Operational Aspects of Type Systems

Inter-Derivable Semantics of Type Checking

and

Gradual Types for Object Ownership

Ilya Sergey

KU LEUVEN

14 November 2012

Types

A type $\mathcal{T}$

- is a set of data instances and operations on them

```
boolean    =   true, false

int        =   0, 1, -1, 2, ...

string     =   “abc”, kuleuven”, ...

array      =   [1, 2, 3], [true, “a”]
```

A type $\mathcal{T}$

- is a statement in a constructive logic

$$(A, B) \to A \qquad \approx \qquad A \wedge B \Rightarrow A$$

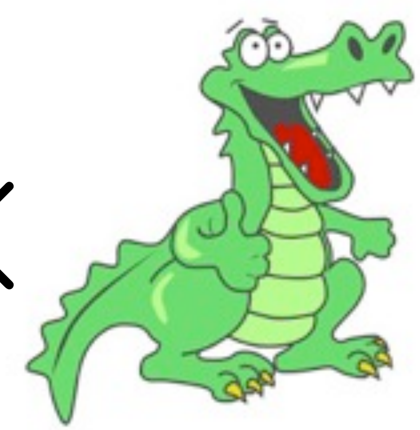
A typed program of type $\mathcal{T}$

- is a *proof* of the statement

$$\lambda(x, y) : (A, B).\ x \qquad \approx \qquad \wedge\text{-left}\ \frac{A \qquad B}{A}$$

Types help to recognize bad programs

$?$

$3\times$  $+\ 2\times$  $=?$

apples

crocodiles

Type Systems

Assigning Types To Programs

$$\overset{\text{Environment}}{\Gamma} \vdash \overset{\text{Program}}{\varsigma} : \overset{\text{Type}}{\tau}$$

- *Well-typed programs cannot go wrong*
  - R. Milner, 1978
- *Well-typed programs cannot get stuck*
  - A. Wright and M. Felleisen, 1992
- *Well-typed programs cannot be blamed*
  - P. Wadler, 2009

Type Systems

Well-Typed Programs Don't Go Wrong

$$\Gamma, \Delta \vdash \varsigma_0 : \tau$$

$$\varsigma_0 \rightarrow \varsigma_1 \rightarrow \varsigma_2 \rightarrow \ldots \rightarrow \varsigma_n \rightarrow \varsigma_{final}$$

$$\varsigma_0 \rightarrow \varsigma_1 \rightarrow \varsigma_2 \rightarrow \ldots \rightarrow \varsigma_n \rightarrow \ldots$$

Type Systems

Well-Typed Programs Don't Go Wrong

$$\Gamma, \Delta \vdash \varsigma_0 : \tau$$

But not

$$\varsigma_0 \rightarrow \varsigma_1 \rightarrow \varsigma_2 \rightarrow \ldots \rightarrow \varsigma_n \rightarrow$$

Type Checking

A Simple Language

Expressions	e	$::=$	$n \mid x \mid \lambda x:\tau.e \mid e\ e$
Numbers	n	$::=$	$number$
Values	v	$::=$	$n \mid \lambda x:\tau.e$
Types	τ	$::=$	$\mathsf{num} \mid \tau \rightarrow \tau$
Typing environments	Γ	$::=$	$\emptyset \mid \Gamma, x:\tau$

$$\text{(t-var)}\ \frac{(x:\tau) \in \Gamma}{\Gamma \vdash x:\tau} \qquad \text{(t-lam)}\ \frac{\Gamma,\ x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \lambda x:\tau_1.e : \tau_1 \rightarrow \tau_2}$$

$$\text{(t-app)}\ \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \text{(t-num)}\ \frac{}{\Gamma \vdash number : \mathsf{num}}$$

Type-checking inference rules

Another ill-typed program
(which also goes wrong)

$$f = \lambda x : \mathsf{num} \to \mathsf{num}.\ \lambda y : \mathsf{num}.\ x\ y\ (\lambda z : \mathsf{num}.\ x\ z)$$

$$(f\ 1)\ 2 =$$ 

Type Checking via Inference Rules

$$
\dfrac{
\dfrac{
\{x : \text{num} \to \text{num}, \ldots\} \vdash x : \text{num} \to \text{num} \qquad \{y : \text{num} \ldots\} \vdash y : \text{num}
}{
\{x : \text{num} \to \text{num}, y : \text{num}\} \vdash x\ y : \overset{\text{num}}{\cancel{(\text{num} \to \text{num})}} \to \tau
}
\qquad
\dfrac{
\dfrac{
\{x : \text{num} \to \text{num}, \ldots\} \vdash x : \text{num} \to \text{num} \qquad \{z : \text{num} \ldots\} \vdash z : \text{num}
}{
\{x : \text{num} \to \text{num}, z : \text{num}, \ldots\} \vdash x\ z : \text{num}
}
}{
\{x : \text{num} \to \text{num}, \ldots\} \vdash \lambda z : \text{num}.\ x\ z : \text{num} \to \text{num}
}
}{
\dfrac{
\{x : \text{num} \to \text{num}, y : \text{num}\} \vdash x\ y\ (\lambda z : \text{num}.\ x\ z) : \tau
}{
\dfrac{
\{x : \text{num} \to \text{num}\} \vdash \lambda y : \text{num}.\ x\ y\ (\lambda z : \text{num}.\ x\ z) : \tau
}{
\emptyset \vdash \lambda x : \text{num} \to \text{num}.\ \lambda y : \text{num}.\ x\ y\ (\lambda z : \text{num}.\ x\ z) : \tau
}
}
}
$$

The Context

Understanding and Tracing a Type System

$$\text{(t-var)}\ \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \text{(t-lam)}\ \frac{\Gamma,\ x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \rightarrow \tau_2}$$

$$\text{(t-app)}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \\ \Gamma \vdash e_2 : \tau_1\end{array}}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \text{(t-num)}\ \frac{}{\Gamma \vdash \mathit{number} : \mathsf{num}}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{k}} M : \Pi^{\mathsf{par}} s{:}\sigma.\, \rho \;\rightsquigarrow\; \Gamma_0 \vdash \bar{M} : \exists \vec{t_0}{::}\hat{\kappa}_0.\, \Pi\bar{\Gamma}.\, \exists \vec{t_1}{::}\hat{\kappa}_1.\, \forall \vec{s}{::}\hat{\bar{\sigma}}.\, \bar{\sigma}\vec{s} \to \exists\bar{\rho} \\ \Gamma \vdash_{\mathsf{P}} N : \sigma \;\rightsquigarrow\; \Gamma_0 \vdash \bar{N} : \Pi\bar{\Gamma}.\, \bar{\sigma}\vec{\tau}\end{array}}{\begin{array}{r}\Gamma \vdash_{\mathsf{k}\sqcup\mathsf{S}} MN : \rho[N/s] \;\rightsquigarrow\; \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t_0}, x\rangle.\, \Lambda\bar{\Gamma}.\, \mathsf{open}\ x\bar{\Gamma}\ \mathsf{as}\ \langle \vec{t_1}, y\rangle.\, y\vec{\tau}(\bar{N}\bar{\Gamma}) \\ : \exists \vec{t_0}{::}\hat{\kappa}_0.\, \Pi\bar{\Gamma}.\, \exists \vec{t_1}{::}\hat{\kappa}_1.\, \exists \vec{t}{::}\hat{\bar{\rho}}.\, \bar{\rho}[\vec{\tau}'/\vec{s}]\vec{t}\end{array}}\;46$$

$$\frac{\ldots : \sigma \;\rightsquigarrow\; \Gamma_0 \vdash \bar{M} : \exists \vec{t_0}{::}\hat{\kappa}_0.\, \Pi\bar{\Gamma}.\, \exists \vec{t_1}{::}\hat{\kappa}_1.\, \bar{\sigma}\vec{\tau} \qquad \Gamma,\ s : \sigma \vdash_{\mathsf{k}} N : \rho \;\rightsquigarrow\; \Gamma_0 \vdash \bar{N} : \exists \vec{t_0'}{::}\hat{\kappa}_0'.\, \Pi\bar{\Gamma}.\, \forall \vec{s}{::}\hat{\bar{\sigma}}.\, \bar{\sigma}\vec{s} \to \exists \vec{t_1'}{::}\hat{\kappa}\ldots}{\begin{array}{r}\ldots M, N\rangle : \Sigma s{:}\sigma.\, \rho \;\rightsquigarrow\; \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t_0}, x\rangle.\, \mathsf{open}\ \bar{N}\ \mathsf{as}\ \langle \vec{t_0'}, y\rangle.\, \Lambda\bar{\Gamma}.\, \mathsf{open}\ x\bar{\Gamma}\ \mathsf{as}\ \langle \vec{t_1}, z\rangle.\, \mathsf{open}\ y\bar{\Gamma}\vec{\tau}z\ \mathsf{as}\ \langle \vec{t_1'}, w\rangle.\, \langle z \ldots \\ : \exists \vec{t_0}{::}\hat{\kappa}_0.\, \exists \vec{t_0'}{::}\hat{\kappa}_0'.\, \Pi\bar{\Gamma}.\, \exists \vec{t_1}{::}\hat{\kappa}_1.\, \exists \vec{t_1'}{::}\hat{\kappa}_1'.\, (\lambda \vec{s}{::}\hat{\bar{\sigma}}.\, \lambda \vec{t}{::}\hat{\bar{\rho}}.\, \bar{\sigma}\vec{s} \times \bar{\rho}\vec{t})\vec{\tau}\ldots\end{array}}$$

$$\frac{\Gamma \vdash_{\mathsf{k}} M : \Sigma s{:}\sigma.\, \rho \;\rightsquigarrow\; \Gamma_0 \vdash \bar{M} : \exists \vec{t_0}{::}\hat{\kappa}_0.\, \Pi\bar{\Gamma}.\, \exists \vec{t_1}{::}\hat{\kappa}_1.\, (\lambda \vec{s}{::}\hat{\bar{\sigma}}.\, \lambda \vec{t}{::}\hat{\bar{\rho}}.\, \bar{\sigma}\vec{s} \times \bar{\rho}\vec{t})\vec{\tau}\vec{\tau}'}{\Gamma \vdash_{\mathsf{k}} \pi_1 M : \sigma \;\rightsquigarrow\; \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t_0}, x\rangle.\, \Lambda\bar{\Gamma}.\, \mathsf{open}\ x\bar{\Gamma}\ \mathsf{as}\ \langle \vec{t_1}, y\rangle.\, \pi_1 y : \exists \vec{t_0}{::}\hat{\kappa}_0.\, \Pi\bar{\Gamma}.\, \exists \vec{t_1}{::}\hat{\kappa}_1.\, \bar{\sigma}\vec{\tau}}\;48$$

$$\frac{\Gamma \vdash_{\mathsf{k}} M : \Sigma s{:}\sigma.\, \rho \;\rightsquigarrow\; \Gamma_0 \vdash \bar{M} : \exists \vec{t_0}{::}\hat{\kappa}_0.\, \Pi\bar{\Gamma}.\, \exists \vec{t_1}{::}\hat{\kappa}_1.\, (\lambda \vec{s}{::}\hat{\bar{\sigma}}.\, \lambda \vec{t}{::}\hat{\bar{\rho}}.\, \bar{\sigma}\vec{s} \times \bar{\rho}\vec{t})\vec{\tau}\vec{\tau}'}{\Gamma \vdash_{\mathsf{k}} \pi_2 M : \sigma \;\rightsquigarrow\; \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t_0}, x\rangle.\, \Lambda\bar{\Gamma}.\, \mathsf{open}\ x\bar{\Gamma}\ \mathsf{as}\ \langle \vec{t_1}, y\rangle.\, \pi_2 y : \exists \vec{t_0}{::}\hat{\kappa}_0.\, \Pi\bar{\Gamma}.\, \exists \vec{t_1}{::}\hat{\kappa}_1.\, \bar{\rho}[\vec{\tau}/\vec{s}]\vec{\tau}'}\;49$$

$$\frac{\Gamma \vdash e : \langle\!|\sigma|\!\rangle \;\rightsquigarrow\; \Gamma_0,\, \bar{\Gamma} \vdash \bar{e} : \exists\bar{\sigma}}{\Gamma \vdash_{\mathsf{S}} \mathsf{unpack}\ e\ \mathsf{as}\ \sigma : \sigma \;\rightsquigarrow\; \Gamma_0 \vdash \Lambda\bar{\Gamma}.\, \bar{e} : \Pi\bar{\Gamma}.\, \exists\bar{\sigma}}\;50$$

