Space-Efficient Gradual Typing

David Herman
Northeastern University
Aaron Tomb, Cormac Flanagan
University of California, Santa Cruz

The point

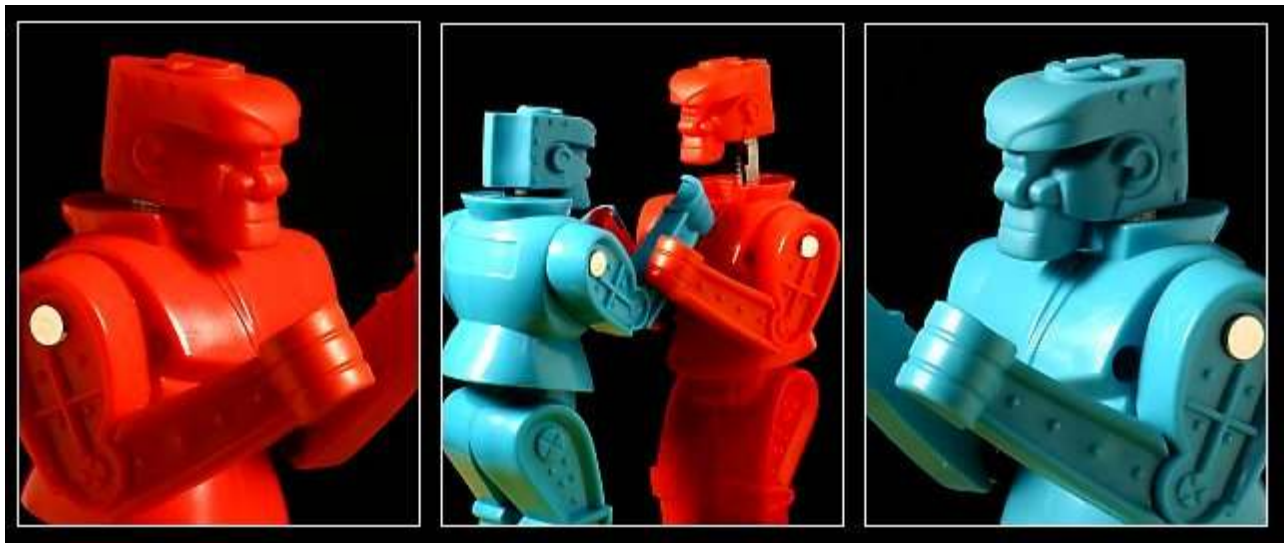
Naïve type conversions in functional programming languages are not safe for space.

But they can and should be.

Gradual Typing:

Software evolution via hybrid type checking

Dynamic vs. static typing



Dynamic Typing

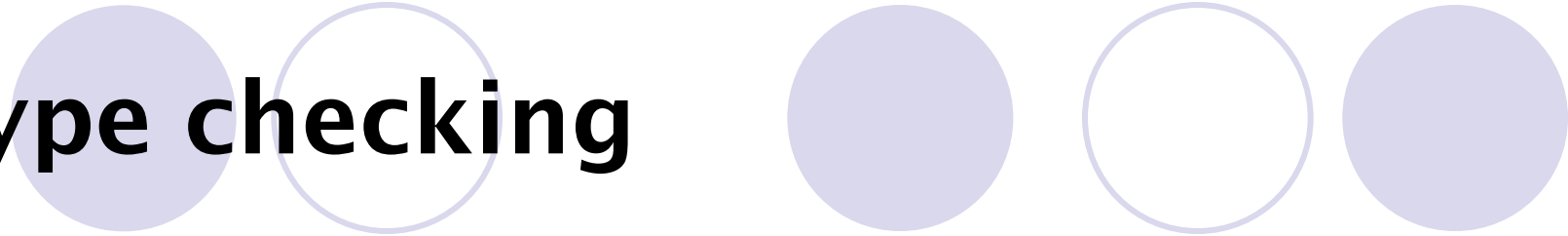
Static Typing

Gradual typing

Dynamic Typing

Static Typing

Type checking



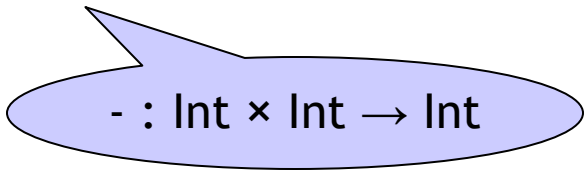
```
let x = f() in
    …
    let y : Int = x - 3 in
        …
```

Type checking

```
let x : ? = f() in
    …
    let y : Int = x - 3 in
        …
```

Type checking

```
let x : ? = f() in
    …
    let y : Int = x - 3 in
        …
```

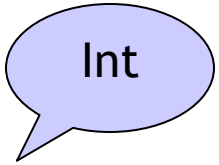


Type checking

```
let x : ? = f() in
    …
    let y : Int = <Int>x - 3 in
        …
```

Type checking

```
let x : ? = f() in
    …
    let y : Int = <Int>x - 3 in
        …
```



Evaluation

```
let x : ? = f() in
    …
    let y : Int = <Int>x - 3 in
        …
```

