A Fix for Dynamic Scope

Ravi Chugh
U. of California, San Diego
Backstory

Goal: Types for “Dynamic” Languages

Syntax and Semantics of JavaScript

Translation à la Guha et al. (ECOOP 2010)

Core $\lambda$-Calculus + Extensions
Backstory

Goal: Types for “Dynamic” Languages

Approach: Type Check Desugared Programs

Simple Imperative, Recursive
Functions are Difficult to Verify

Core $\lambda$-Calculus + Extensions
```javascript
var fact = function(n) {
    if (n <= 1) { return 1; }
    else { return n * fact(n-1); }
};

Translate to statically typed calculus?
```
let fact = \( \lambda n. \) if \( n \leq 1 \) then 1 else \( n \times \text{fact}(n-1) \)

Option 1: λ-Calculus (\( \lambda \))

let fact = \( \lambda n. \) if \( n \leq 1 \) then 1 else \( n \times \text{fact}(n-1) \)

\text{fact} \text{ is not bound yet}...
let fact = fix (\fact. \n. if n <= 1 then 1 else n * fact(n-1))

Desugared fact should be mutable…

Option 2: λ-Calculus + Fix (\fix)

let fact = fix (\fact. \n. if n <= 1 then 1 else n * fact(n-1))

Translate to statically typed calculus?
let fact = ref (fix (λfact. λn.
if n <= 1 then 1 else n * fact(n-1)
))

Recursive call doesn’t go through reference…
Okay here, but prevents **mutual** recursion
let fact = ref (λn.
  if n <= 1 then 1 else n * !fact(n-1)
)

Yes, this captures the semantics!
But how to type check ???
let fact = ref (λn.
    if n <= 1 then 1 else n * !fact(n-1)
)
let fact = ref (\n . 0) in

fact := \n .
    if n <= 1 then 1 else n * !fact(n-1)

Assignment relies on and guarantees the reference invariant.
Backpatching Pattern in $\lambda_{ref}$

Simple types suffice if reference is **initialized** with dummy function...

```lambda
let fact = ref (\n. 0) in

fact := \n.
  if n <= 1 then 1 else n * !fact(n-1)
;
```

But we want to **avoid** initializers to:

- Facilitate JavaScript translation
- Have some fun!
Proposal: Open Types for $\lambda_{\text{ref}}$

**fact** :: (fact :: Ref (Int $\rightarrow$ Int)) $\Rightarrow$ Ref (Int $\rightarrow$ Int)

Assuming **fact** points to an integer function, **fact** points to an integer function

```
let fact = ref (\n  if n <= 1 then 1 else n * !fact(n-1)
)
```
Proposal: Open Types for $\lambda_{\text{ref}}$

\[
\text{let fact} = \text{ref} (\lambda n. \\
\quad \text{if } n \leq 1 \text{ then } 1 \text{ else } n \times \! \text{fact}(n-1) \\
\quad ) \\
\text{in } \! \text{fact}(5)
\]

Assuming \texttt{fact} points to an integer function, \texttt{fact} points to an integer function.

Type system must check assumptions at every dereference.
Motivated by $\lambda_{\text{ref}}$ programs that result from desugaring JavaScript programs

```ocaml
let fact = ref (\n  n ->
    if n <= 1 then 1 else n * !fact(n-1)
  )
```
Motivated by $\lambda_{\text{ref}}$ programs that result from desugaring JavaScript programs

Proposal also applies to, and is easier to show for, the dynamically-scoped $\lambda$-calculus ($\lambda^{\text{dyn}}$)

```plaintext
let fact = \n. if n <= 1 then 1 else n * fact(n-1)
```

Bound at the time of call...

No need for fix or references
Lexical vs. Dynamic Scope

\[ e ::= c \mid \lambda x.e \mid x \mid e_1 e_2 \mid \text{let } x \ e_1 e_2 \mid \text{fix } e \text{ (lexical only)} \]

\[ \lambda \]

Environment at definition is frozen in closure...