$$\frac{\Gamma \vdash_{\mathsf{k}} M : \sigma \;\rightsquigarrow\; \Gamma_0 \vdash \bar{M} : \exists \vec{t_0}{::}\hat{\kappa}_0.\, \Pi\bar{\Gamma}.\, \exists \vec{t_1}{::}\hat{\kappa}_1.\, \bar{\sigma}\vec{\tau}}{\begin{array}{r}\Gamma \vdash_{\mathsf{k}\sqcup\mathsf{D}} (M :: \sigma) : \sigma \;\rightsquigarrow\; \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t_0}, x\rangle.\, \langle \vec{t} = \lambda\bar{\Gamma}.\, \lambda \vec{t_1}{::}\hat{\kappa}_1.\, \vec{\tau}, x : \Pi\bar{\Gamma}.\, \exists \vec{t_1}{::}\hat{\kappa}_1.\, \bar{\sigma}(\vec{t}\bar{\Gamma}\vec{t_1})\rangle \\ : \exists \vec{t_0}{::}\hat{\kappa}_0.\, \exists \vec{t}{::}\bar{\Gamma} \Rightarrow \hat{\kappa}_1 \Rightarrow \hat{\bar{\sigma}}.\, \Pi\bar{\Gamma}.\, \exists \vec{t_1}{::}\hat{\kappa}_1.\, \bar{\sigma}(\vec{t}\bar{\Gamma}\vec{t_1})\end{array}}\;51$$

$$\frac{\Gamma \vdash_{\mathsf{k}} M : \sigma \;\rightsquigarrow\; \Gamma_0 \vdash \bar{M} : \exists \vec{t_0}{::}\hat{\kappa}_0.\, \Pi\bar{\Gamma}.\, \exists \vec{t_1}{::}\hat{\kappa}_1.\, \bar{\sigma}\vec{\tau}}{\begin{array}{r}\Gamma \vdash_{\mathsf{W}} (M :> \sigma) : \sigma \;\rightsquigarrow\; \Gamma_0 \vdash \mathsf{open}\ \bar{M}\ \mathsf{as}\ \langle \vec{t_0}, x\rangle.\, \Lambda\bar{\Gamma}.\, \mathsf{open}\ x\bar{\Gamma}\ \mathsf{as}\ \langle \vec{t_1}, y\rangle.\, \langle \vec{t} = \vec{\tau}, y : \bar{\sigma}\vec{t}\rangle \\ : \exists \vec{t_0}{::}\hat{\kappa}_0.\, \Pi\bar{\Gamma}.\, \exists \vec{t_1}{::}\hat{\kappa}_1.\, \exists\bar{\sigma}\end{array}}\;52$$

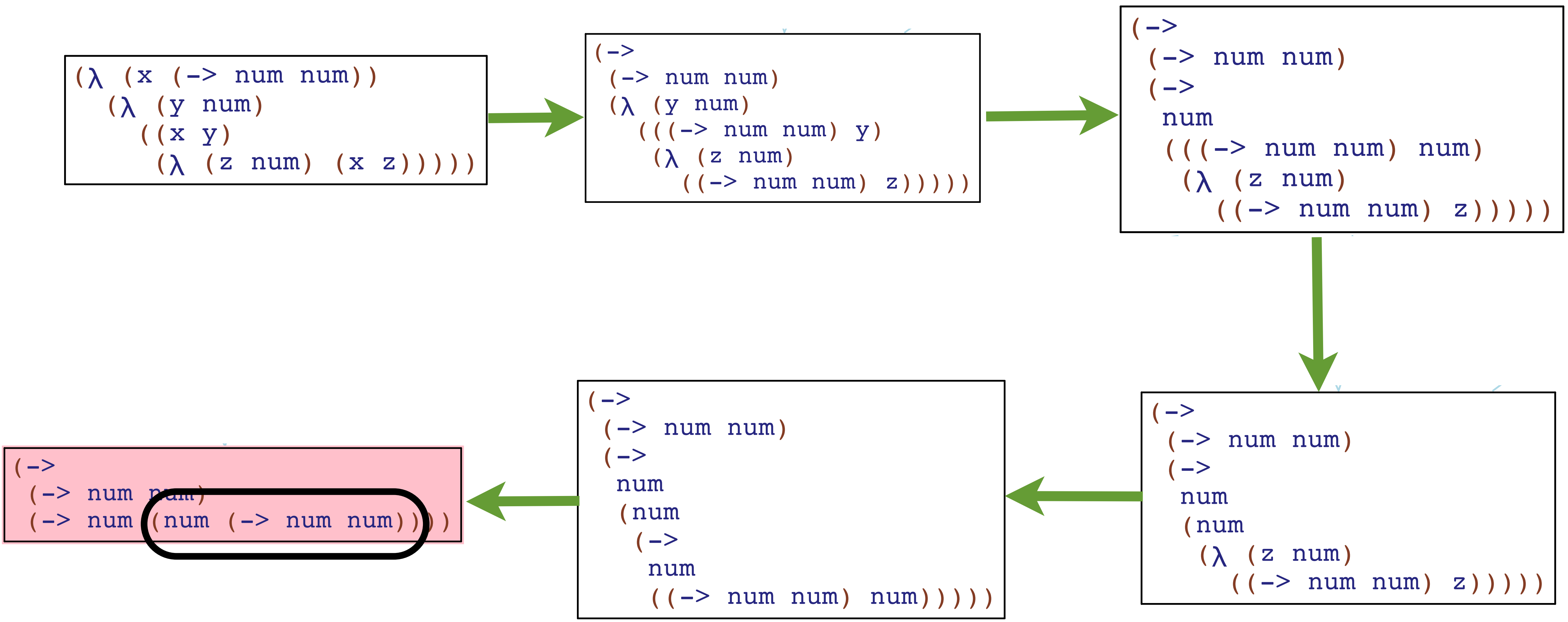
$$\frac{\Gamma \vdash_{\mathsf{P}} M : [\![T]\!] \;\rightsquigarrow\; \Gamma_0 \vdash \bar{M} : \Pi\bar{\Gamma}.\, \mathsf{Ty}\,\bar{\tau}}{\Gamma \vdash_{\mathsf{P}} M : \mathfrak{S}(M) \;\rightsquigarrow\; \Gamma_0 \vdash \bar{M} : \Pi\bar{\Gamma}.\, \mathsf{Ty}\,\bar{\tau}}\;53$$

$$\frac{\Gamma \vdash_{\mathsf{P}} \lambda s{:}\sigma.\, M s : \Pi^{\mathsf{tot}} s{:}\sigma.\, \rho \;\rightsquigarrow\; \Gamma_0 \vdash \bar{M} : \bar{\tau}}{\Gamma \vdash_{\mathsf{P}} M : \Pi^{\mathsf{tot}} s{:}\sigma.\, \rho \;\rightsquigarrow\; \Gamma_0 \vdash \bar{M} : \bar{\tau}}\;54 \qquad \frac{\Gamma \vdash_{\mathsf{P}} \langle s = \pi_1 M, \pi_2 M\rangle : \Sigma s{:}\sigma.\, \rho \;\rightsquigarrow\; \Gamma_0 \vdash \bar{M} : \bar{\tau}}{\Gamma \vdash_{\mathsf{P}} M : \Sigma s{:}\sigma.\, \rho \;\rightsquigarrow\; \Gamma_0 \vdash \bar{M} : \bar{\tau}}\;55$$

Thinking of a Type System

Operationally

Type Checking as a Rewriting System

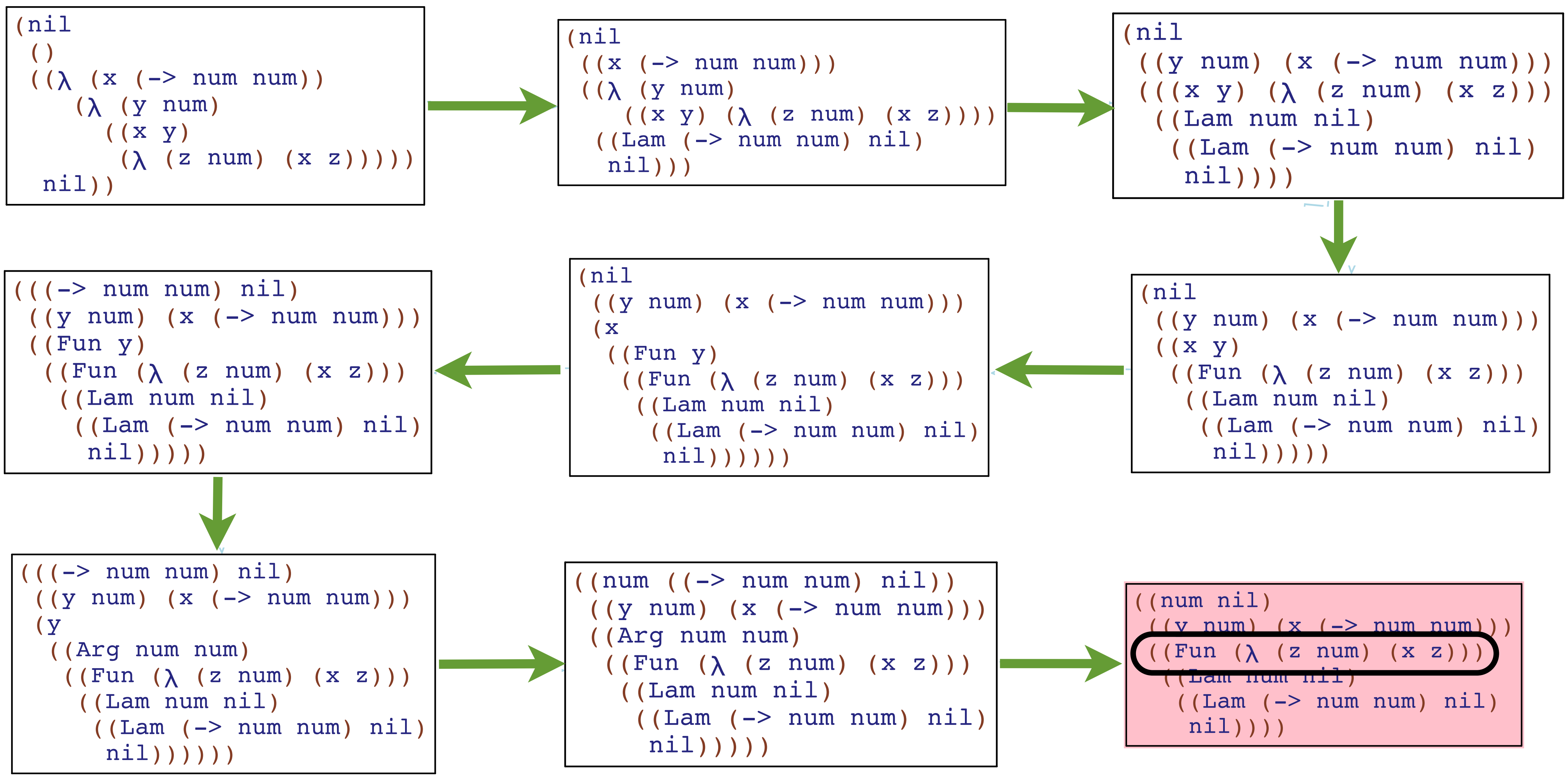


G. Kuan, D. MacQueen, R. B. Findler A Rewriting Semantics for Type Inference, ESOP 07

Tracing Type Error Origin

```
(->
 (-> num num)
 (-> num (num (-> num num))))
```

