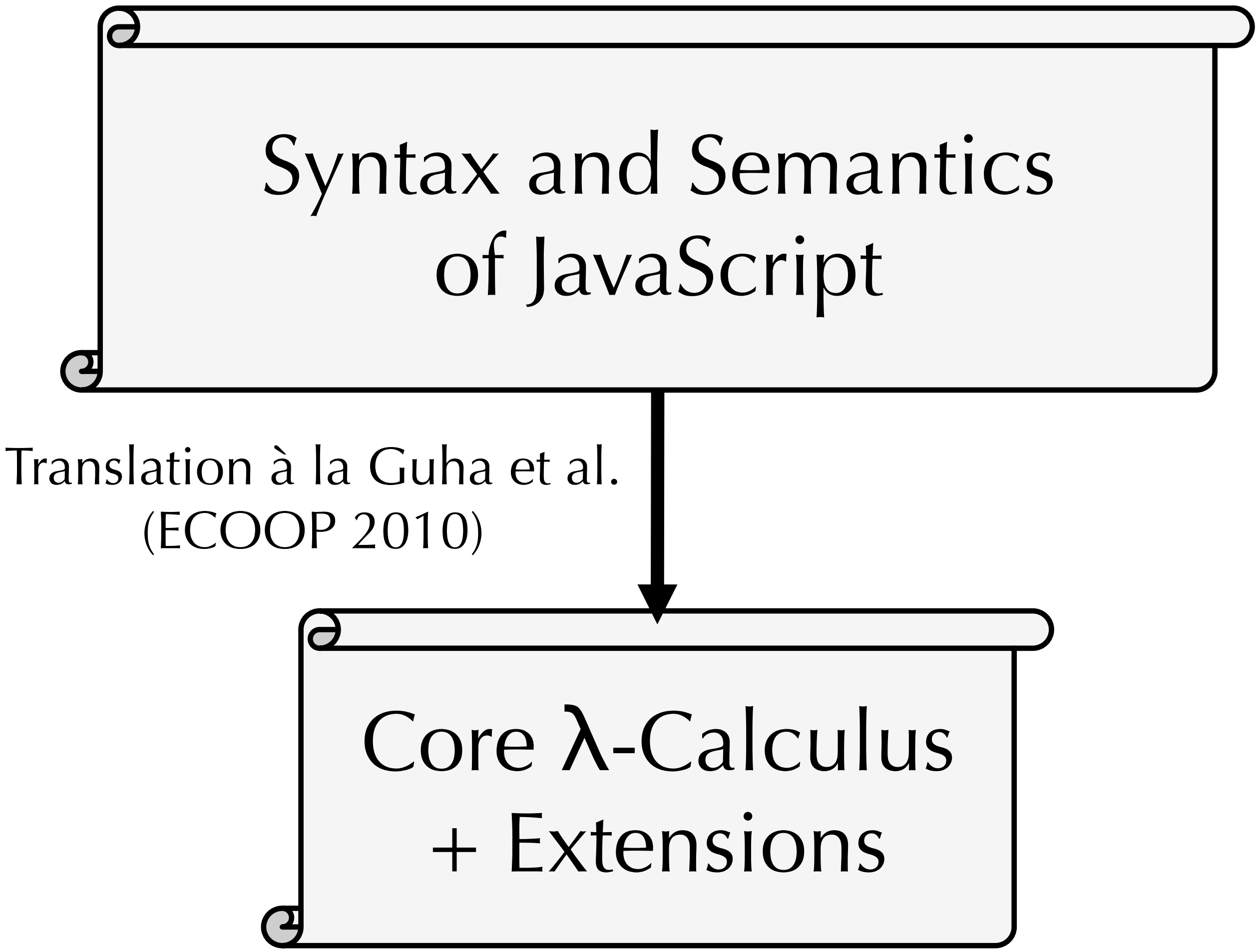
A Fix for Dynamic Scope

Ravi Chugh
U. of California, San Diego

Backstory

Goal: Types for “Dynamic” Languages



Backstory

Goal: Types for "Dynamic" Languages

Approach: Type Check Desugared Programs

Simple **Imperative**, **Recursive**
Functions are Difficult to Verify

Core λ-Calculus
+ Extensions

```
var fact = function(n) {
  if (n <= 1) { return 1;                }
  else        { return n * fact(n-1); }
};
```

Translate to statically typed calculus?

```
var fact = function(n) {
  if (n <= 1) { return 1;               }
  else        { return n * fact(n-1); }
};
```

Translate to statically typed calculus?