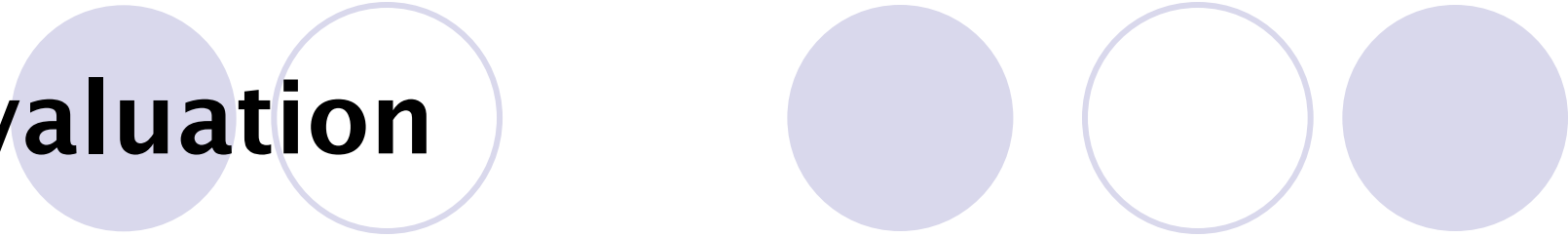
Evaluation

```
let x : ? = 45 in
    …
    let y : Int = <Int>x - 3 in
        …
```

Evaluation

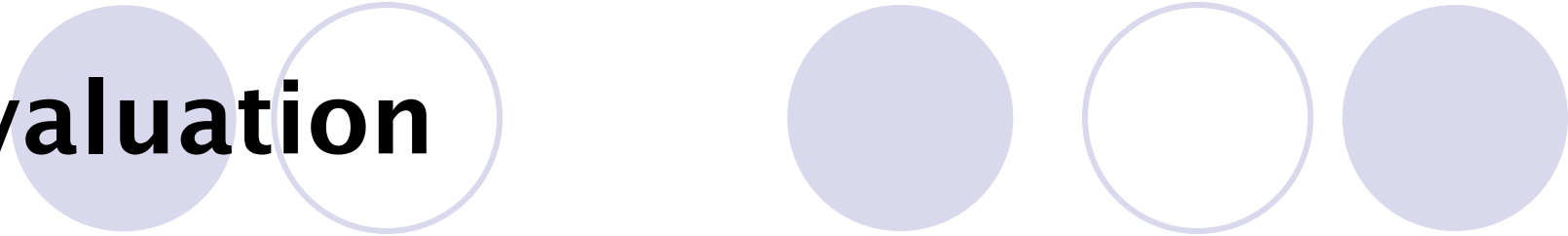
```
let y : Int = <Int>45 - 3 in
    …
```

Evaluation



```
let y : Int = 45 - 3 in
    …
```

Evaluation



```
let y : Int = 42 in
    …
```

Evaluation (take 2)

```
let x : ? = f() in
    …
    let y : Int = <Int>x - 3 in
        …
```

Evaluation (take 2)

```
let x : ? = true in
    …
    let y : Int = <Int>x - 3 in
        …
```

Evaluation (take 2)

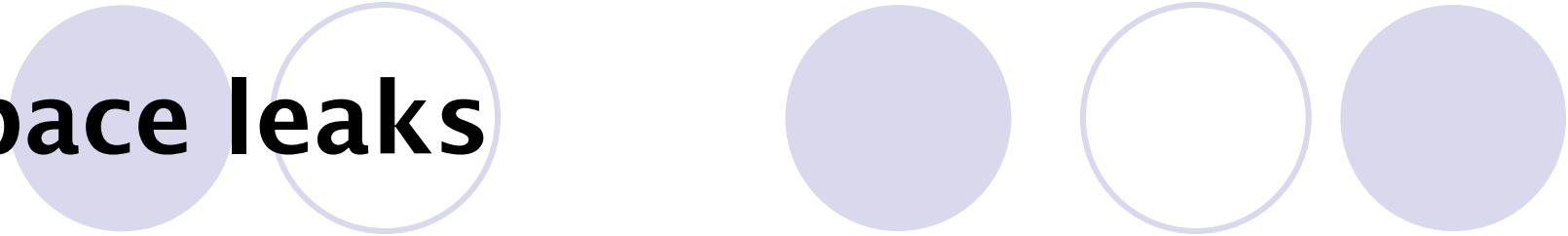
```
let y : Int = <Int>true - 3 in
    …
```

Evaluation (take 2)

`error: “true is not an Int”`

Space Leaks

Space leaks



```
fun even(n) =
    if (n = 0) then true
               else odd(n - 1)

fun odd(n) =
    if (n = 0) then false
               else even(n - 1)
```

Space leaks

```
fun even(n : Int) =
    if (n = 0) then true
               else odd(n - 1)

fun odd(n : Int) : Bool =
    if (n = 0) then false
               else even(n - 1)
```

Space leaks

```
fun even(n : Int) =
    if (n = 0) then true
               else odd(n - 1)

fun odd(n : Int) : Bool =
    if (n = 0) then false
               else <Bool>even(n - 1)
```

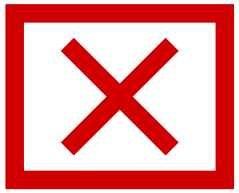
Space leaks

```
fun even(n : Int) =
    if (n = 0) then true
               else odd(n - 1)

fun odd(n : Int) : Bool =
    if (n = 0) then false
               else <Bool>even(n - 1)
```

Space leaks

```
even(n)
→* odd(n - 1)
→* <Bool>even(n - 2)
→* <Bool>odd(n - 3)
→* <Bool><Bool>even(n - 4)
→* <Bool><Bool>odd(n - 5)
→* <Bool><Bool><Bool>even(n - 6)
→* …
```