Type Checking as an Abstract Machine



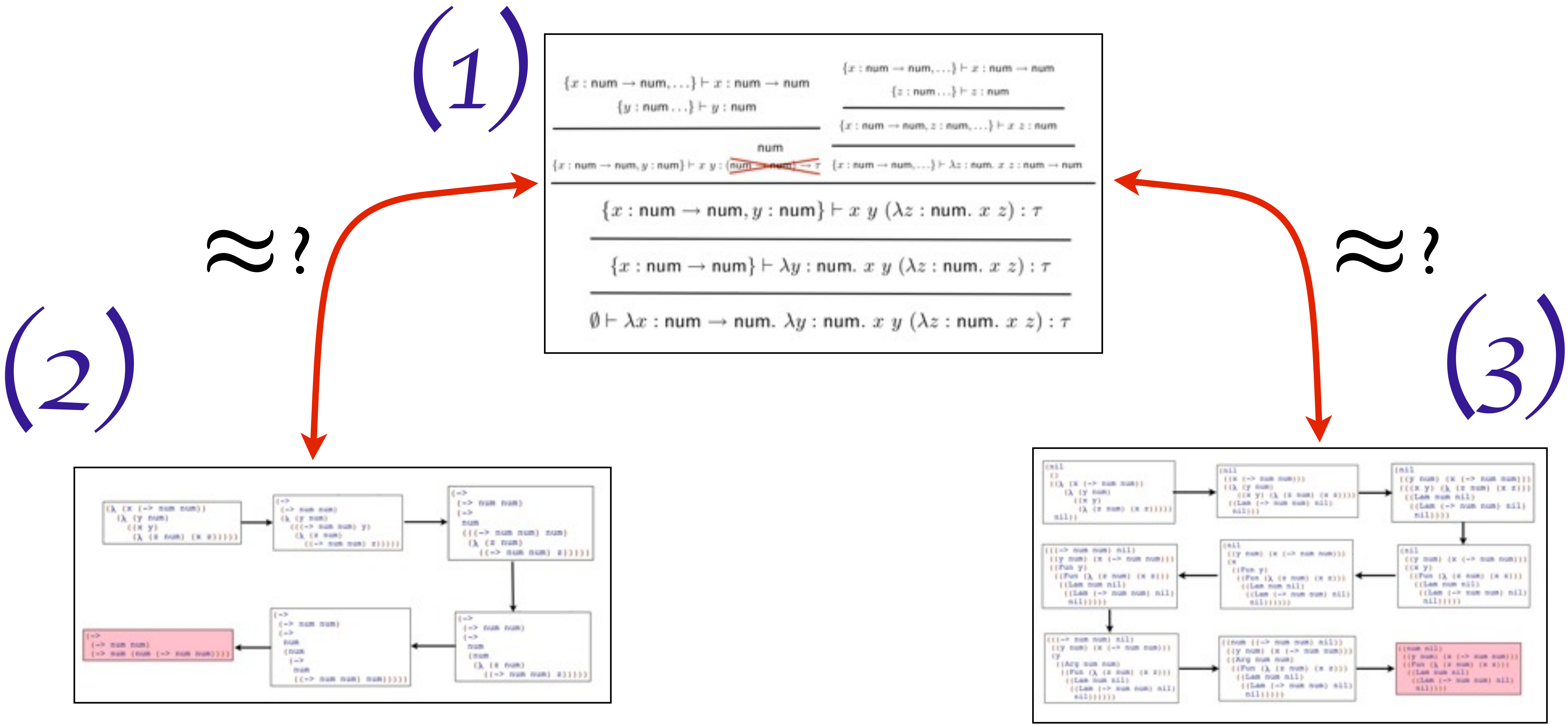
C. Hankin, D. Le Métayer. Deriving Algorithms From Type Inference Systems, POPL 94

Recovering Type Checking Context

```
((num nil)
 ((y num) (x (-> num num)))
 ((Fun (λ (z num) (x z)))
  ((Lam num nil)
   ((Lam (-> num num) nil)
    nil))))
```

The Problem

Equivalence of Type Checking Semantics



A *Hard* Solution

To *prove* soundness and completeness

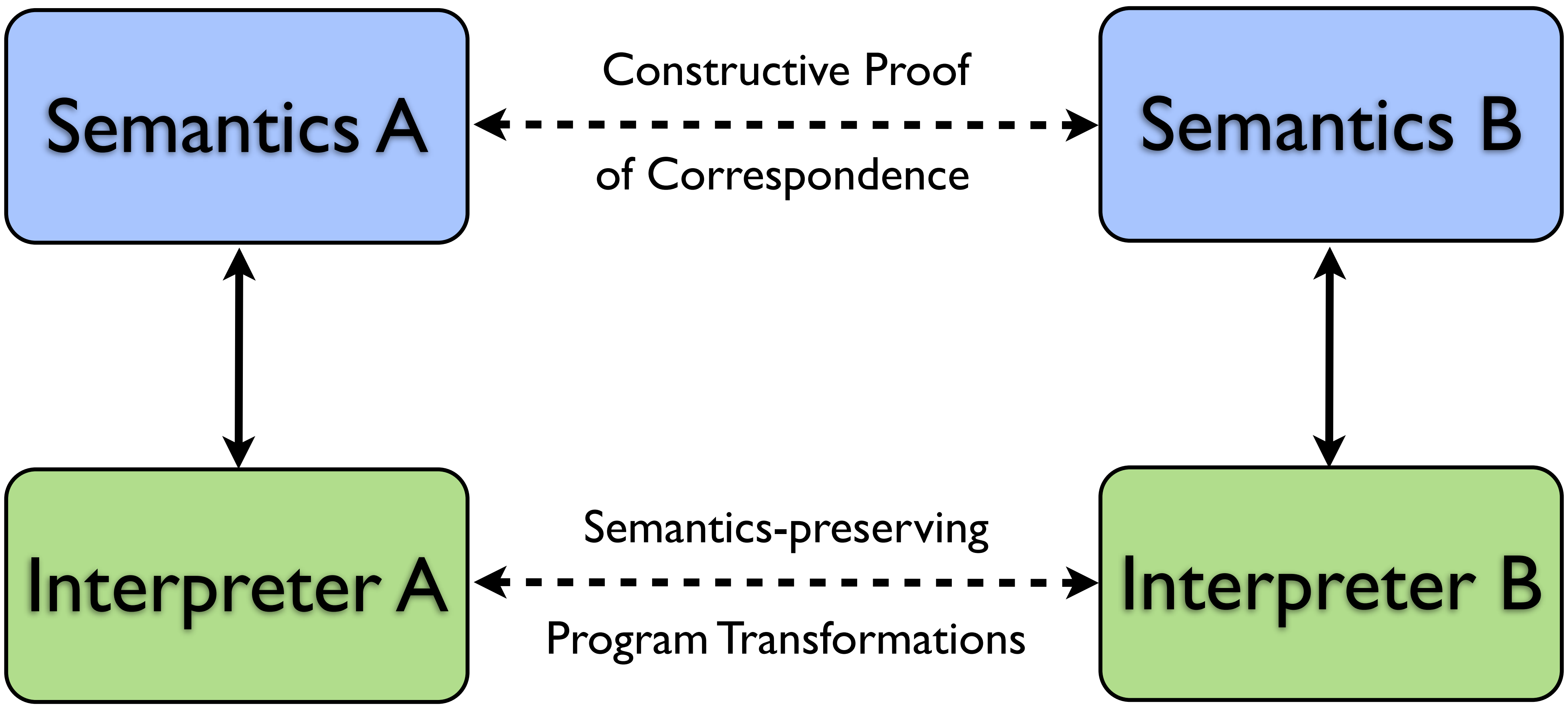
$(1) \approx (2)$ G. Kuan, D. MacQueen, R. B. Findler, ESOP 07

$(1) \approx (3)$ C. Hankin, D. Le Métayer, POPL'94

- Non-reusable, should be proven for each *new* pair of semantics
- Should be done *a posteriori*, after the semantics is constructed

Our Solution

Applying the Functional Correspondence



Example: Semantics of Fibonacci Numbers

$$\text{(fib-1)}\ \frac{}{1 \Downarrow_{fib} 1} \qquad \text{(fib-2)}\ \frac{}{2 \Downarrow_{fib} 1}$$

$$\text{(fib-n)}\ \frac{(n-1) \Downarrow_{fib} v_1 \qquad (n-2) \Downarrow_{fib} v_2}{n \Downarrow_{fib} v_1 + v_2}$$

$$\Updownarrow$$

```
fun fib0 n
  = if n = 1 orelse n = 2 then 1
    else let val v1 = fib0 (n - 1)
             val v2 = fib0 (n - 2)
         in v1 + v2 end
```

Example: Semantics of Fibonacci Numbers

```
fun fib_stack (s: int list, n: int)
  = if n = 1 orelse n = 2 then 1 :: s
    else let val s1 = fib_stack (s, n - 1)
             val s2 = fib_stack (s1, n - 2)
             in case s2 of
                  v1 :: v2 :: s3 => (v1 + v2) :: s3
             end

fun fib1 n = fib_stack (nil, n)
```

Example: Semantics of Fibonacci Numbers

```
fun fib_cps (s, n, k)
  = if n = 1 orelse n = 2 then k (1 :: s)
    else fib_cps (s, n - 1, fn s1 =>
           fib_cps (s1, n - 2, fn s2 =>
             case s2 of
               v1 :: v2 :: s3 => k ((v1 + v2) :: s3)))

fun fib2 n = fib_cps (nil, n, fn (x :: _) => x )
```

Example: Semantics of Fibonacci Numbers

```
datatype cont = CONT_MT
              | CONT_FIB1 of int * cont
              | CONT_FIB2 of cont

fun fib_defun (s, n, C)
  = if n = 1 orelse n = 2 then continue (1 :: s, C)
    else fib_defun (s, n - 1, CONT_FIB1 (n, C) )

and continue (s, CONT_MT )
    = (case s of (x :: _) => x)
  | continue (s, CONT_FIB1 (n, C) )
    = fib_defun (s, n - 2, CONT_FIB2 C )
  | continue (s, CONT_FIB2 C )
    = case s of (v1 :: v2 :: s3) => continue ((v1 + v2) :: s3, C)

fun fib3 n = fib_defun (nil, n, CONT_MT )
```

Example: Semantics of Fibonacci Numbers

```
datatype cont' = CONT_MT'
               | CONT_FIB1' of int * cont'
               | CONT_FIB2' of cont'
               | NUM' of int * cont'

fun fib_defun' (s, NUM' (n, C) )
  = if n = 1 orelse n = 2 then continue1 (1 :: s, C)
    else fib_defun' (s, NUM' (n - 1, CONT_FIB1' (n, C)))

and continue1 (s, CONT_MT' )
    = (case s of (x :: _) => x)
  | continue1 (s, CONT_FIB1' (n, C) )
    = fib_defun' (s, NUM' (n - 2, CONT_FIB2' C))
  | continue1 (s, CONT_FIB2' C )
    = case s of (v1 :: v2 :: s3) => continue1 ((v1 + v2) :: s3, C)

fun fib4 n = fib_defun' (nil, NUM' (n, CONT_MT'))
```

Example: Semantics of Fibonacci Numbers

```
datatype control_element = NUM of int
                         | CF1 of int
                         | CF2

fun fib_control (s, NUM n :: C)
  = if n = 1 orelse n = 2 then fib_control (1 :: s, C)
    else fib_control(s, NUM (n - 1) :: CF1 n :: C)
  | fib_control (s, CF1 n :: C)
    = fib_control (s, NUM (n - 2) :: CF2 :: C)
  | fib_control (s, CF2 :: C)
    = (case s of (v1 :: v2 :: s3) => fib_control ((v1 + v2) :: s3, C))
  | fib_control (s, nil)
    = (case s of (x :: _) => x)

fun fib5 n = fib_control (nil, NUM n :: nil)
```

Example: Semantics of Fibonacci Numbers

$$
\begin{array}{rcl}
\langle S,\ Num(1) :: C\rangle & \Rightarrow_{\mathrm{SC}_{fib}} & \langle 1 :: S,\ C\rangle \\
\langle S,\ Num(2) :: C\rangle & \Rightarrow_{\mathrm{SC}_{fib}} & \langle 1 :: S,\ C\rangle \\
\langle S,\ Num(n) :: C\rangle & \Rightarrow_{\mathrm{SC}_{fib}} & \langle S,\ Num(n-1) :: CF_1(n) :: C\rangle \\
\langle S,\ CF_1(n) :: C\rangle & \Rightarrow_{\mathrm{SC}_{fib}} & \langle S,\ Num(n-2) :: CF_2 :: C\rangle \\
\langle v_1 :: v_2 :: S,\ CF_2 :: C\rangle & \Rightarrow_{\mathrm{SC}_{fib}} & \langle (v_1 + v_2) :: S,\ C\rangle
\end{array}
$$

$$\Updownarrow$$

```
type state = int list * control_element list

(*  step : state -> state  *)
fun step ( s, NUM 1 ::  C )
     = (1 :: s, C)
  | step ( s, NUM 2 ::  C )
     = (1 :: s, C)
  | step ( s, NUM n ::  C )
     = (s, NUM (n - 1) :: CF1 n :: C)
  | step (s,  CF1 n ::  C )
     = (s, NUM (n - 2) :: CF2 :: C)
  | step ( v1 ::  v2 ::  s3, CF2 ::  C )
     = ((v1 + v2) :: s3,  C)

(*  step : state -> int  *)
fun iterate (v :: _, nil)
     = v
  | iterate (s, C)
     = iterate (step (s, C))
```

Functional Correspondence, applied

- Evaluators with computational effects [Ager-al:TCS05]
- Object calculi inter-derivation [Danvy-Johannsen:JCSS10]
- Landin's SECD machine [Danvy-Millikin:LMCS08]
- Abstract machine for call-by-need lambda calculus [Ager-al:IPL04, Danvy-al:FLOPS10]
- Formalizing semantics of Scheme [Biernacka-Danvy:LNCS5700]
- Abstract Interpretation-based analyses [VanHorn-Might:ICFP10]
- ...