Option 1: λ-Calculus (λ) ✗

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

`fact` is not bound yet…

```
var fact = function(n) {
   if (n <= 1) { return 1;              }
   else        { return n * fact(n-1); }
};
```

Translate to statically typed calculus?

Option 2: λ-Calculus + Fix (λ_{fix}) ✗

```
let fact = fix (λfact. λn.
   if n <= 1 then 1 else n * fact(n-1)
)
```

Desugared `fact` should be mutable…

```
var fact = function(n) {
  if (n <= 1) { return 1;              }
  else        { return n * fact(n-1); }
};
```

Translate to statically typed calculus?

Option 3: λ-Calculus + Fix + Refs ($\lambda_{fix,ref}$) ✗

```
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

```
var fact = function(n) {
  if (n <= 1) { return 1;                 }
  else        { return n * fact(n-1); }
};
```

Translate to statically typed calculus?

Option 4: λ-Calculus + Fix + Refs (λ_{ref}) ✓

```
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)
)
```

Backpatching Pattern in λ_{ref}

fact :: Ref (Int → Int)

```
let fact = ref (λn. 0) in
fact := λn.
  if n <= 1 then 1 else n * !fact(n-1)
;
```

Assignment **relies on** and **guarantees** the reference invariant

Backpatching Pattern in λ_{ref}

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

```
let fact = ref (λn. 0) in
fact := λn.
  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 λ_{ref}

fact :: (fact :: Ref (Int → Int)) ⇒ Ref (Int → Int)

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

fact points to an integer function

```
let fact = ref (λn.
  if n <= 1 then 1 else n * !fact(n-1)
)
```

Proposal: **Open Types** for λ_{ref}

`fact` :: (`fact` :: Ref (Int → Int)) ⇒ Ref (Int → Int)

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

```
let fact = ref (λn.
  if n <= 1 then 1 else n * !fact(n-1)
)
in !fact(5)
```

Type system must check assumptions at every dereference

Proposal: **Open Types** for λ_{ref}

Motivated by λ_{ref} programs that result from desugaring JavaScript programs

```
let fact = ref (λn.
  if n <= 1 then 1 else n * !fact(n-1)
)
```

