{\text{ref, free}}$ *(ii)*

- Easier to separate temporaries from locations
- Stores are sets of pairs $(x \mapsto v)$ — temporaries
- Memories are sets of pairs $(\ell \mapsto x)$ — locations
- Looking up the store: linear values are removed
- Compatible type environments $\Gamma 1 \sim \Gamma 2$
- Typing uses two environments $\Gamma ; M \vdash e : \tau$

```
inductive Typing intros
  t_app: ⟦ Γ1 ∼ Γ2; Γ1∪Γ2 ⊨ OK;
           Γ1;M ⊢ e1 : Qual q (τ1→τ2);
           Γ2;M ⊢ e2 : τ1 ⟧ ⟹
             Γ1∪Γ2;M ⊢ e1·e2 : τ2
```

Adding linearity: $\lambda^{\text{ref, free}}$ *(iii)*

- Substitution lemma: can only substitute free variables for DeBruijn indices

Adding linearity: $\lambda^{\text{ref, free}}$ *(iii)*

- Substitution lemma: can only substitute free variables for DeBruijn indices
- Store typing and memory typing are inductively defined

Adding linearity: $\lambda^{\text{ref, free}}$ *(iii)*

- Substitution lemma: can only substitute free variables for DeBruijn indices
- Store typing and memory typing are inductively defined
- Two invariants on stores
  - locations are only top-level, i.e. $(x \mapsto {}^{q}\texttt{loc}\ \ell)$
  - linear locations appear only once

Adding linearity: $\lambda^{\text{ref, free}}$ *(iii)*

- Substitution lemma: can only substitute free variables for DeBruijn indices
- Store typing and memory typing are inductively defined
- Two invariants on stores
  - locations are only top-level, i.e. $(x \mapsto {}^{q}\texttt{loc}\ \ell)$
  - linear locations appear only once
- Hack: the semantics of `new` places the new location in the store and returns a temporary

Adding linearity: $\lambda^{\text{ref, free}}$ *(iv)*

- Preservation

$$\left.\begin{array}{l} S; \mu; e \hookrightarrow S'; \mu'; e' \\ \Gamma_e; \emptyset \vdash e : \tau \end{array}\right\} \Longrightarrow \begin{array}{l} \exists \Gamma'_e. \\ \quad \Gamma'_e; \emptyset \vdash e' : \tau \end{array}$$

Adding linearity: $\lambda^{\text{ref, free}}$ *(iv)*

- Preservation

$$\left.\begin{array}{l} S;\mu;e \hookrightarrow S';\mu';e' \\ \Gamma_e;\emptyset \vdash e : \tau \\ M \models S : \Gamma_s \cup \Gamma_m \\ \Gamma_m \models \mu : M \\ \Gamma_s \sim \Gamma_m \\ \text{invariants}(S) \\ \Gamma_e \subseteq \Gamma_s \end{array}\right\} \Longrightarrow \begin{array}{l} \exists \Gamma_e', \Gamma_s', \Gamma_m', M'. \\ \quad \Gamma_e';\emptyset \vdash e' : \tau \\ \quad M' \models S' : \Gamma_s' \cup \Gamma_m' \\ \quad \Gamma_m' \models \mu' : M' \\ \quad \Gamma_s' \sim \Gamma_m' \\ \quad \text{invariants}(S') \\ \quad \Gamma_e' \subseteq \Gamma_s' \end{array}$$

Adding linearity: $\lambda^{\text{ref, free}}$ *(iv)*

- Preservation

$$\left.\begin{array}{l} S;\mu;e \hookrightarrow S';\mu';e' \\ \Gamma_e;\emptyset \vdash e : \tau \\ M \models S : \Gamma_s \cup \Gamma_m \\ \Gamma_m \models \mu : M \\ \Gamma_s \sim \Gamma_m \\ \text{invariants}(S) \\ \Gamma_e \cup \Gamma_r \subseteq \Gamma_s \\ \Gamma_e \sim \Gamma_r \end{array}\right\} \Longrightarrow \begin{array}{l} \exists \Gamma'_e, \Gamma'_s, \Gamma'_m, M'. \\ \quad \Gamma'_e;\emptyset \vdash e' : \tau \\ \quad M' \models S' : \Gamma'_s \cup \Gamma'_m \\ \quad \Gamma'_m \models \mu' : M' \\ \quad \Gamma'_s \sim \Gamma'_m \\ \quad \text{invariants}(S') \\ \quad \Gamma'_e \cup \Gamma_r \subseteq \Gamma'_s \\ \quad \Gamma'_e \sim \Gamma_r \end{array}$$