Operational Aspects of Type Systems

Inter-Derivable Semantics of Type Checking

and

Gradual Types for Object Ownership

Based on the Publications

- Ilya Sergey and Dave Clarke.
  A correspondence between type checking via reduction and type checking via evaluation
  Information Processing Letters, January 2012. Elsevier.

- Ilya Sergey and Dave Clarke.
  A correspondence between type checking via reduction and type checking via evaluation
  Accompanying code overview
  CW Reports, volume CW617. KU Leuven. January 2012.

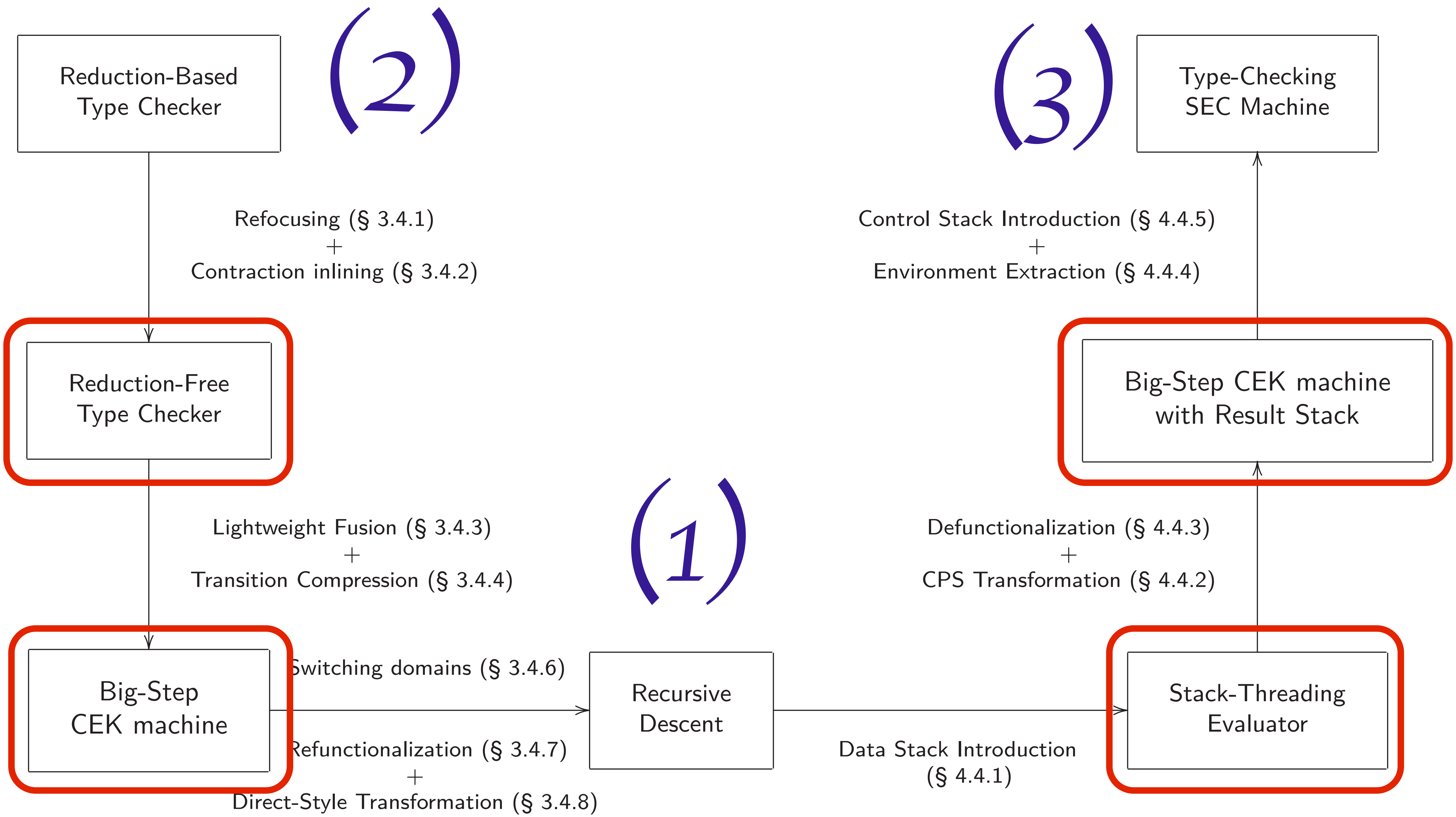
- Ilya Sergey and Dave Clarke.
  From type checking by recursive descent to type checking with an abstract machine
  In proceedings of the 11th Workshop on Language Descriptions, Tools and Applications (LDTA 2011), March 2011. ACM.

A part of this work was carried out while visiting
the BRICS PhD School of Aarhus University in September 2010.

Employed Program Transformations

- CPS Transformation
- Direct-style transformation
- Defunctionalization
- Refunctionalization
- Transition compression
- Lightweight Fusion
- Lambda Lifting
- Closure Conversion
- Control Stack Extraction
- Refocusing

The Resulting Derivation



Summary

1. Type checking is a computation over a program's syntax;
   its semantics may be described in different ways;
2. *Different* formalisms and corresponding implementations might be used,
   but *equivalence* between them should be proved;
3. *Functional correspondence* makes it possible to derive a family of
   algorithms for type checking, rather than invent them from scratch;
4. A tool-chain of program transformations is applied
   to derive those algorithms;
5. All derived semantics correspond to each other *by construction*.

Contributions I

1. A *mechanical* correspondence between type checking via <u>reductions</u> and type checking via <u>evaluation</u>

2. A *mechanical* correspondence between type checking via <u>evaluation</u> and type checking via an <u>abstract machine</u>

3. A family of *novel, semantically equivalent* artifacts for type checking

4. A proof-of-concept implementation of the derivation in *Standard ML* and *PLT Redex*, available at

   `http://github.com/ilyasergey/typechecker-transformations`

Applications

1. Type debugging
   - *Figuring out what has gone wrong during type checking*
2. Incremental type checking
   - *Since a type checker is just an interpreter, the usual memoization techniques can be applied*
3. Conservative type checking via abstract interpretation
   - *Can be applied for effect inference systems, e.g., strictness analysis in the form of a type system*

Future Work I

1. Handling type system evolution
   - *Transformations should not be re-done again*
2. Tool support for transformations
   - *The transformations should be automated*
3. Mechanization of the metatheory
   - *So far, done only for some of the transformations from the toolchain*

Type Systems

Well-Typed Programs Don't Go Wrong

$$\Gamma, \Delta \vdash \varsigma_0 : \tau$$

$$\varsigma_{\mathit{0}} \rightarrow \varsigma_{\mathit{1}} \rightarrow \varsigma_{\mathit{2}} \rightarrow \ldots \rightarrow \varsigma_{n} \rightarrow \varsigma_{\mathit{final}}$$

$$\varsigma_{\mathit{0}} \rightarrow \varsigma_{\mathit{1}} \rightarrow \varsigma_{\mathit{2}} \rightarrow \ldots \rightarrow \varsigma_{n} \rightarrow \ldots$$

Domain-Specific Type Systems

Well-Typed Programs Still Don't Go Wrong

$$\widehat{\Gamma}, \widehat{\Delta} \cancel{\vdash} \varsigma_0 : \widehat{\tau}$$

$$\varsigma_0 \rightarrow \varsigma_1 \rightarrow \varsigma_2$$

$$\rightarrow \varsigma_n \rightarrow \varsigma_{final}$$

Some Domain-Specific Type Systems

- NonNull Types [Fändrich-Leino:OOPSLA03]
- Types for Information Flow Control [Myers:POPL99, Hunt:POPL06]
- Uniqueness Type Systems [Aldrich-al:OOPSLA02, Boyland:SPE01]
- Universe Types [Cunningham-al:FMCO07]
- Ownership Types [Clarke-al:OOPSLA98]

The Problem

A program should not run,
when something is actually *Wrong*.

but

A program should be executable,
even if it might possibly go *Wrong*.

A Solution

Gradual Domain-Specific Type Systems

Inspired by Gradual Types of J. Siek, W. Taha.

Gradual Domain-Specific Type Systems

$$\widehat{\Gamma}, \widehat{\Delta} \not\vdash \varsigma_0 : \widehat{\widetilde{\tau}}$$

$$\varsigma_0 \rightarrow \varsigma_1 \rightarrow \varsigma_2 \rightarrow \ldots \rightarrow \boxed{\varsigma_n} \rightarrow$$ 

Next one is a bad state

Gradual Domain-Specific Type Systems

$$\widehat{\Gamma}, \widehat{\Delta} \cancel{\vdash} \varsigma_0 : \widetilde{\tau}$$

This Work

A Case Study

Making
a Domain-Specific Type System
Gradual

Ownership Types

- data-race freedom [Boyapati-Rinard:OOPSLA01]
- disjointness of effects [Clarke-Drossopoulou:OOPSLA02]
- various confinement properties [Vitek-Bokowski:OOPSLA99]
- modular reasoning about aliasing [Müller:VSTTE05]
- effective memory management [Boyapati-et-al:PLDI03]

But also

- Verbosity of ownership types is a problem
  for practical adaptation
- Sometimes, the imposed invariant is too restrictive
- A <u>type</u> debugging support would require
  to trace the *execution* of <u>programs</u>

Operational Aspects of Type Systems

Inter-Derivable Semantics of Type Checking

and

Gradual Types for Object Ownership

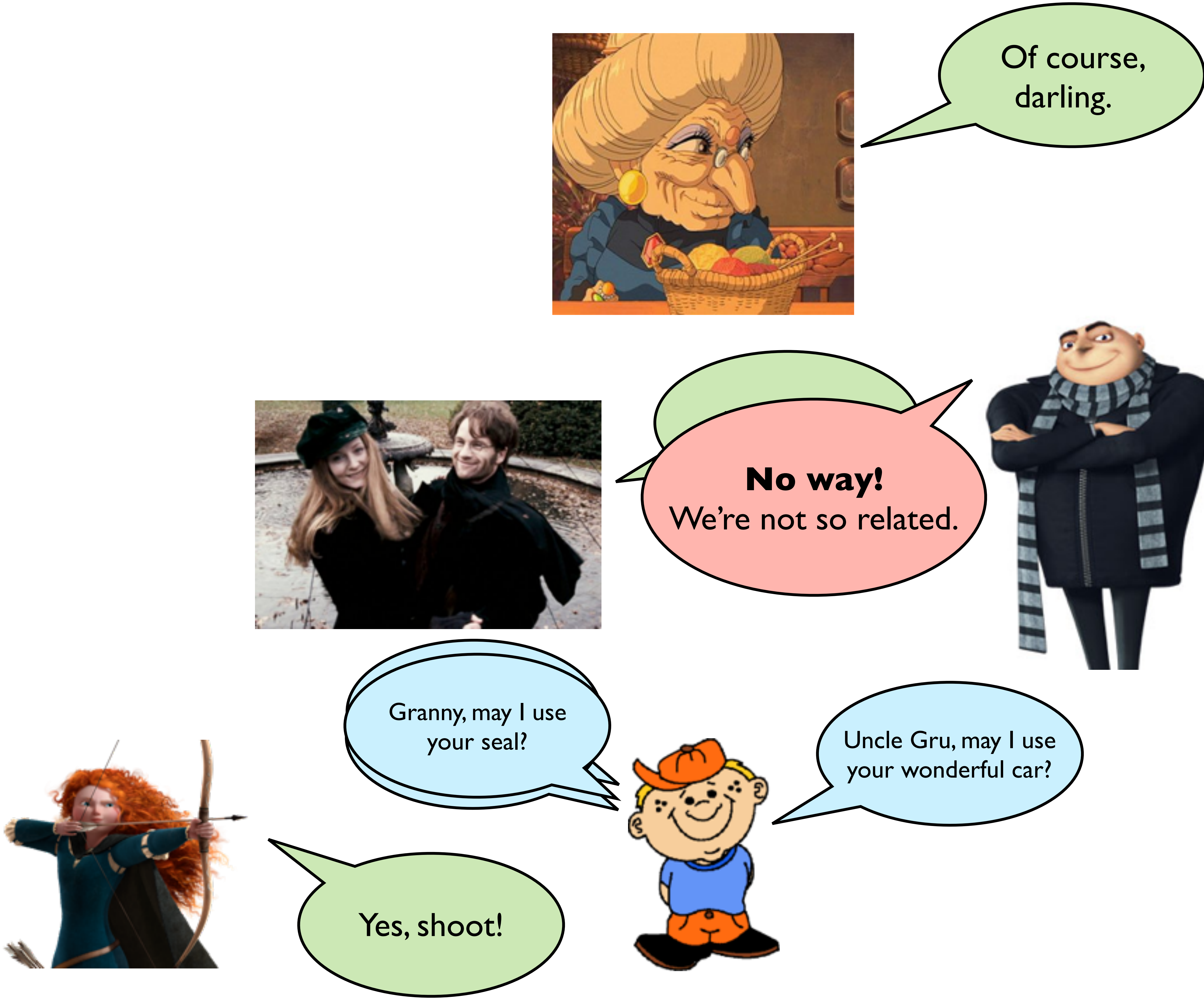
Based on the Publications

- Ilya Sergey and Dave Clarke
  Gradual Ownership Types
  In proceedings of the 21th European Symposium on Programming (ESOP 2012), April 2012. Volume 7211 of LNCS, Springer.