And is used to evaluate function body

Explicit evaluation rule for recursion
Lexical vs. Dynamic Scope

\[ e ::= c \mid \lambda x.e \mid x \mid e_1 \; e_2 \mid \text{let } x \; e_1 \; e_2 \mid \text{fix } e \text{ (lexical only)} \]

**λ**

\[
E \vdash \lambda x.e \Downarrow (\lambda x.e, E)
\]
\[
E \vdash e_1 \Downarrow (\lambda x.e, E')
\]
\[
E' \xrightarrow{x \mapsto v_2} E \vdash e \Downarrow v
\]
\[
E \vdash e_1 \; e_2 \Downarrow v
\]

\[
E \vdash e[\text{fix } \lambda x.e / x] \Downarrow v
\]
\[
E \vdash \text{fix } \lambda x.e \Downarrow v
\]

**λ\text{dyn}**

\[
E \vdash \lambda x.e \Downarrow \lambda x.e\text{ [E-FUN]}
\]
\[
E \vdash e_1 \Downarrow \lambda x.e
\]
\[
E \vdash e_2 \Downarrow v_2
\]
\[
E, x \mapsto v_2 \vdash e \Downarrow v\text{ [E-APP]}
\]
\[
E \vdash e_1 \; e_2 \Downarrow v
\]

Bare lambdas, so all free variables resolved in calling environment
Lexical vs. Dynamic Scope

\[ e ::= c \mid \lambda x.e \mid x \mid e_1 e_2 \mid \text{let } x \ e_1 \ e_2 \mid \text{fix } e \ (\text{lexical only}) \]

\[ \lambda \]

\[ \begin{align*}
E \vdash \lambda x.e \downarrow (\lambda x.e, E) \\
E \vdash e_1 \downarrow (\lambda x.e, E') \\
E \vdash e_2 \downarrow v_2 \\
E', x \mapsto v_2 \vdash e \downarrow v \\
E \vdash e_1 e_2 \downarrow v
\end{align*} \]

\[ \lambda_{\text{dyn}} \]

\[ \begin{align*}
E \vdash \lambda x.e \downarrow \lambda x.e \quad [\text{E-FUN}] \\
E \vdash e_1 \downarrow \lambda x.e \\
E \vdash e_2 \downarrow v_2 \\
E, x \mapsto v_2 \vdash e \downarrow v \quad [\text{E-APP}] \\
E \vdash e_1 e_2 \downarrow v
\end{align*} \]

\[ \begin{align*}
E \vdash e[\text{fix } \lambda x.e / x] \downarrow v \\
E \vdash \text{fix } \lambda x.e \downarrow v
\end{align*} \]

No need for fix construct
Open Types for $\lambda^{\text{dyn}}$

\[
\begin{align*}
T & ::= B \mid T_1 \to T_2 \quad \text{Types (Base or Arrow)} \\
S & ::= (\Gamma) \Rightarrow T \quad \text{Open Types} \\
\Gamma & ::= \Gamma, x : T \mid \emptyset \quad \text{Type Environments} \\
\gamma & ::= \gamma, x : S \mid \emptyset \quad \text{Open Type Environments}
\end{align*}
\]
Open Types for $\lambda^{\text{dyn}}$

\[ T ::= B \mid T_1 \rightarrow T_2 \]
\[ S ::= (\Gamma) \Rightarrow T \]
\[ \Gamma ::= \Gamma, x:T \mid \emptyset \]
\[ \gamma ::= \gamma, x:S \mid \emptyset \]

Open Typing \[ \gamma \vdash e :: S \]

\( \text{Lift}(\Gamma, x:T_1) \vdash e :: T_2 \)
\[ \gamma \vdash \lambda x.e :: (\Gamma) \Rightarrow T_1 \rightarrow T_2 \quad [\text{T-FUN}] \]

- Open function type describes environment for function body
Open Types for $\lambda^{\text{dyn}}$

$T ::= B \mid T_1 \rightarrow T_2$

$S ::= (\Gamma) \Rightarrow T$

$\Gamma ::= \Gamma, x:T \mid \emptyset$

$\gamma ::= \gamma, x:S \mid \emptyset$

**Open Typing**

$\gamma \vdash e :: S$

$Lift(\Gamma, x:T_1) \vdash e :: T_2$

$\gamma \vdash \lambda x.e :: (\Gamma) \Rightarrow T_1 \rightarrow T_2$ [T-FUN]