Proposal: **Open Types** for λ_{ref}

Motivated by λ_{ref} programs that result from desugaring JavaScript programs

Proposal also applies to, and is easier to show for, the **dynamically-scoped λ-calculus (λ^{dyn})**

```
let fact = λn.
  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

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

λ

$$\frac{}{E \vdash \lambda x.e \Downarrow (\lambda x.e, E)}$$

$$\frac{E \vdash e_1 \Downarrow (\lambda x.e, E') \quad E \vdash e_2 \Downarrow v_2 \quad E', x \mapsto v_2 \vdash e \Downarrow v}{E \vdash e_1 \; e_2 \Downarrow v}$$

$$\frac{E \vdash e[\texttt{fix} \; \lambda x.e \, / \, x] \Downarrow v}{E \vdash \texttt{fix} \; \lambda x.e \Downarrow v}$$

Environment at definition is frozen in closure…

And is used to evaluate function body

Explicit evaluation rule for recursion

Lexical vs. Dynamic Scope

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

λ

$$\frac{}{E \vdash \lambda x.e \Downarrow (\lambda x.e, E)}$$

$$\frac{E \vdash e_1 \Downarrow (\lambda x.e, E') \quad E \vdash e_2 \Downarrow v_2 \quad E', x \mapsto v_2 \vdash e \Downarrow v}{E \vdash e_1\ e_2 \Downarrow v}$$

$$\frac{E \vdash e[\texttt{fix}\ \lambda x.e \,/\, x] \Downarrow v}{E \vdash \texttt{fix}\ \lambda x.e \Downarrow v}$$

λ^dyn

$$\frac{}{E \vdash \lambda x.e \Downarrow \lambda x.e}\ \text{[E-Fun]}$$

$$\frac{E \vdash e_1 \Downarrow \lambda x.e \quad E \vdash e_2 \Downarrow v_2 \quad E, x \mapsto v_2 \vdash e \Downarrow v}{E \vdash e_1\ e_2 \Downarrow v}\ \text{[E-App]}$$

Bare lambdas, so **all** free variables resolved in calling environment

Lexical vs. Dynamic Scope

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

λ

$$\frac{}{E \vdash \lambda x.e \Downarrow (\lambda x.e, E)}$$

$$\frac{E \vdash e_1 \Downarrow (\lambda x.e, E') \quad E \vdash e_2 \Downarrow v_2 \quad E', x \mapsto v_2 \vdash e \Downarrow v}{E \vdash e_1\ e_2 \Downarrow v}$$

$$\frac{E \vdash e[\texttt{fix}\ \lambda x.e\,/\,x] \Downarrow v}{E \vdash \texttt{fix}\ \lambda x.e \Downarrow v}$$

λ^dyn

$$\frac{}{E \vdash \lambda x.e \Downarrow \lambda x.e}\ \text{[E-Fun]}$$

$$\frac{E \vdash e_1 \Downarrow \lambda x.e \quad E \vdash e_2 \Downarrow v_2 \quad E, x \mapsto v_2 \vdash e \Downarrow v}{E \vdash e_1\ e_2 \Downarrow v}\ \text{[E-App]}$$

No need for fix construct

Open Types for λ^{dyn}

T	$::=$	$B \mid T_1 \to T_2$	Types (Base or Arrow)
S	$::=$	$(\Gamma) \Rightarrow T$	Open Types
Γ	$::=$	$\Gamma, x:T \mid \emptyset$	Type Environments
γ	$::=$	$\gamma, x:S \mid \emptyset$	Open Type Environments

Open Types for λ^{dyn}

$$
\begin{array}{rcl}
T & ::= & B \mid T_1 \to T_2 \\
S & ::= & (\Gamma) \Rightarrow T \\
\Gamma & ::= & \Gamma, x:T \mid \emptyset \\
\gamma & ::= & \gamma, x:S \mid \emptyset
\end{array}
$$

Open Typing $\boxed{\gamma \vdash e :: S}$

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

- Open function type describes environment for function body

Open Types for λ^{dyn}

$$
\begin{array}{lcl}
T & ::= & B \mid T_1 \to T_2 \\
S & ::= & (\Gamma) \Rightarrow T \\
\Gamma & ::= & \Gamma, x : T \mid \emptyset \\
\gamma & ::= & \gamma, x : S \mid \emptyset
\end{array}
$$

Open Typing $\boxed{\gamma \vdash e :: S}$

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

- Open function type describes environment for function body
- Type environment at function definition is **irrelevant**

Open Types for λ^{dyn}

$$
\begin{array}{lcl}
T & ::= & B \mid T_1 \to T_2 \\
S & ::= & (\Gamma) \Rightarrow T \\
\Gamma & ::= & \Gamma, x\!:\!T \mid \emptyset \\
\gamma & ::= & \gamma, x\!:\!S \mid \emptyset
\end{array}
$$

Open Typing $\boxed{\gamma \vdash e :: S}$

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

In some sense, extreme generalization of [T-Fix]

STLC + Fix

$$\frac{\Gamma, x\!:\!T_1 \vdash e :: T_2}{\Gamma \vdash \lambda x.e :: T_1 \to T_2} \qquad \frac{\Gamma, f\!:\!T \vdash e :: T}{\Gamma \vdash \texttt{fix}\ \lambda f.e : T}$$

Open Types for λ^{dyn}

$$
\begin{array}{lcl}
T & ::= & B \mid T_1 \to T_2 \\
S & ::= & (\Gamma) \Rightarrow T \\
\Gamma & ::= & \Gamma, x{:}T \mid \emptyset \\
\gamma & ::= & \gamma, x{:}S \mid \emptyset
\end{array}
$$

STLC + Fix

$$\frac{\Gamma \vdash e_1 :: T_1 \to T_2 \qquad \Gamma \vdash e_2 :: T_1}{\Gamma \vdash e_1\ e_2 :: T_2}$$

Open Typing $\boxed{\gamma \vdash e :: S}$

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

$$\frac{\begin{array}{c}\gamma \vdash e_1 :: (\Gamma) \Rightarrow T_1 \to T_2 \\ \gamma \vdash e_2 :: (\emptyset) \Rightarrow T_1 \\ \text{RelyGuarantee}(\gamma) = (\Gamma_r, \Gamma_g) \\ \Gamma_g \supseteq \Gamma_r \qquad \Gamma_g \supseteq \Gamma\end{array}}{\gamma \vdash e_1\ e_2 :: T_2}\ \text{[T-App]}$$

Open Types for λ^{dyn}

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

- **Rely-set** Γ_r contains Γ
- **Guarantee-set** Γ_g contains $x : T$

(most recent, if multiple)

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

Open Types for λ^{dyn}

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

- **Rely-set** Γ_r contains Γ
- **Guarantee-set** Γ_g contains $x\!:\!T$

(most recent, if multiple)

Discharge all assumptions

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

Open Types for λ^{dyn}

$$
\begin{array}{rcl}
T & ::= & B \mid T_1 \to T_2 \\
S & ::= & (\Gamma) \Rightarrow T \\
\Gamma & ::= & \Gamma, x\!:\!T \mid \emptyset \\
\gamma & ::= & \gamma, x\!:\!S \mid \emptyset
\end{array}
$$

Open Typing $\boxed{\gamma \vdash e :: S}$

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

$$
\frac{\begin{array}{c}\gamma \vdash e_1 :: (\Gamma) \Rightarrow T_1 \to T_2 \\ \gamma \vdash e_2 :: (\emptyset) \Rightarrow T_1 \\ \text{RelyGuarantee}(\gamma) = (\Gamma_r, \Gamma_g) \\ \Gamma_g \supseteq \Gamma_r \qquad \Gamma_g \supseteq \Gamma\end{array}}{\gamma \vdash e_1\ e_2 :: T_2} \;[\text{T-App}]
$$

Examples

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

in fact 5
```