- Ilya Sergey and Dave Clarke
  Gradual Ownership Types, the Accompanying Technical Report
  CW Reports, volume CW613. KU Leuven. December 2011.

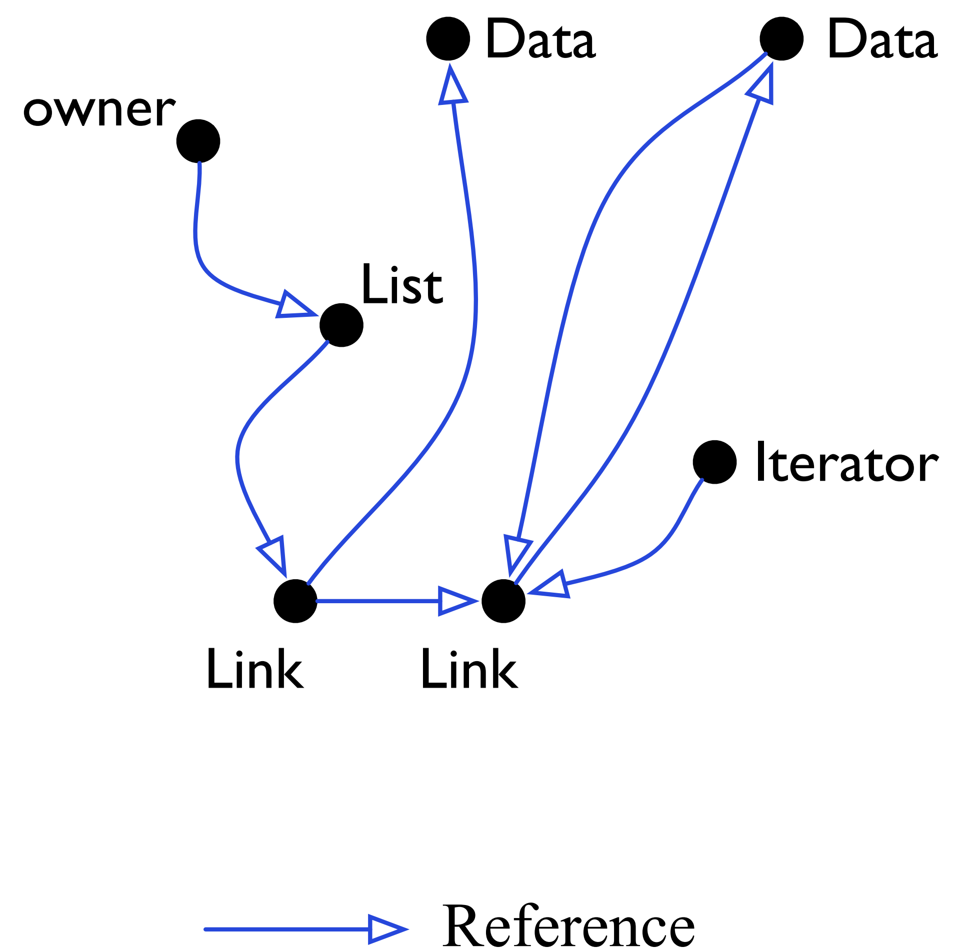
- Ilya Sergey and Dave Clarke
  Towards Gradual Ownership Types
  In International Workshop on Aliasing, Confinement and Ownership (IWACO 2011). July 2011.

Ownership


Of course, darling.
No way! We're not so related.
Granny, may I use your seal?
Uncle Gru, may I use your wonderful car?
Yes, shoot!

Ownership Types*
(a bit more formally)

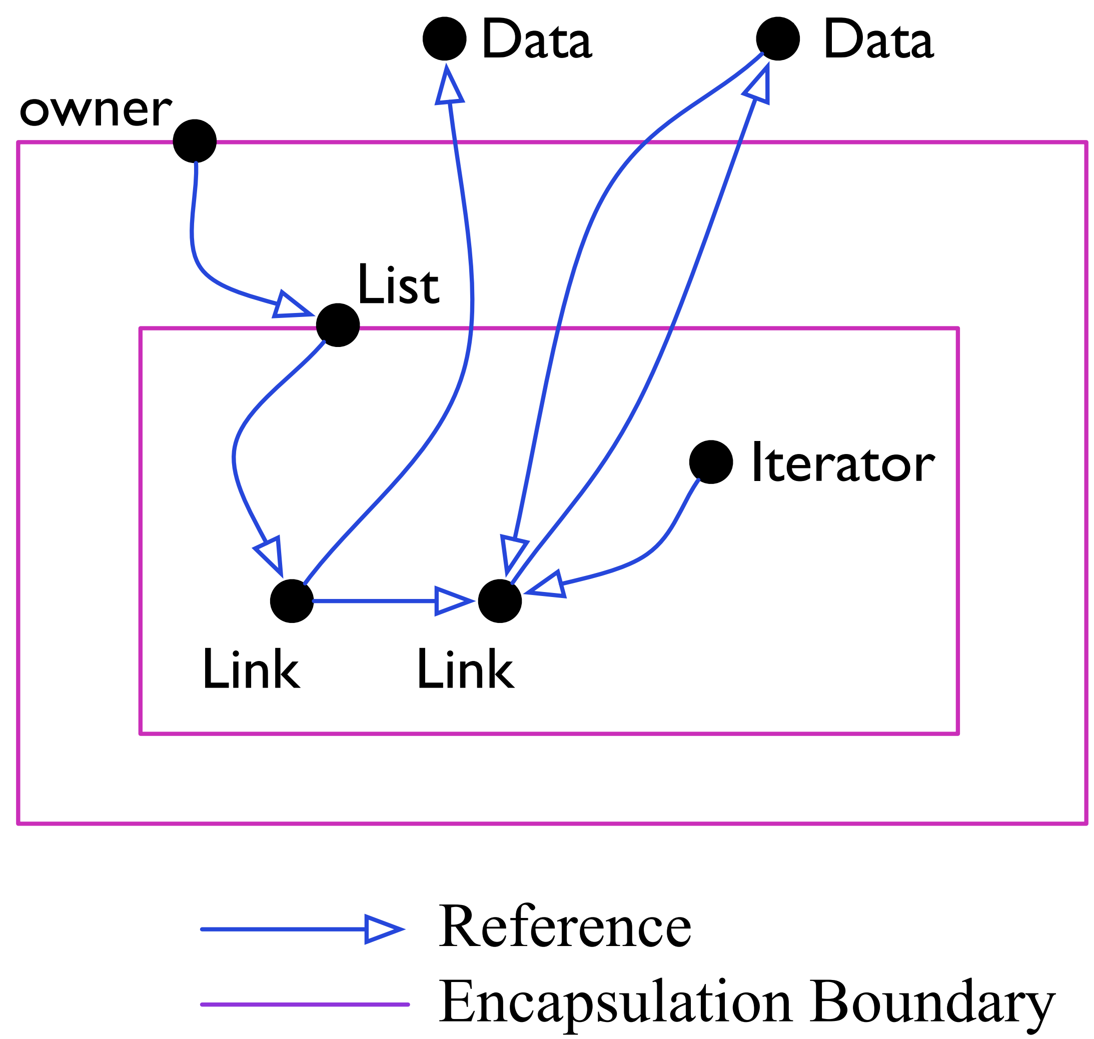
```
class List {
  Link head;
  void add(Data d) {
    head = new Link(head, d);
  }
  Iterator makeIterator() {
    return new Iterator(head);
  }
}
class Link {
  Link next;
  Data data;
  Link(Link next, Data data) {
    this.next = next; this.data = data;
  }
}
class Iterator {
  Link current;
  Iterator(Link first) {
    current = first;
  }
  void next() { current = current.next; }
  Data elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```



Clarke, Noble, Potter, OOPSLA '98

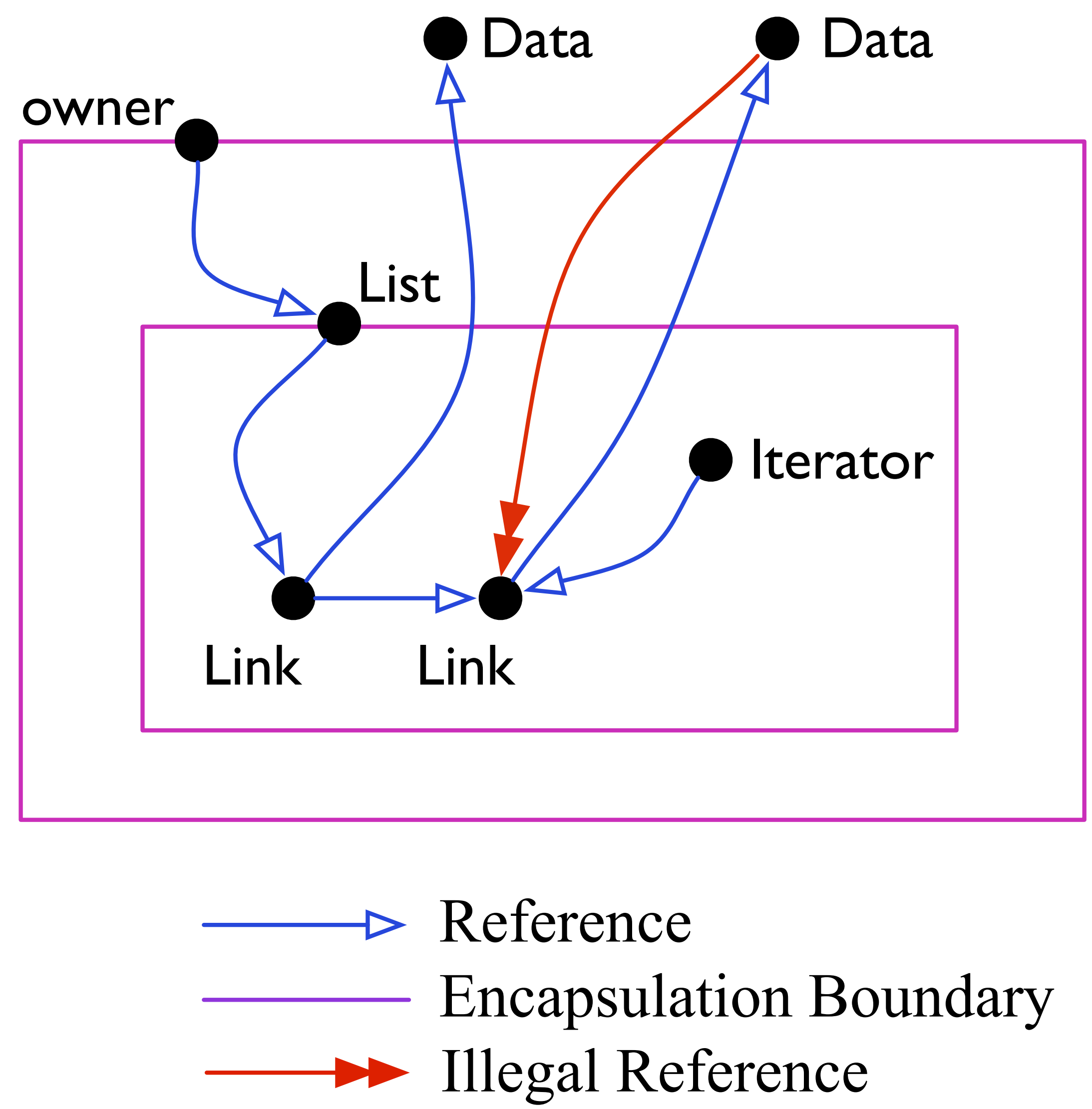
Ownership Types

```
class List {
  Link head;
  void add(Data d) {
    head = new Link(head, d);
  }
  Iterator makeIterator() {
    return new Iterator(head);
  }
}
class Link {
  Link next;
  Data data;
  Link(Link next, Data data) {
    this.next = next; this.data = data;
  }
}
class Iterator {
  Link current;
  Iterator(Link first) {
    current = first;
  }
  void next() { current = current.next; }
  Data elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```