- Open function type describes environment for function body
- Type environment at function definition is **irrelevant**
Open Types for $\lambda^\text{dyn}$

$T ::= B \mid T_1 \to T_2$

$S ::= (\Gamma) \Rightarrow T$

$\Gamma ::= \Gamma, x:T \mid \emptyset$

$\gamma ::= \gamma, x:S \mid \emptyset$

Open Typing $\gamma \vdash e :: S$

```
\text{STLC + Fix}
```

```
\frac{\Gamma \vdash e :: T_2}{\Gamma \vdash \lambda x.e :: (\Gamma) \Rightarrow T_1 \to T_2} \quad \text{[T-FUN]}
```

```
\text{In some sense, extreme generalization of [T-Fix]}
```

```
\frac{\Gamma, x:T_1 \vdash e :: T_2}{\Gamma \vdash \lambda x.e :: T_1 \to T_2}
```

```
\frac{\Gamma, f:T \vdash e :: T}{\Gamma \vdash \text{fix}\, \lambda f.e :: T}
```

22
Open Types for $\lambda^\text{dyn}$

$T ::= B \mid T_1 \to T_2$

$S ::= (\Gamma) \Rightarrow T$

$\Gamma ::= \Gamma, x: T \mid \emptyset$

$\gamma ::= \gamma, x: S \mid \emptyset$

**Open Typing**

$\gamma \vdash e :: S$

**STLC + Fix**

$\Gamma \vdash e_1 :: T_1 \to T_2$

$\Gamma \vdash e_2 :: T_1$

$\Gamma \vdash e_1 \ e_2 :: T_2$

**Lift**

$\frac{\text{Lift}(\Gamma, x:T_1) \vdash e :: T_2}{\gamma \vdash \lambda x. e :: (\Gamma) \Rightarrow T_1 \to T_2}$  \hfill [T-FUN]

**RelyGuarantee**

$\gamma \vdash e_1 :: (\Gamma) \Rightarrow T_1 \to T_2$

$\gamma \vdash e_2 :: (\emptyset) \Rightarrow T_1$

$\text{RelyGuarantee}(\gamma) = (\Gamma_r, \Gamma_g)$

$\Gamma_g \supseteq \Gamma_r \quad \Gamma_g \supseteq \Gamma$

$\gamma \vdash e_1 \ e_2 :: T_2$  \hfill [T-APP]
Open Types for $\lambda^{\text{dyn}}$

For every $(x: (\Gamma) \Rightarrow T) \in \gamma$

- **Rely-set** $\Gamma_r$ contains $\Gamma$
- **Guarantee-set** $\Gamma_g$ contains $x:T$

(most recent, if multiple)

$$
\gamma \vdash e_1 :: (\Gamma) \Rightarrow T_1 \rightarrow T_2 \\
\gamma \vdash e_2 :: (\emptyset) \Rightarrow T_1 \\
\text{RelyGuarantee}(\gamma) = (\Gamma_r, \Gamma_g) \\
\Gamma_g \supseteq \Gamma_r \quad \Gamma_g \supseteq \Gamma \\
\frac{}{\gamma \vdash e_1 \ e_2 :: T_2} \quad \text{[T-App]}
$$
For every $(x : (\Gamma) \Rightarrow T) \in \gamma$

- **Rely-set** $\Gamma_r$ contains $\Gamma$
- **Guarantee-set** $\Gamma_g$ contains $x : T$

(most recent, if multiple)

\[
\Gamma_g \supseteq \Gamma_r \quad \Gamma_g \supseteq \Gamma
\]

\[
\gamma \vdash e_1 :: (\Gamma) \Rightarrow T_1 \rightarrow T_2 \\
\gamma \vdash e_2 :: (\emptyset) \Rightarrow T_1
\]

\[
\text{RelyGuarantee}(\gamma) = (\Gamma_r, \Gamma_g)
\]

\[
\gamma \vdash e_1 \; e_2 :: T_2
\]