Naïve Function Casts

Casts in functional languages

$\langle \mathtt{Int} \rangle n \rightarrow n$

$\langle \mathtt{Int} \rangle v \rightarrow \mathtt{error:}$ “failed cast” (if $v \notin \mathtt{Int}$)

$\langle \sigma \rightarrow \tau \rangle \lambda x\mathtt{:?.}e \rightarrow \ldots$

Casts in functional languages

$\langle \mathtt{Int} \rangle n \rightarrow n$

$\langle \mathtt{Int} \rangle v \rightarrow$ error: "failed cast" (if $v \notin \mathtt{Int}$)

$\langle \sigma \rightarrow \tau \rangle \lambda x{:}?.e \rightarrow \lambda z{:}\sigma.\langle \tau \rangle((\lambda x{:}?.e)\ z)$

Casts in functional languages

$\langle \mathtt{Int} \rangle n \rightarrow n$

$\langle \mathtt{Int} \rangle v \rightarrow$ error: "failed cast" (if $v \notin \mathtt{Int}$)

$\langle \sigma \rightarrow \tau \rangle \lambda x{:}?.e \rightarrow \lambda z{:}\sigma.\langle \tau \rangle((\lambda x{:}?.e)\ z)$

fresh, typed proxy

cast result

Casts in functional languages

$\langle \mathtt{Int} \rangle n \rightarrow n$

$\langle \mathtt{Int} \rangle v \rightarrow \mathtt{error:}$ "failed cast" (if $v \notin \mathtt{Int}$)

$\langle \sigma \rightarrow \tau \rangle \lambda x{:}?.e \rightarrow \lambda z{:}\sigma.\langle \tau \rangle((\lambda x{:}?.e)\ z)$

Very useful, very popular… *unsafe for space*.

More space leaks

```
fun evenk(n : Int, k : ? → ?) =
    if (n = 0)
      then k(true)
      else oddk(n - 1, k)

fun oddk(n : Int, k : Bool → Bool) =
    if (n = 0)
      then k(false)
      else evenk(n - 1, k)
```

More space leaks

```
fun evenk(n : Int, k : ? → ?) =
    if (n = 0)
      then k(true)
      else oddk(n - 1, <Bool→Bool>k)

fun oddk(n : Int, k : Bool → Bool) =
    if (n = 0)
      then k(false)
      else evenk(n - 1, <?→?>k)
```

More space leaks

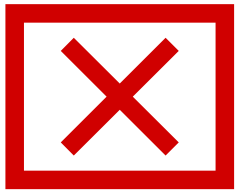
evenk(n, k_0)

$\rightarrow^*$ oddk(n - 1, <Bool→Bool>k_0)

$\rightarrow^*$ oddk(n - 1, λz:Bool.<Bool>k_0(z))

$\rightarrow^*$ evenk(n - 2, <?→?>λz:Bool.<Bool>k_0(z))

$\rightarrow^*$ evenk(n - 2, λy:?.(λz:Bool.<Bool>k_0(z))(y))



$\rightarrow^*$ oddk(n - 3, <Bool→Bool>λy:?.(λz:Bool.<Bool>k_0(z))(y))

$\rightarrow^*$ oddk(n - 3, λx:Bool.(λy:?.(λz:Bool.<Bool>k_0(z))(y))(x))

$\rightarrow^*$ evenk(n - 4, <?→?>λx:Bool.(λy:?.(λz:Bool.<Bool>k_0(z))(y))(x))

$\rightarrow^*$ evenk(n - 4, λw:?.(λx:Bool.(λy:?.(λz:Bool.<Bool>k_0(z))(y))(x))(w))

$\rightarrow^*$ oddk(n - 5, <Bool→Bool>λw:?.(λx:Bool.(λy:?.(λz:Bool.<Bool>k_0(z))(y))(x))(w))

$\rightarrow^*$ oddk(n - 5, λv:Bool.<Bool>(λw:?.(λx:Bool.(λy:?.(λz:Bool.<Bool>k_0(z))(y))(x))(w))(v))

$\rightarrow^*$ …

(…without even *using* k_0!)

Space-Efficient Gradual Typing

Intuition

Casts are like *function restrictions* (Findler and Blume, 2006)

Can their representation exploit the properties of restrictions?

Exploiting algebraic properties

Closure under composition:

$$\langle \mathtt{Bool} \rangle(\langle \mathtt{Bool} \rangle\ v) = (\langle \mathtt{Bool} \rangle \circ \langle \mathtt{Bool} \rangle)\ v$$

Exploiting algebraic properties

Idempotence:

$$\begin{aligned} &\langle Bool\rangle(\langle Bool\rangle\ v) \\ &\quad = (\langle Bool\rangle \circ \langle Bool\rangle)\ v \\ &\quad = \langle Bool\rangle\ v \end{aligned}$$

Exploiting algebraic properties

Distributivity:

$$(\langle ? \to ? \rangle \circ \langle Bool \to Bool \rangle)\ v$$
$$= \langle (Bool \circ ?) \to (? \circ Bool) \rangle\ v$$

Space-efficient gradual typing

- Generalize casts to *coercions* (Henglein, 1994)
- Change representation of casts from <τ> to <c>
- Merge casts at runtime:

This coercion can be simplified!