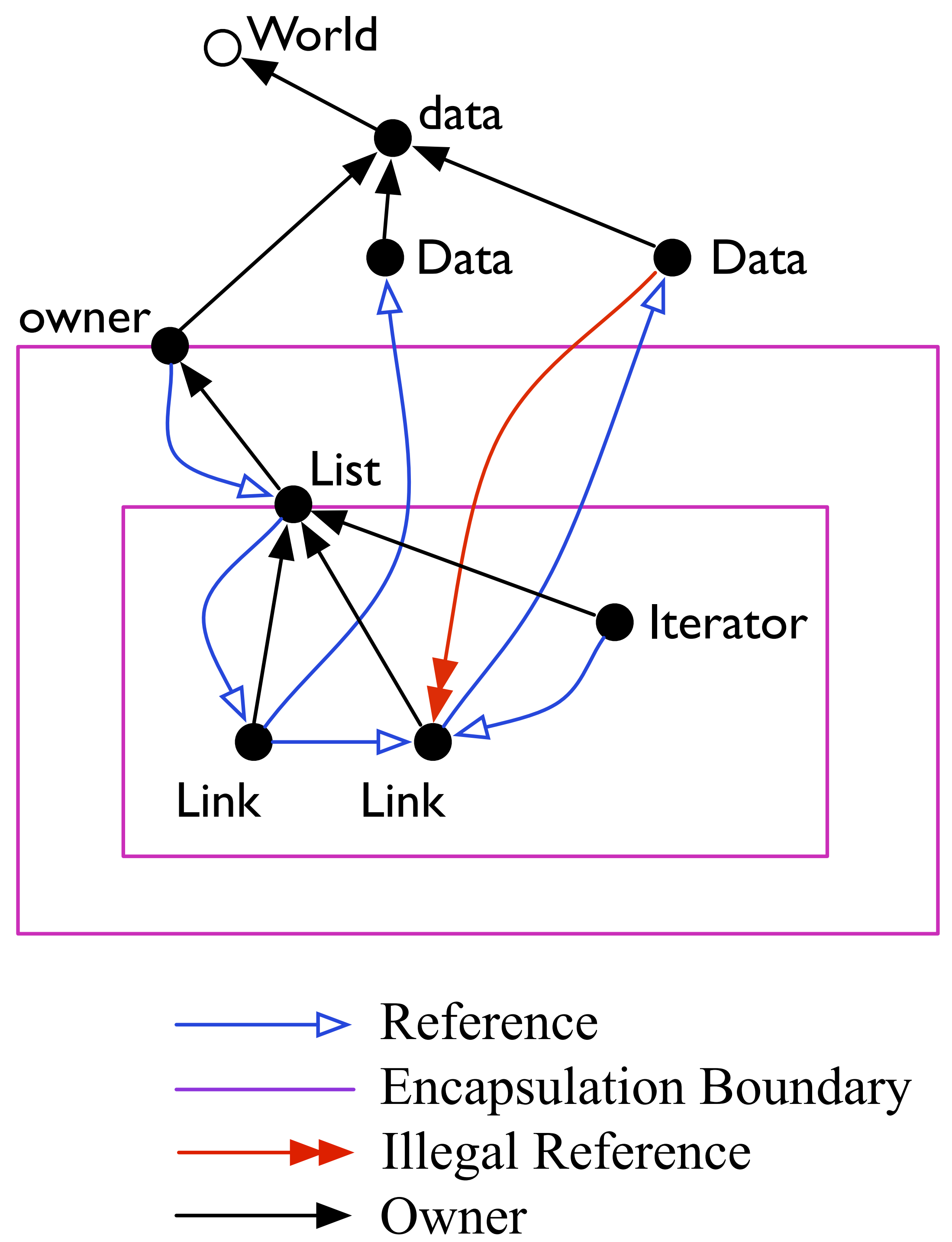
Ownership Types

```
class List {
  Link head;
  void add(Data d) {
    head = new Link(head, d);
  }
  Iterator makeIterator() {
    return new Iterator(head);
  }
}
class Link {
  Link next;
  Data data;
  Link(Link next, Data data) {
    this.next = next; this.data = data;
  }
}
class Iterator {
  Link current;
  Iterator(Link first) {
    current = first;
  }
  void next() { current = current.next; }
  Data elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```



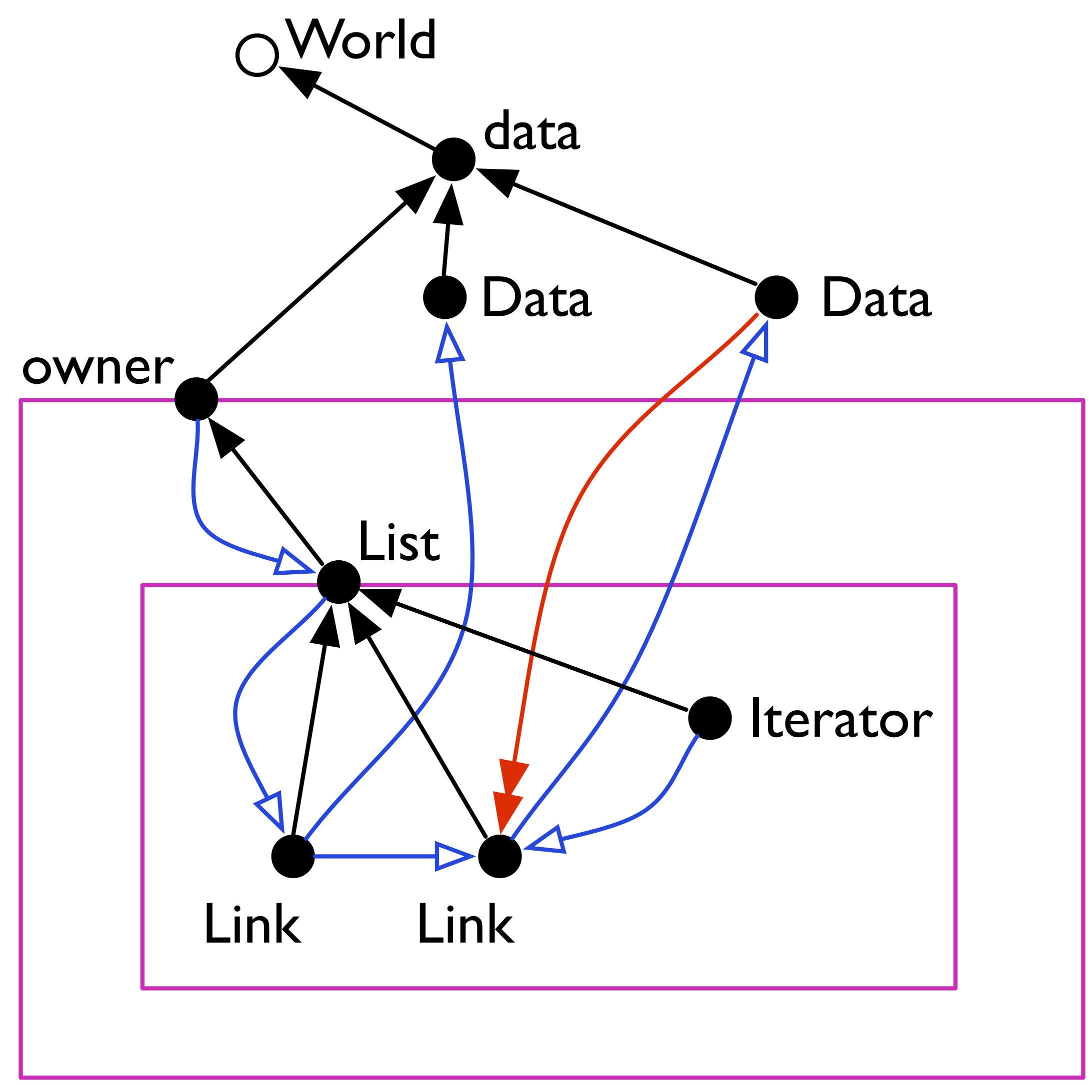
Ownership Types

```
class List {
  Link head;
  void add(Data d) {
    head = new Link(head, d);
  }
  Iterator makeIterator() {
    return new Iterator(head);
  }
}
class Link {
  Link next;
  Data data;
  Link(Link next, Data data) {
    this.next = next; this.data = data;
  }
}
class Iterator {
  Link current;
  Iterator(Link first) {
    current = first;
  }
  void next() { current = current.next; }
  Data elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```