[T-App]
Open Types for $\lambda^{\text{dyn}}$

\[
T ::= B \mid T_1 \rightarrow T_2
\]
\[
S ::= (\Gamma) \Rightarrow T
\]
\[
\Gamma ::= \Gamma, x : T \mid \emptyset
\]
\[
\gamma ::= \gamma, x : S \mid \emptyset
\]

Open Typing

\[
\gamma \vdash e :: S
\]

\[
\text{Lift}(\Gamma, x : T_1) \vdash e :: T_2 \\
\gamma \vdash \lambda x.e :: (\Gamma) \Rightarrow T_1 \rightarrow T_2 \quad \text{[T-FUN]}
\]

\[
\gamma \vdash e_1 :: (\Gamma) \Rightarrow T_1 \rightarrow T_2 \\
\gamma \vdash e_2 :: (\emptyset) \Rightarrow T_1 \\
RelyGuarantee(\gamma) = (\Gamma_r, \Gamma_g) \\
\Gamma_g \supseteq \Gamma_r \quad \Gamma_g \supseteq \Gamma \\
\gamma \vdash e_1 e_2 :: T_2 \quad \text{[T-APP]}
\]
Examples

\[
\text{let } \text{fact} = \lambda n. \\
\quad \text{if } n \leq 1 \text{ then } 1 \\
\quad \quad \text{else } n \ast \text{fact}(n-1) \\
\text{in } \text{fact } 5
\]

\[
\text{fact :: (fact :: Int } \rightarrow \text{ Int) } \Rightarrow \text{ Int } \rightarrow \text{ Int}
\]
let fact = \n. 
  if n <= 1 then 1 
  else n * fact(n-1) 

in fact 5

Guarantee ⇐ Rely
let tick n =
  if n > 0 then “tick ” ++ tock n
  else “” in

tick 2; Error: var [tock] not found
let tick n =
  if n > 0 then “tick ” ++ tock n
  else “” in

tick 2;

tick :: (tock :: Int → Str) ⇒ Int → Str
let tick n =
  if n > 0 then “tick ” ++ tock n
  else “” in

tick 2;

tick :: (tock :: Int → Str) ⇒ Int → Str

Guarantee ⇄ Rely
let tick n =
  if n > 0 then “tick ” ++ tock n
  else “” in

let tock n =
  “tock ” ++ tick (n-1) in

tick 2; ↓ “tick tock tick tick tock ”
Examples

```haskell
let tick n = if n > 0 then "tick " ++ tock n else "" in

let tock n = "tock " ++ tick (n-1) in

tick 2;
```

```
tick :: (tock :: Int -> Str) -> Int -> Str

tock :: (tick :: Int -> Str) -> Int -> Str
```
Examples

let tick n =
    if n > 0 then “tick ” ++ tock n
    else “” in

let tock n =
    “tock ” ++ tick (n-1) in

tick 2;

tick :: (tock :: Int → Str) ⇒ Int → Str
tock :: (tick :: Int → Str) ⇒ Int → Str

Guarantee ⇐ Rely
let tick n =
  if n > 0 then "tick " ++ tock n
  else "" in

let tock n =
  "tock " ++ tick (n-1) in

let tock =
  "bad" in

tick 2; ↓ Error: "bad" not a function
Examples

let tick n =
    if n > 0 then "tick " ++ tock n
    else "" in

let tock n =
    "tock " ++ tick (n-1) in

let tock =
    "bad" in

tick 2;

tick :: (tock :: Int -> Str) => Int -> Str
tock :: (tick :: Int -> Str) => Int -> Str
tock :: Str
let tick n =
  if n > 0 then “tick ” ++ tock n
  else “” in

let tock n =
  “tock ” ++ tick (n-1) in

let tock =
  “bad” in

tick 2;

 Guarantee  Rely
Examples

```ocaml
let tick n =  
  if n > 0 then "tick " ++ tock n  
  else "" in

let tock n =  
  "tock " ++ tick (n-1) in

let tock =  
  tick (n-1) in

tick 2; \(\Downarrow\) "tick tick "
```
let tick n =
    if n > 0 then "tick " ++ tock n
    else "" in

let tock n =
    "tock " ++ tick (n-1) in

let tock =
    tick (n-1) in

tick 2;
let tick n = 
  if n > 0 then “tick ” ++ tock n 
  else “” in

let tock n = 
  “tock ” ++ tick (n-1) in

let tock = 
  tick (n-1) in

tick 2;