$$\langle c \rangle(\langle d \rangle\ e) \quad \rightarrow \quad \langle c \circ d \rangle e$$

merged *before* evaluating e

Space-efficient gradual typing

- Generalize casts to *coercions* (Henglein, 1994)
- Change representation of casts from <τ> to <c>
- Merge casts at runtime:

$$\langle c \rangle(\langle d \rangle\ e) \rightarrow \langle c \circ d \rangle e$$
$$\rightarrow \langle c' \rangle e$$

Tail recursion

```
even(n)
→* odd(n - 1)
→* <Bool>even(n - 2)
→* <Bool>odd(n - 3)
→* <Bool><Bool>even(n - 4)
→* <Bool>even(n - 4)
→* <Bool>odd(n - 5)
→* <Bool><Bool>even(n - 6)
→* <Bool>even(n - 6)
→* …
```

Bounded proxies

evenk(n, k_0)
$\rightarrow^*$ oddk(n - 1, <Bool→Bool>k_0)
$\rightarrow^*$ evenk(n - 2, <?→?><Bool→Bool>k_0)
$\rightarrow^*$ evenk(n - 2, <Bool→Bool>k_0)
$\rightarrow^*$ oddk(n - 3, <Bool→Bool>k_0)
$\rightarrow^*$ evenk(n - 4, <?→?><Bool→Bool>k_0)
$\rightarrow^*$ evenk(n - 4, <Bool→Bool>k_0)
$\rightarrow^*$ oddk(n - 5, <Bool→Bool>k_0)
$\rightarrow^*$ …

Guaranteed.

Theorem: any program state S during evaluation of a program P is bounded by

$$k_P \cdot size_{OR}(S)$$

$size_{OR}(S)$ = size of S *without any casts*

Earlier error detection

```
<Int→Int>(<Bool→Bool> e)
```

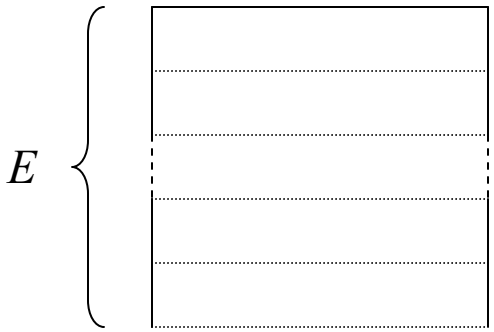
→ `<Fail→Fail> e`

→ `error:` “incompatible casts”

Implementation

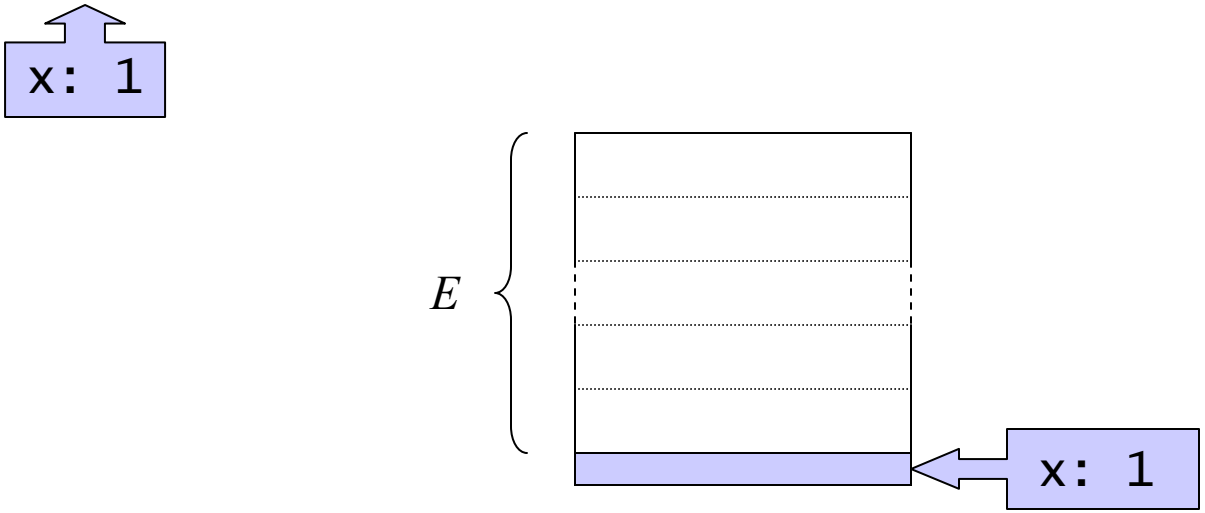
Continuation marks

E[mark x = 1 in e end]