Ownership Types

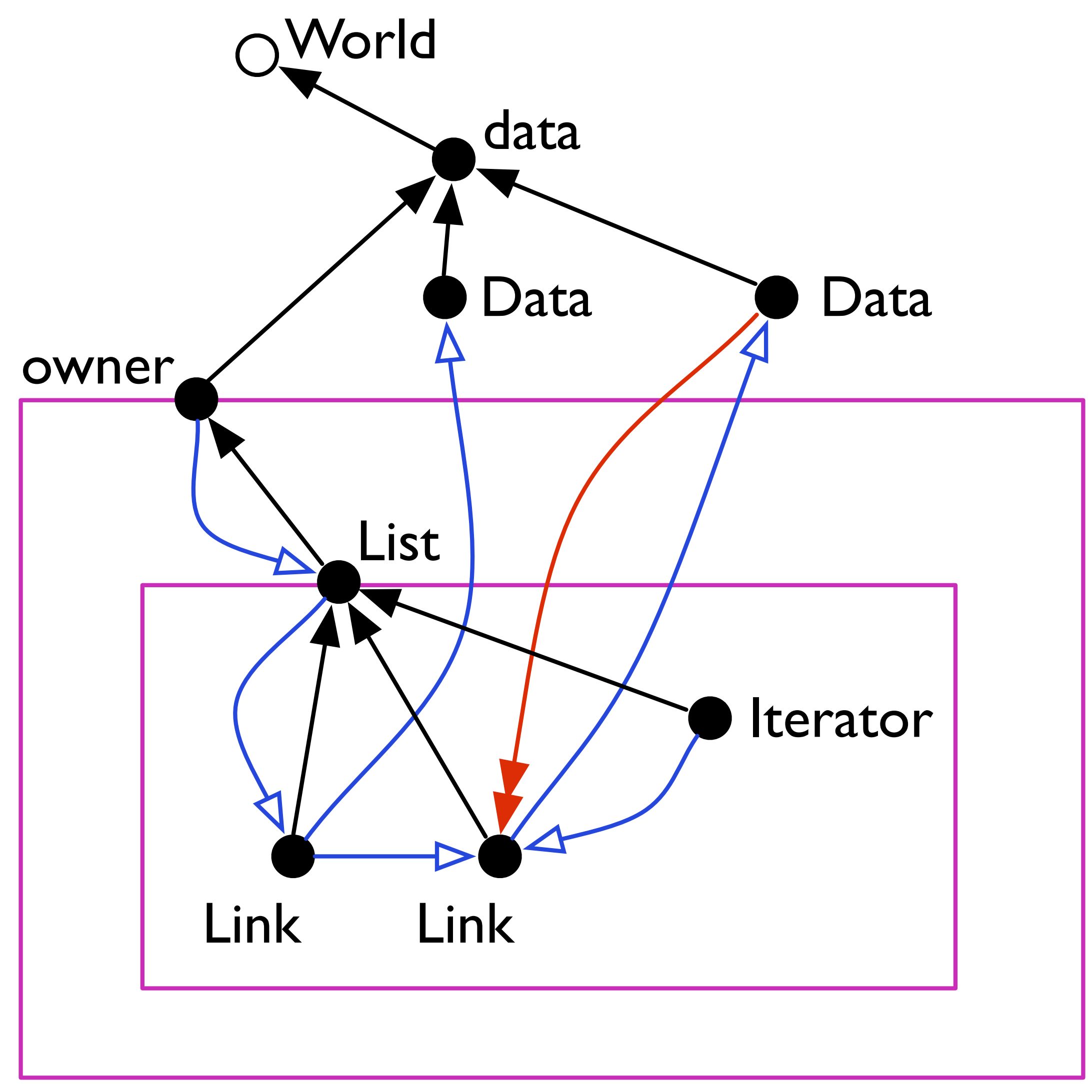
```
class List {
  Link head;
  void add(Data d) {
    head = new Link(head, d);
  }
  Iterator makeIterator() {
    return new Iterator(head);
  }
}
class Link {
  Link next;
  Data data;
  Link(Link next, Data data) {
    this.next = next; this.data = data;
  }
}
class Iterator {
  Link current;
  Iterator(Link first) {
    current = first;
  }
  void next() { current = current.next; }
  Data elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```



Owners-as-Dominators (OAD)

Ownership Types

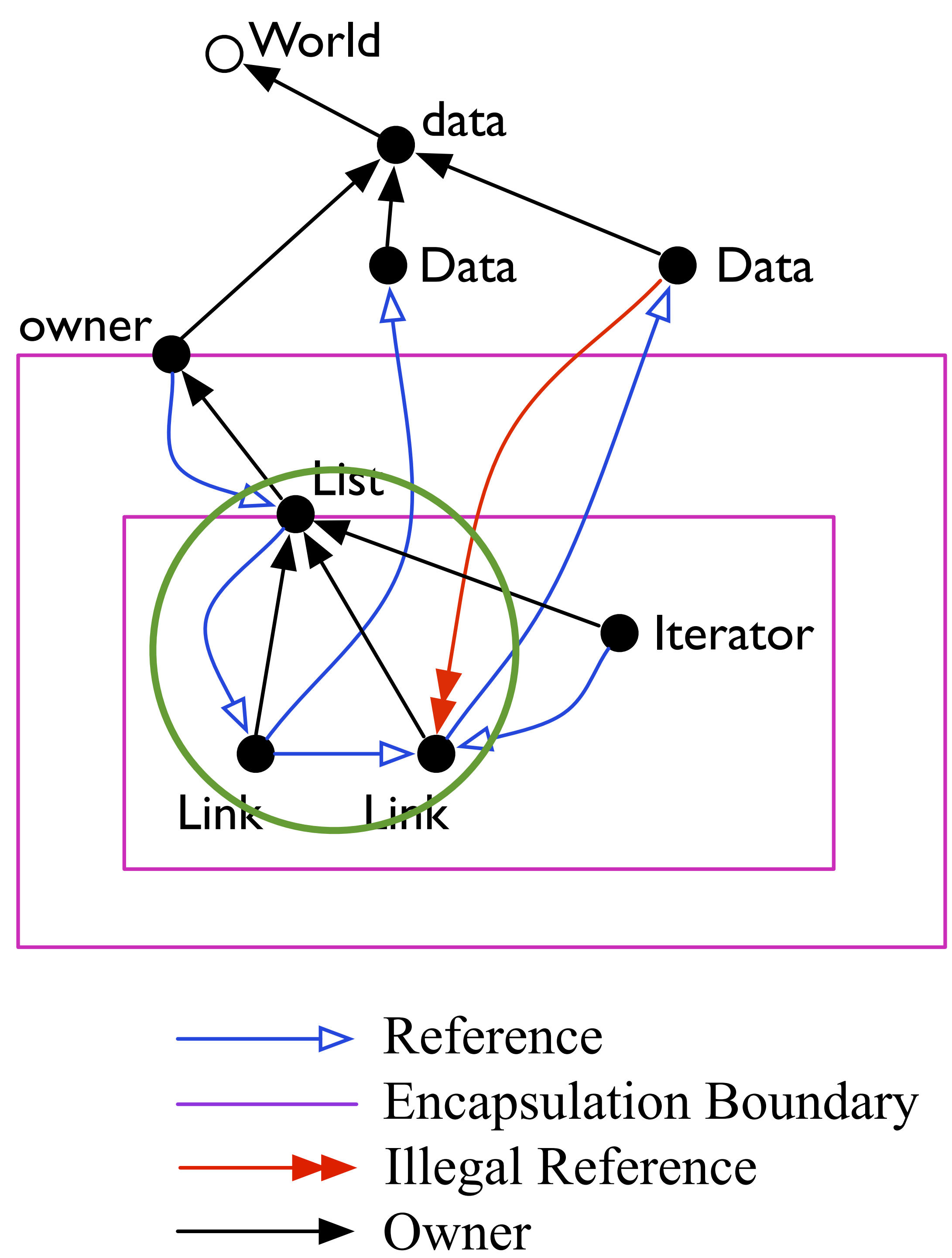
```
class List<owner, data> {
  Link head<this, data>;
  void add(Data<data> d) {
    head = new Link<this, data>(head, d);
  }
  Iterator<this, data> makeIterator() {
    return new Iterator<this, data>(head);
  }
}
class Link<owner, data> {
  Link<owner, data> next;
  Data<data> data;
  Link(Link<owner, data> next, Data<data> data) {
    this.next = next; this.data = data;
  }
}
class Iterator<owner, data> {
  Link<owner, data> current;
  Iterator(Link<owner, data> first) {
    current = first;
  }
  void next() { current = current.next; }
  Data<data> elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```



Owners-as-Dominators (OAD)

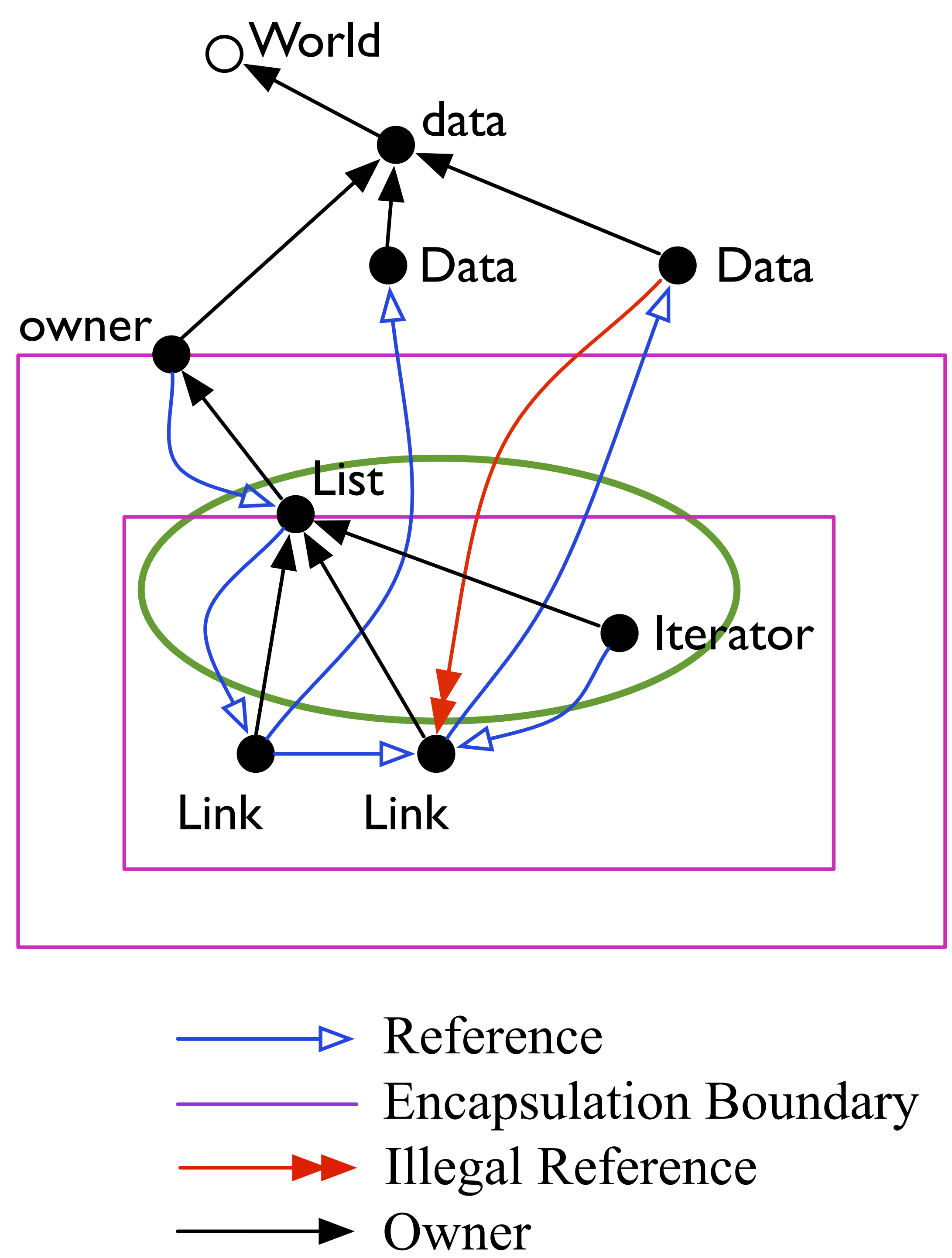
The Essence of Ownership Types

```
class List<owner, data> {
  Link head<this, data>;
  void add(Data<data> d) {
    head = new Link<this, data>(head, d);
  }
  Iterator<this, data> makeIterator() {
    return new Iterator<this, data>(head);
  }
}
class Link<owner, data> {
  Link<owner, data> next;
  Data<data> data;
  Link(Link<owner, data> next, Data<data> data) {
    this.next = next; this.data = data;
  }
}
class Iterator<owner, data> {
  Link<owner, data> current;
  Iterator(Link<owner, data> first) {
    current = first;
  }
  void next() { current = current.next; }
  Data<data> elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```



The Essence of Ownership Types

```
class List<owner, data> {
  Link head<this, data>;
  void add(Data<data> d) {
    head = new Link<this, data>(head, d);
  }
  Iterator<this, data> makeIterator() {
    return new Iterator<this, data>(head);
  }
}
class Link<owner, data> {
  Link<owner, data> next;
  Data<data> data;
  Link(Link<owner, data> next, Data<data> data) {
    this.next = next; this.data = data;
  }
}
class Iterator<owner, data> {
  Link<owner, data> current;
  Iterator(Link<owner, data> first) {
    current = first;
  }
  void next() { current = current.next; }
  Data<data> elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```

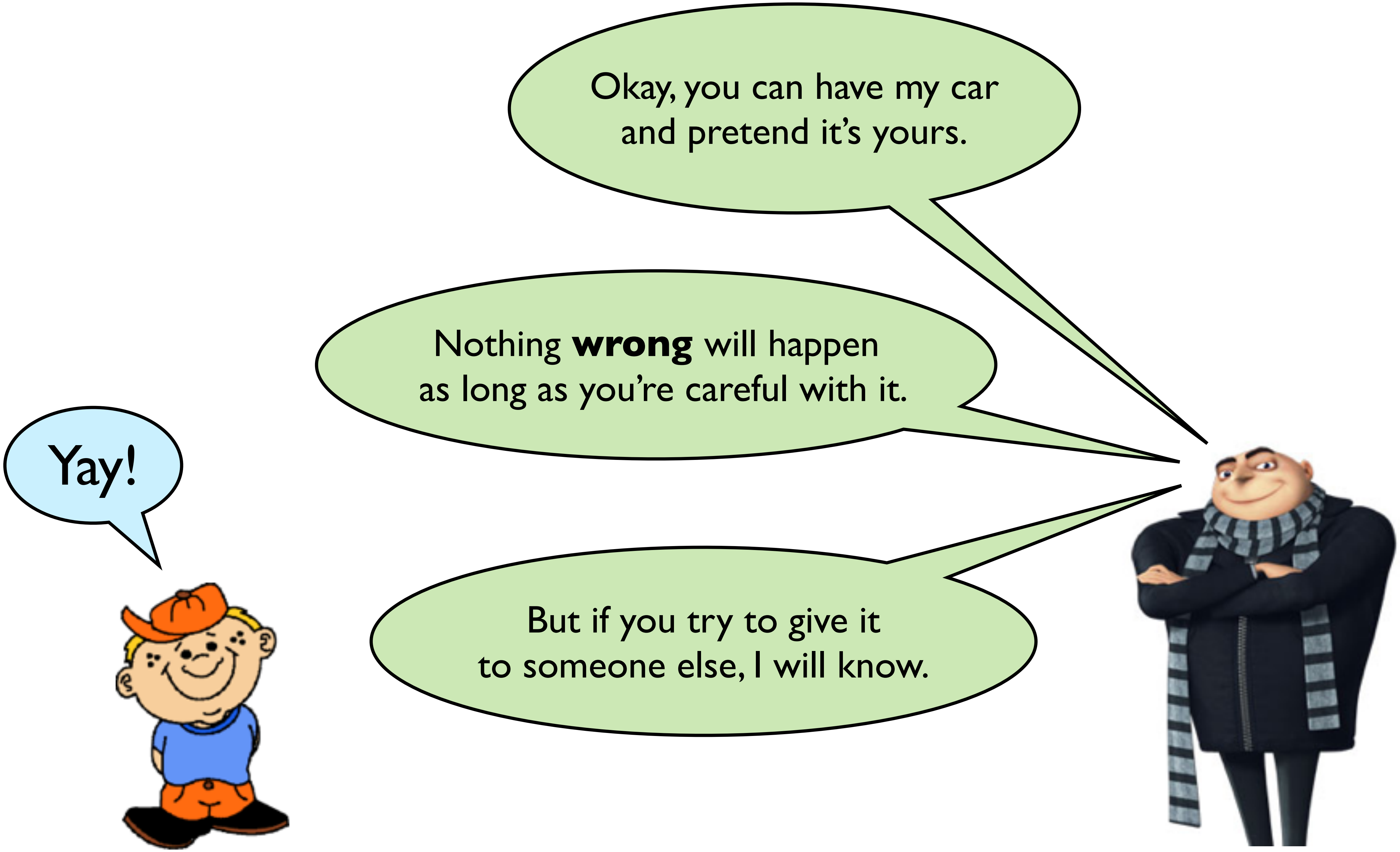


Can we implement
the same intention with a
fewer amount of annotations?