Guarantee ⇒ Rely
Open Types capture “late packaging” of recursive definitions in dynamically-scoped languages

(Metatheory has not yet been worked)

Several potential applications in lexically-scoped languages...
For open types in this setting, type system must support strong updates to mutable variables

References in $\lambda_{ref}$ are similar to dynamically-scoped bindings in $\lambda_{dyn}$

e.g. Thiemann et al. (ECOOP 2010), Chugh et al. (OOPSLA 2012)
let tick = ref null in

let tock = ref null in

tick := \n. if n > 0 then "tick " ++ !tock(n) else "";

tock := \n. "tock " ++ !tick(n-1);

!tick(2);

Guarantee \( \Rightarrow \) Rely
let tick n =
  if n > 0 then "tick " ++ this.tock(n)
  else "" in
let tock n = "tock " ++ this.tick(n-1) in
let obj = {tick=tick; tock=tock} in
obj.tick(2);

**Open Types for Methods**

```
tick :: (this :: {tock: Int -> Str}) => Int -> Str
tock :: (this :: {tick: Int -> Str}) => Int -> Str
```
Related Work

• Dynamic scope in lexical settings for:
  – Software updates
  – Dynamic code loading
  – Global flags

• **Implicit Parameters** of Lewis et al. (POPL 2000)
  – Open types mechanism resembles their approach
  – We aim to support recursion and references

• Other related work
  – Bierman et al. (ICFP 2003)
  – Dami (TCS 1998), Harper (Practical Foundations for PL)
  – ...

Open Types for Checking Recursion in:

1. Dynamically-scoped $\lambda$-calculus
2. Lexically-scoped $\lambda$-calculus with references

Thoughts? Questions?
EXTRA SLIDES
Higher-Order Functions

\[
\gamma \vdash e_1 :: (\Gamma) \Rightarrow T_1 \rightarrow T_2 \\
\gamma \vdash e_2 :: (\emptyset) \Rightarrow T_1 \\
\text{RelyGuarantee}(\gamma) = (\Gamma_r, \Gamma_g) \\
\Gamma_g \supseteq \Gamma_r \quad \Gamma_g \supseteq \Gamma \\
\gamma \vdash e_1 \cdot e_2 :: T_2
\]

- Disallows function arguments with free variables
- This restriction precludes “downwards funarg problem”
- To be less restrictive:

\[
\gamma \vdash e_1 :: (\Gamma) \Rightarrow T_1 \rightarrow T_2 \\
\gamma \vdash e_2 :: (\Gamma') \Rightarrow T_1 \\
\text{RelyGuarantee}(\gamma) = (\Gamma_r, \Gamma_g) \\
\Gamma_g \supseteq \Gamma_r \quad \Gamma_g \supseteq \Gamma \quad \Gamma \supseteq \Gamma' \\
\gamma \vdash e_1 \cdot e_2 :: T_2
\]
let negateInt () = 0 - x in
let negateBool () = not x in
let x = 17 in

Safe at run-time even though not all assumptions satisfied

\[ \text{negateInt} () \downarrow -17 \]

\[
\text{negateInt} :: (x :: \text{Int}) \rightarrow \text{Unit} \rightarrow \text{Int}
\]

\[
\text{negateBool} :: (x :: \text{Bool}) \rightarrow \text{Unit} \rightarrow \text{Bool}
\]

\[
x :: \text{Int}
\]
Partial Environment Consistency

• To be less restrictive:

\[
\gamma \vdash e_1 :: (\Gamma) \Rightarrow T_1 \rightarrow T_2
\]

\[
\ldots
\]

\[
\text{PartialRely}(\gamma, \Gamma) = \Gamma_r
\]

\[
\text{Guarantee}(\gamma) = \Gamma_g
\]

\[
\Gamma_g \supseteq \Gamma_r \quad \ldots
\]

\[
\gamma \vdash e_1 \ e_2 :: T_2
\]

Only \( \Gamma \) and its transitive dependencies in \( \gamma \)