$\texttt{fact} :: (\texttt{fact} :: \mathsf{Int} \rightarrow \mathsf{Int}) \Rightarrow \mathsf{Int} \rightarrow \mathsf{Int}$

Examples

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

in fact 5
```

fact :: (fact :: Int → Int) ⇒ Int → Int

Guarantee $\supseteq$ **Rely** ✓

Examples

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

```
tick 2; ⇓ Error: var [tock] not found
```

Examples

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

tick 2;
```

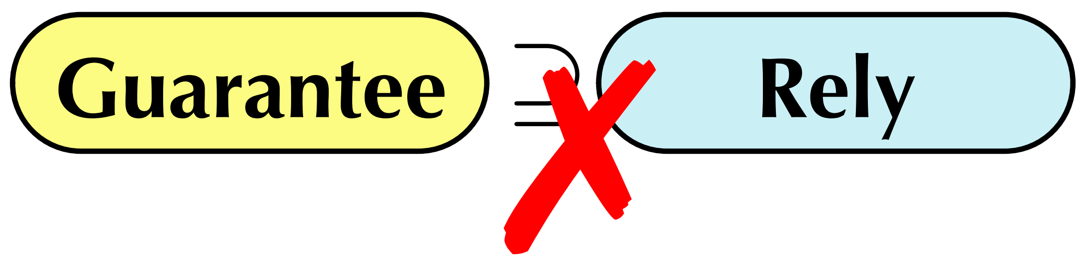
$\texttt{tick} :: (\texttt{tock} :: \text{Int} \rightarrow \text{Str}) \Rightarrow \text{Int} \rightarrow \text{Str}$

Examples

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

tick 2;
```

tick :: (tock :: 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 tick tock ”
```

Examples

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

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

tick 2;
```

$\texttt{tick} :: (\texttt{tock} :: \mathrm{Int} \rightarrow \mathrm{Str}) \Rightarrow \mathrm{Int} \rightarrow \mathrm{Str}$

$\texttt{tock} :: (\texttt{tick} :: \mathrm{Int} \rightarrow \mathrm{Str}) \Rightarrow \mathrm{Int} \rightarrow \mathrm{Str}$

Examples

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

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

tick 2;
```