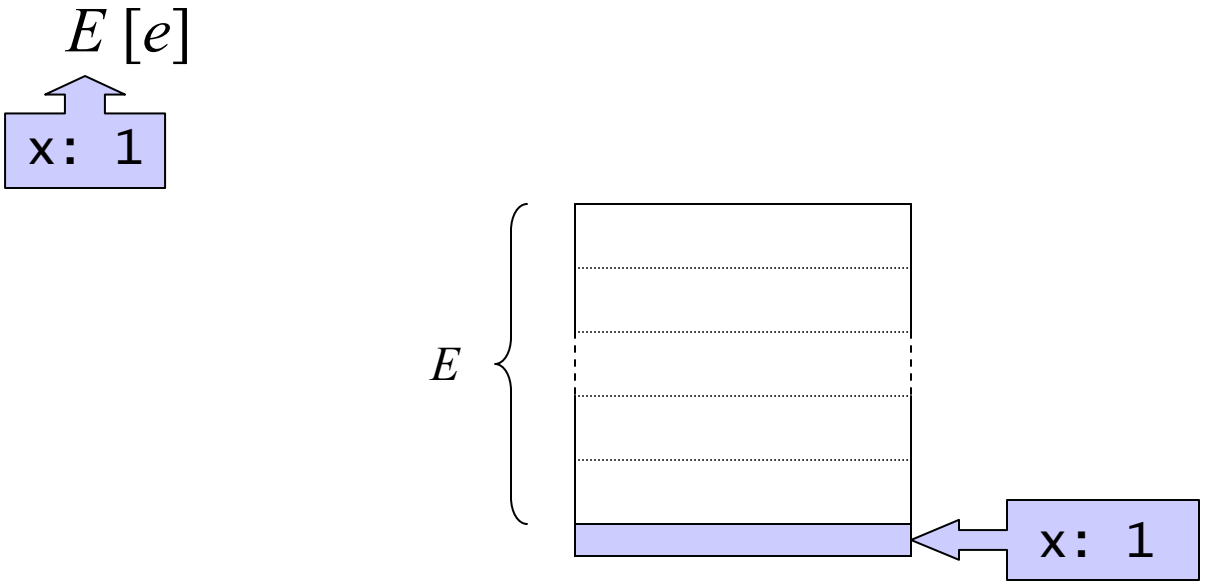


Continuation marks

E [`mark x = 1 in` e `end`]

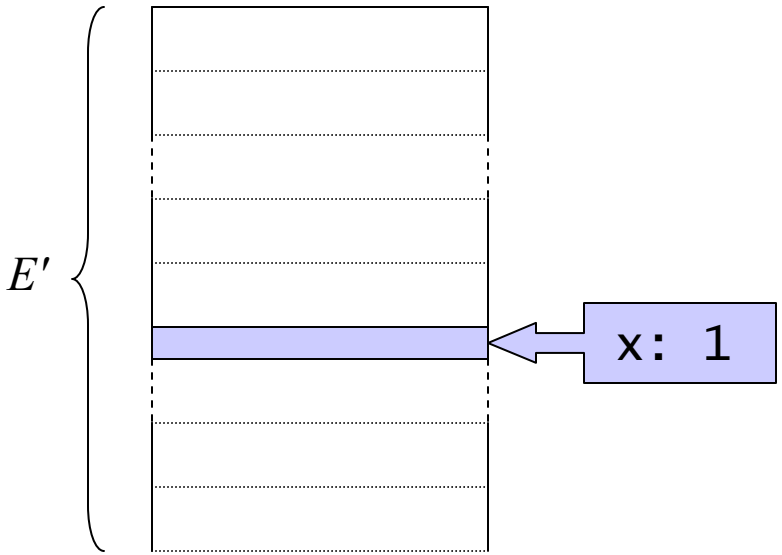


Continuation marks



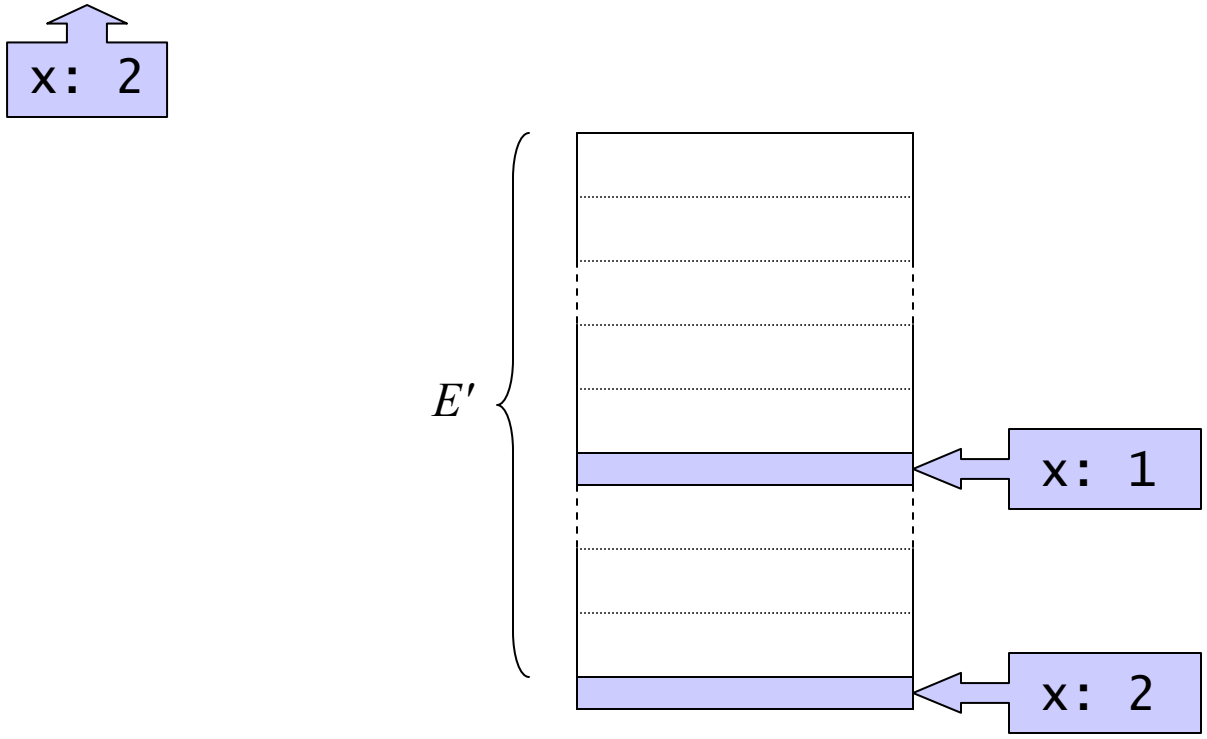
Continuation marks and tail calls

E'[mark x = 2 in mark x = 3 in e end end]