Adding linearity: $\lambda^{\text{ref, free}}$ (iv)

- Preservation

$$\left.\begin{array}{l} S;\mu;e \hookrightarrow S';\mu';e' \\ \Gamma_e;\emptyset \vdash e : \tau \\ M \models S : \Gamma_s \cup \Gamma_m \\ \Gamma_m \models \mu : M \\ \Gamma_s \sim \Gamma_m \\ \text{invariants}(S) \\ \Gamma_e \cup \Gamma_r \subseteq \Gamma_s \\ \Gamma_e \sim \Gamma_r \end{array}\right\} \Longrightarrow \begin{array}{l} \exists \Gamma'_e, \Gamma'_s, \Gamma'_m, M'. \\ \quad \Gamma'_e;\emptyset \vdash e' : \tau \\ \quad M' \models S' : \Gamma'_s \cup \Gamma'_m \\ \quad \Gamma'_m \models \mu' : M' \\ \quad \Gamma'_s \sim \Gamma'_m \\ \quad \text{invariants}(S') \\ \quad \Gamma'_e \cup \Gamma_r \subseteq \Gamma'_s \\ \quad \Gamma'_e \sim \Gamma_r \end{array}$$

- Γ_r contains temporaries that have been used elsewhere in the evaluation

Adding linearity: $\lambda^{\text{ref, free}}$ *(v)*

- Progress

$$\Gamma_e; \emptyset \vdash e : \tau \implies \text{not\_stuck}(e, S, \mu)$$

Adding linearity: $\lambda^{\text{ref, free}}$ *(v)*

- Progress

$$\left.\begin{array}{l} \Gamma_e; \emptyset \vdash e : \tau \\ \Gamma_e \subseteq \Gamma_s \\ M \models S : \Gamma_s \cup \Gamma_m \\ \Gamma_m \models \mu : M \\ \text{invariants}(S) \end{array}\right\} \implies \text{not\_stuck}(e, S, \mu)$$

Adding linearity: $\lambda^{\text{ref, free}}$ *(vi)*

File	$\lambda^{\rightarrow}$	λ^{ref}	$\lambda^{\forall,\text{ref}}$	$\lambda^{\text{ref, free}}$
`Environ.thy`	46	46	46	46
`Syntax.thy`	94	116	699	139
`Typing.thy`	74	83	738	366
`Semantics.thy`	47	143	138	231
`Metatheory.thy`	153	553	1151	2865
Total	414	941	2772	3647

Adding linearity: $\lambda^{\text{ref, free}}$ *(vi)*

File	$\lambda^{\text{ref, free}}$
`Environ.thy`	46
`Syntax.thy`	139
`Typing.thy`	366
`Semantics.thy`	231
`Metatheory.thy`	2865
Total	3647

`Metatheory.thy` $\longrightarrow$

weakening	5
substitution	489
store typing	565
invariants	75
preservation	1360
progress	326
safety	32

Conclusions

- Related work: fully fledged languages with polymorphism and references
  - ML (Dubois, 2000; Lee, Crary & Harper, 2007)
  - Java (von Oheimb, 2001; Leavens, Naumann & Rosenberg, 2006)

Conclusions

- Related work: fully fledged languages with polymorphism and references
  - ML (Dubois, 2000; Lee, Crary & Harper, 2007)
  - Java (von Oheimb, 2001; Leavens, Naumann & Rosenberg, 2006)
  - references and impredicative polymorphism?

Conclusions

- Related work: fully fledged languages with polymorphism and references
  - ML (Dubois, 2000; Lee, Crary & Harper, 2007)
  - Java (von Oheimb, 2001; Leavens, Naumann & Rosenberg, 2006)
  - references and impredicative polymorphism?
- Related work: references and linear type systems
  - Walker & Watkins, 2001
  - Fluet, Morrisett & Ahmed, 2005, 2006
  - ...

Conclusions

- Related work: fully fledged languages with polymorphism and references
  - ML (Dubois, 2000; Lee, Crary & Harper, 2007)
  - Java (von Oheimb, 2001; Leavens, Naumann & Rosenberg, 2006)
  - references and impredicative polymorphism?
- Related work: references and linear type systems
  - Walker & Watkins, 2001
  - Fluet, Morrisett & Ahmed, 2005, 2006
  - ...
- Contribution
  - mechanized proofs of type safety for $\lambda^{\forall,\,\mathrm{ref}}$ and $\lambda^{\mathrm{ref},\,\mathrm{free}}$ in Isabelle/HOL

Thank you...
Questions?

by auto

```
*** Terminal proof method failed
*** At command "by".
```

sorry