$\texttt{tick} :: (\texttt{tock} :: \text{Int} \to \text{Str}) \Rightarrow \text{Int} \to \text{Str}$

$\texttt{tock} :: (\texttt{tick} :: \text{Int} \to \text{Str}) \Rightarrow \text{Int} \to \text{Str}$

Guarantee $\supseteq$ **Rely** ✓

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; ⇓ 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;
```

$\texttt{tick} :: (\texttt{tock} :: \text{Int} \rightarrow \text{Str}) \Rightarrow \text{Int} \rightarrow \text{Str}$

$\texttt{tock} :: (\texttt{tick} :: \text{Int} \rightarrow \text{Str}) \Rightarrow \text{Int} \rightarrow \text{Str}$

$\texttt{tock} :: \quad \text{Str}$

Examples

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

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

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

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

```
let tock =
  "bad" in
```

tock :: Str

```
tick 2;
```

Guarantee $\not\Rightarrow$ **Rely**

Examples

```
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; ⇓ “tick tick ”
```

Examples

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

$\texttt{tick} :: (\texttt{tock} :: \mathrm{Int} \to \mathrm{Str}) \Rightarrow \mathrm{Int} \to \mathrm{Str}$

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

$\texttt{tock} :: (\texttt{tick} :: \mathrm{Int} \to \mathrm{Str}) \Rightarrow \mathrm{Int} \to \mathrm{Str}$

```
let tock =
  tick (n-1) in
```

$\texttt{tock} :: (\texttt{tick} :: \mathrm{Int} \to \mathrm{Str}) \Rightarrow \mathrm{Int} \to \mathrm{Str}$

```
tick 2;
```

Examples

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

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

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

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

```
let tock =
  tick (n-1) in
```

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

```
tick 2;
```

Guarantee ⊇ ✓ **Rely**

Recap

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…

Open Types for λ_{ref}

References in λ_{ref} are similar to dynamically-scoped bindings in λ^{dyn}

For open types in this setting,
type system must support
strong updates to mutable variables

e.g. Thiemann et al. (ECOOP 2010), Chugh et al. (OOPSLA 2012)

Open Types for λ_{ref}

```
let tick = ref null in
```

tick :: Ref Null

```
let tock = ref null in
```

tick :: Ref Null
tock :: Ref Null

```
tick := λn.
  if n > 0 then “tick ” ++ !tock(n)
           else “”;
```

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

```
tock := λn.
  “tock ” ++ !tick(n-1);
```

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

```
!tick(2);
```

Guarantee ⊇ ✓ **Rely**

Open Types for Methods

```
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);
```

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)
  - …

Thanks!

Open Types for Checking Recursion in:

1. Dynamically-scoped λ-calculus
2. Lexically-scoped λ-calculus with references

Thoughts? Questions?

EXTRA SLIDES

Higher-Order Functions

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

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

$$\frac{\gamma \vdash e_1 :: (\Gamma) \Rightarrow T_1 \to T_2 \quad \gamma \vdash e_2 :: (\Gamma') \Rightarrow T_1 \quad \text{RelyGuarantee}(\gamma) = (\Gamma_r, \Gamma_g) \quad \Gamma_g \supseteq \Gamma_r \quad \Gamma_g \supseteq \Gamma \quad \Gamma \supseteq \Gamma'}{\gamma \vdash e_1\ e_2 :: T_2}$$

Partial Environment Consistency

```
let negateInt  () = 0 - x in
let negateBool () = not x in
let x = 17 in
negateInt (); ⇓ -17
```

Safe at run-time
even though **not all**
assumptions satisfied

negateInt :: (x :: Int) ⇒ Unit → Int
negateBool :: (x :: Bool) ⇒ Unit → Bool
x :: Int

Partial Environment Consistency

- To be less restrictive:

$$\frac{\gamma \vdash e_1 :: (\Gamma) \Rightarrow T_1 \to T_2 \quad \ldots \quad \text{PartialRely}(\gamma, \Gamma) = \Gamma_r \quad \text{Guarantee}(\gamma) = \Gamma_g \quad \Gamma_g \supseteq \Gamma_r \quad \ldots}{\gamma \vdash e_1\ e_2 :: T_2}$$

Only Γ and its transitive dependencies in γ