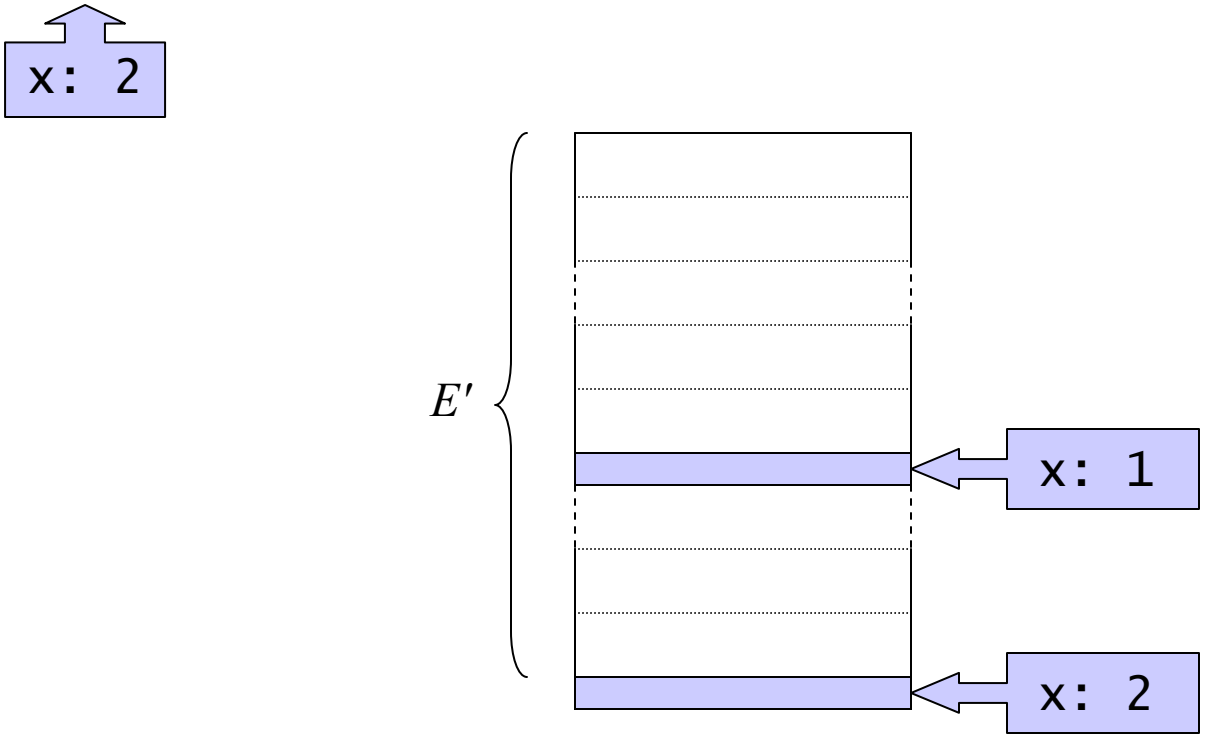
Continuation marks and tail calls

E'[mark x = 2 in mark x = 3 in e end end]



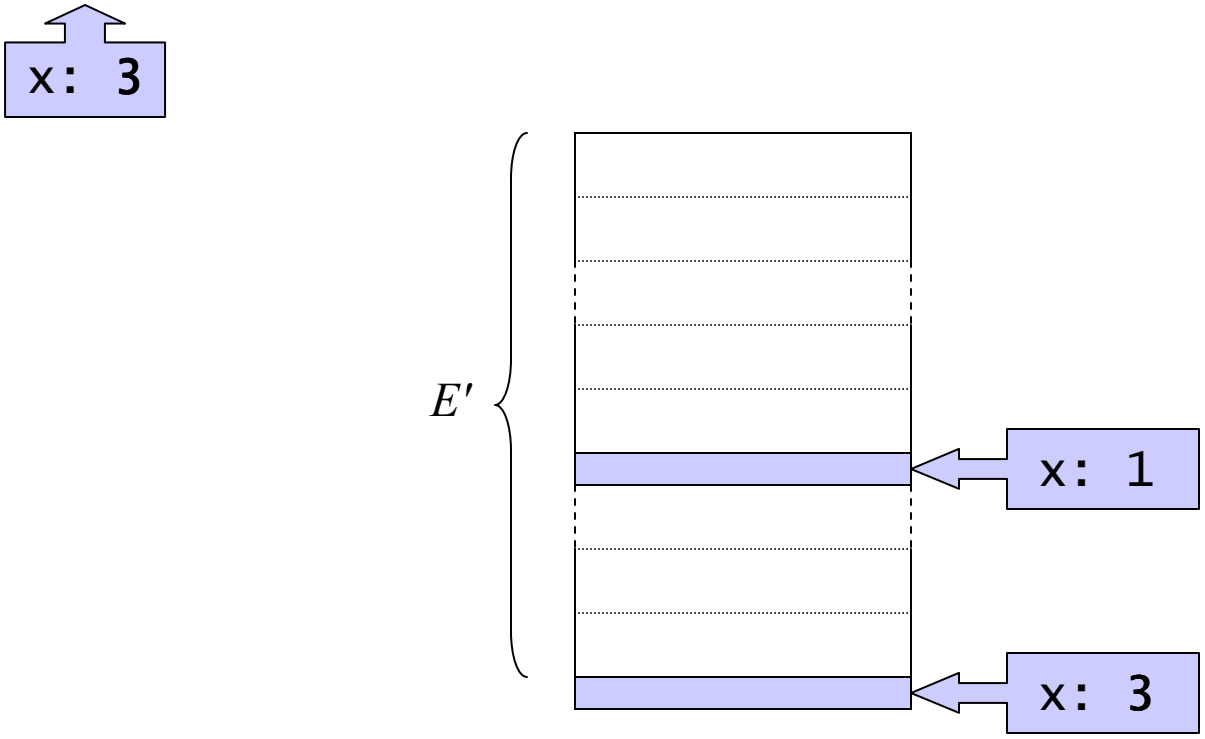
Continuation marks and tail calls

E'[mark x = 3 in e end]

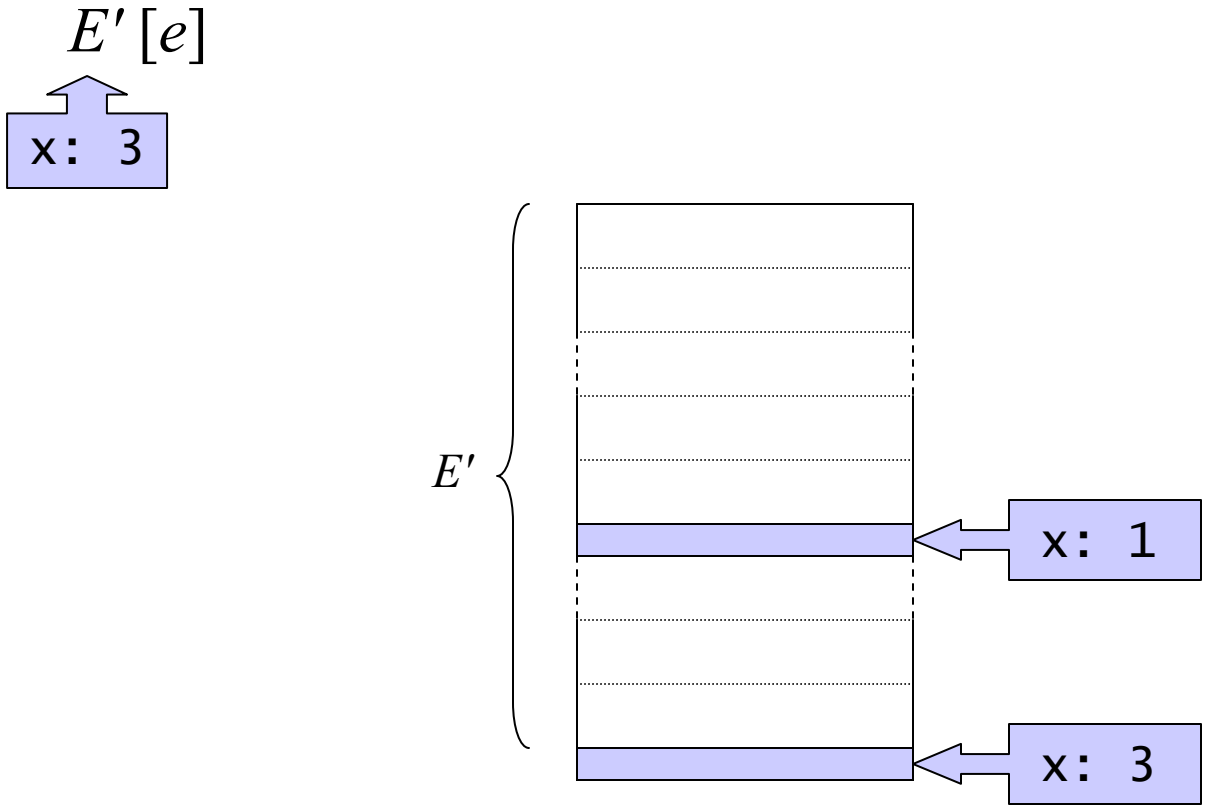


Continuation marks and tail calls

E'[`mark x = 3 in` e `end`]

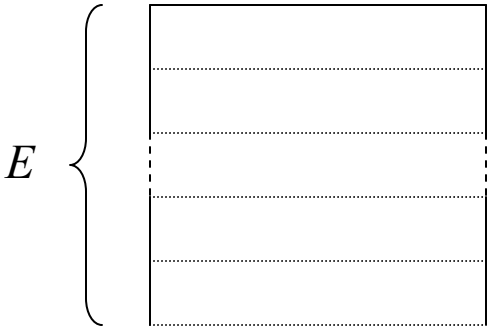


Continuation marks and tail calls



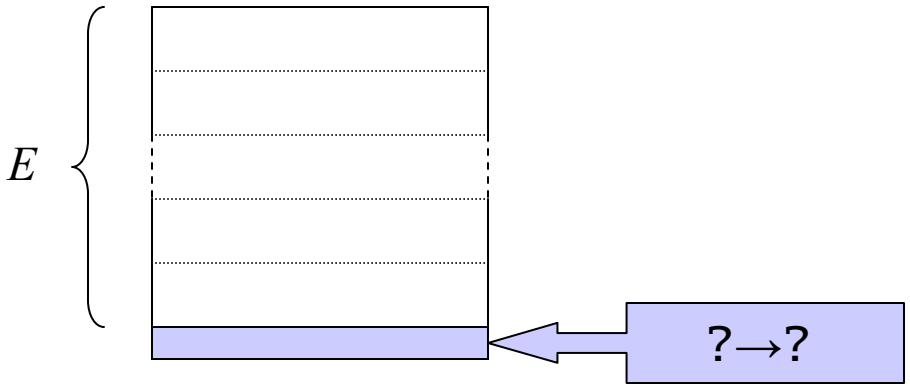
Coercions as continuation marks

E [<?→?><Bool→Bool> e]



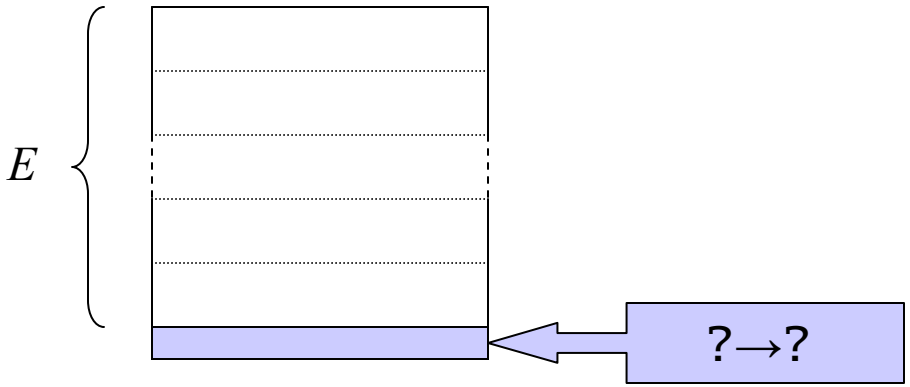
Coercions as continuation marks

E [<?→?><Bool→Bool> e]



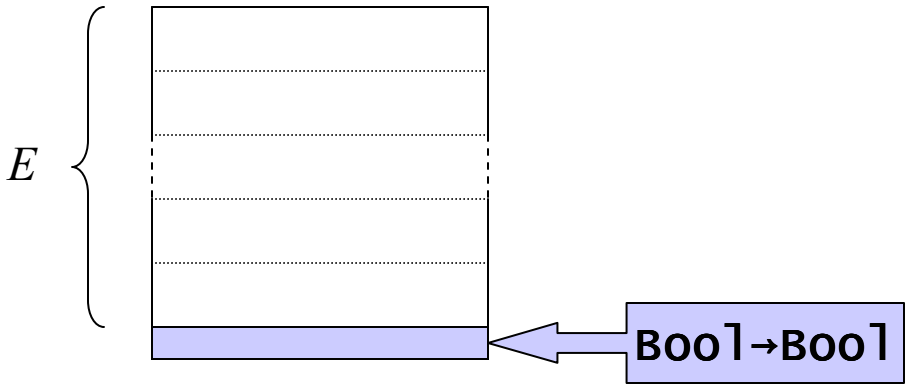
Coercions as continuation marks

E [<Bool→Bool> e]



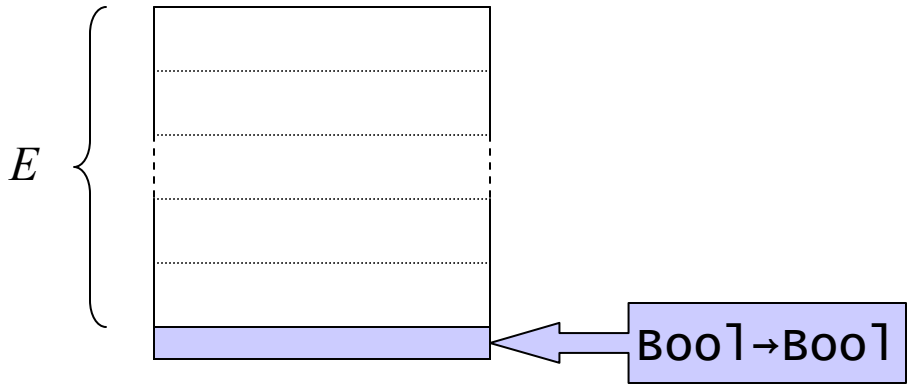
Coercions as continuation marks

E [<Bool→Bool> e]



Coercions as continuation marks

$E\,[e]$



Alternative approaches

- Coercion-passing style

```
λ(x,c).f(x,simplify(c∘d))
```

- Trampoline

```
λ(x).(d,λ().f(x))
```

Parting Thoughts

Related work

- Gradual typing
  - Siek and Taha (2006, 2007)
- Function proxies
  - Findler and Felleisen (1998, 2006): *Software contracts*
  - Gronski, Knowles, Tomb, Freund, Flanagan (2006): *Hybrid typing*, *Sage*
  - Tobin-Hochstadt and Felleisen (2006): *Interlanguage migration*
- Coercions
  - Henglein (1994): *Dynamic typing*
- Space efficiency
  - Clinger (1998): *Proper tail recursion*

Contributions

- Space-safe representation and semantics of casts for functional languages
- Supports function casts and tail recursion
- Earlier error detection
- Proof of space efficiency
- Three implementation strategies

The point, again

Naïve type conversions in functional programming languages are not safe for space.

But they can and should be.

Thank you.

dherman@ccs.neu.edu