The Essence of Gradual Types

- Programmers may omit type annotations and run the program immediately
  - Run-time checks are *inserted* to ensure *type safety*
- Programmers may add type annotations to increase static checking
  - When all sites are annotated,
    all type errors are caught at compile-time

Gradual Ownership


Okay, you can have my car and pretend it's yours.
Nothing **wrong** will happen as long as you're careful with it.
But if you try to give it to someone else, I will know.
Yay!

Gradual Ownership Types

A syntactic type parametrized with owners:

```
Car<Gru, Dad_Of_Gru>
```

Some owners *might* be *unknown:*

```
Car<?, Dad_Of_Gru>
```

Or even all of them:

```
Car ≡ Car<?, ?>
```

Type equality: types T_1 and T_2 are *equal*:

```
C<owner, outer> = C<owner, outer>
```

Type equality: types T_1 and T_2 are *consistent*

```
C<owner, ?> ~ C<?, outer>
```

T_1 and T_2 *might* correspond
to the *same* runtime values

Traditional Subtyping

```
class D<MyOwner> {...}

class C<Owner1, Owner2> extends D<Owner1> {...}
```

Subtyping: T_1 is a *subtype* of T_2

`C<owner, outer> ≤ D<owner>`

$$\boxed{E;B \vdash t \leq t'}$$

$$\text{(SUB-REFL)}\quad \frac{E;B \vdash t}{E;B \vdash t \leq t}$$

Reflexive

$$\text{(SUB-TRANS)}\quad \frac{E;B \vdash t \leq t' \quad E;B \vdash t' \leq t''}{E;B \vdash t \leq t''}$$

Transitive

$$\text{(SUB-CLASS)}\quad \frac{\begin{array}{c} E;B \vdash c\langle\sigma\rangle \\ \texttt{class}\ c\langle\alpha_{i\in 1..n}\rangle\ \texttt{extends}\ c'\langle r_{i\in 1..n'}\rangle\{\ldots\} \end{array}}{E;B \vdash c\langle\sigma\rangle \leq c'\langle\sigma(r_i)_{i\in 1..n'}\rangle}$$

Nominal

Gradual Subtyping

```
class D<MyOwner> {...}

class C<Owner1, Owner2> extends D<Owner1> {...}
```

$$
\begin{tikzcd}
\texttt{D<?>} \arrow[r, "\sim"] & \texttt{D<owner>} \\
\texttt{C<?, outer>} \arrow[u, "\leq"] \arrow[ur, "\lesssim"'] &
\end{tikzcd}
$$

$$\texttt{C<?, outer>} \lesssim \texttt{D<owner>}$$

Gradual Ownership Type System

$\boxed{E;B \vdash p \sim p'}$

Consistent owners

$$\frac{E;B \vdash p}{E;B \vdash p \sim p}\ (\text{CON-REFL}) \qquad \frac{E;B \vdash p}{E;B \vdash\ ? \sim p}\ (\text{CON-RIGHT}) \qquad \frac{E;B \vdash p}{E;B \vdash p \sim ?}\ (\text{CON-LEFT})$$

$$\frac{E;B \vdash p \quad E;B \vdash x^{c.i}}{E;B \vdash p \sim x^{c.i}}\ (\text{CON-DEPENDENT1}) \qquad \frac{E;B \vdash p \quad E;B \vdash x^{c.i}}{E;B \vdash x^{c.i} \sim p}\ (\text{CON-DEPENDENT2})$$

$\boxed{E;B \vdash t \leq t'}$

Traditional subtyping

$$\frac{E;B \vdash t}{E;B \vdash t \leq t}\ (\text{SUB-REFL}) \qquad \frac{E;B \vdash t \leq t' \quad E;B \vdash t' \leq t''}{E;B \vdash t \leq t''}\ (\text{SUB-TRANS})$$

$$\frac{E;B \vdash c\langle\sigma\rangle \qquad \texttt{class}\ c\langle\alpha_{i\in 1..n}\rangle\ \texttt{extends}\ c'\langle r_{i\in 1..n'}\rangle\{\ldots\}}{E;B \vdash c\langle\sigma\rangle \leq c'\langle\sigma(r_i)_{i\in 1..n'}\rangle}\ (\text{SUB-CLASS})$$

$\boxed{E;B \vdash t \sim t'}$ $\boxed{E;B \vdash t \lesssim t'}$ $\boxed{E;B \vdash t}$

"Good type"

$$\frac{E;B \vdash c\langle p_{i\in 1..n}\rangle \quad E;B \vdash c\langle q_{i\in 1..n}\rangle \quad p_i \sim q_i \forall i \in 1..n}{E;B \vdash (c\langle p_{i\in 1..n}\rangle \sim c\langle q_{i\in 1..n}\rangle)}\ (\text{CON-TYPE})$$

$$\frac{E;B \vdash c\langle\sigma\rangle \leq c'\langle\sigma'\rangle \quad E;B \vdash c'\langle\sigma'\rangle \sim c'\langle\sigma''\rangle}{E;B \vdash (c\langle\sigma\rangle \lesssim c'\langle\sigma''\rangle)}\ (\text{GRAD-SUB})$$

$$\frac{\text{arity}(c) = n \quad E;B \vdash p_1 \preceq p_i \quad \forall i \in 1..n}{E;B \vdash c\langle p_{i\in 1..n}\rangle}\ (\text{G-TYPE})$$

Consistent types

"Gradual Subtyping"

Type-Directed Compilation

Runtime checks are inserted
basing on the type information.

$$\boxed{E;B \vdash b \,:\, s}$$

(T-NEW)
$$\frac{E;B \vdash c\langle r_{i\in 1..n}\rangle}{E;B \vdash \texttt{new}\ c\langle r_{i\in 1..n}\rangle \,:\, c\langle r_{i\in 1..n}\rangle}$$

(T-LKP)
$$\frac{E;B \vdash z \,:\, c\langle\sigma\rangle \qquad \mathcal{F}_c(f)=t}{E;B \vdash z.f \,:\, \sigma_z(t)}$$

(T-LET)
$$\frac{E;B \vdash b \,:\, t \qquad E,x \,:\, \mathsf{fill}(x,t);B \vdash e \,:\, s}{E;B \vdash \texttt{let}\ x=b\ \texttt{in}\ e \,:\, s}$$

Field update

(T-UPD)
$$\frac{E;B \vdash z \,:\, c\langle\sigma\rangle \quad \mathcal{F}_c(f)=t \qquad E;B \vdash y \,:\, s \qquad E;B \vdash s \lesssim \sigma_z(t)}{E;B \vdash z.f=y \,:\, \sigma_z(t)}$$

Method call

(T-CALL)
$$\frac{E;B \vdash y \,:\, s \quad \mathcal{MT}_c(m)=(y',t\rightarrow t') \qquad E;B \vdash z \,:\, c\langle\sigma\rangle \quad E;B \vdash s \lesssim \sigma_z(t) \qquad \sigma' \equiv \sigma \uplus \{y' \mapsto y\}}{E;B \vdash z.m(y) \,:\, \sigma'_z(t')}$$

(VAL-w)
$$\frac{E;B \vdash \diamond\ w \,:\, s \in E}{E;B \vdash w \,:\, s}$$

(VAL-NULL)
$$\frac{E;B \vdash t}{E;B \vdash \texttt{null} \,:\, t}$$

$$\boxed{E \vdash t'\ m(t\ y)\ \{e\}} \qquad \boxed{\vdash P;\ e}$$

Method return

(METHOD)
$$\frac{E,y \,:\, \mathsf{fill}(y,t) \vdash e \,:\, s \qquad E \vdash s \lesssim t'}{E \vdash t'\ m(t\ y)\ \{e\}}$$

(PROGRAM)
$$\frac{\vdash class_j\ \ \forall class_j \in P \qquad E \vdash e \,:\, t}{E \vdash P;\ e}$$

Gradual subtyping
might cause
check insertion

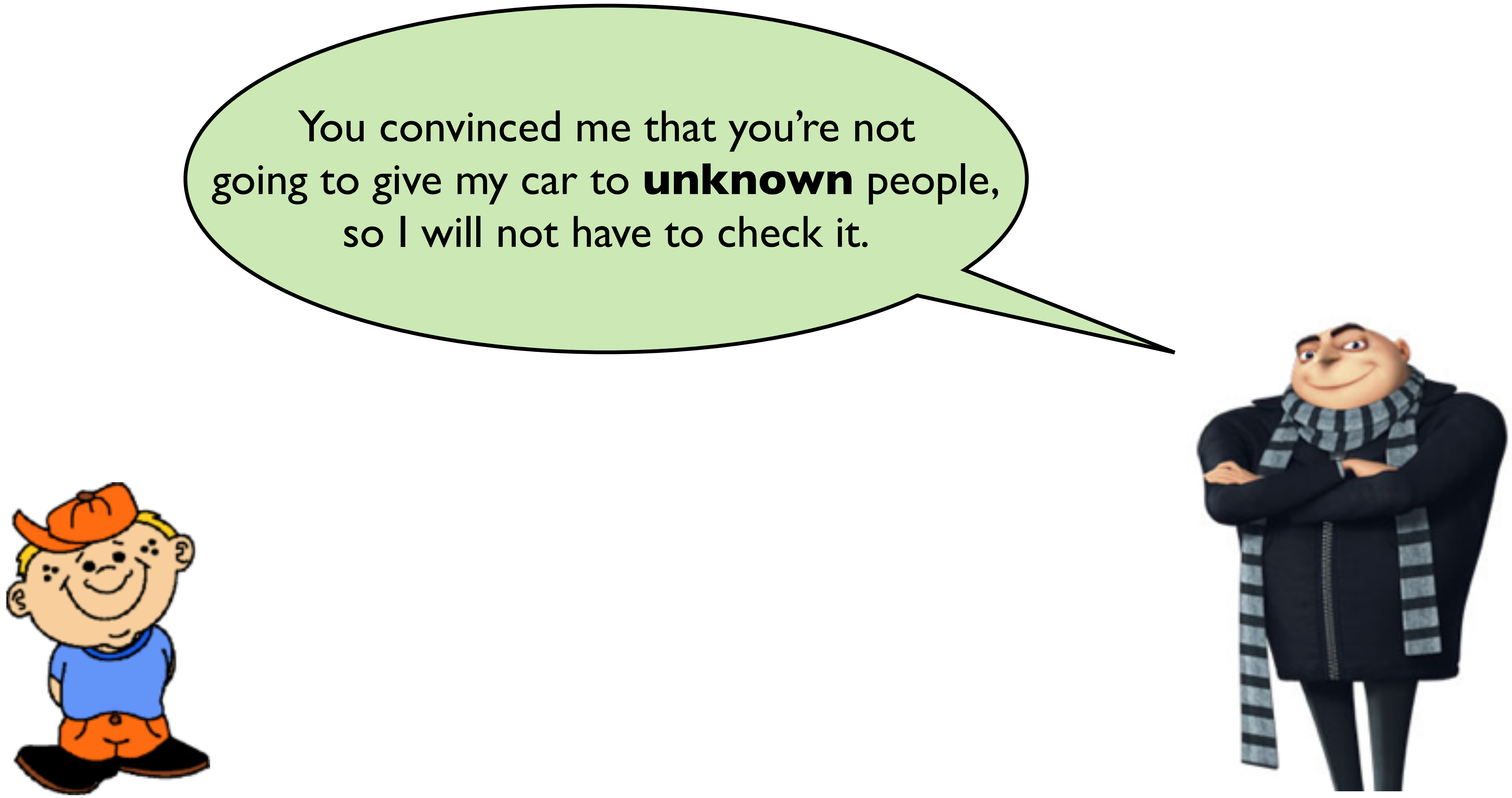
Gradual Typing and Compilation
(informally)

Theorem 1:
No unknown owners ⇒ no dynamic casts

Corollary :
No unknown owners ⇒ static invariant guaranty

(And also, no runtime overhead and failed casts)

Theorem 2:
A (gradually) well-typed program is compiled into a (statically) well-typed program.

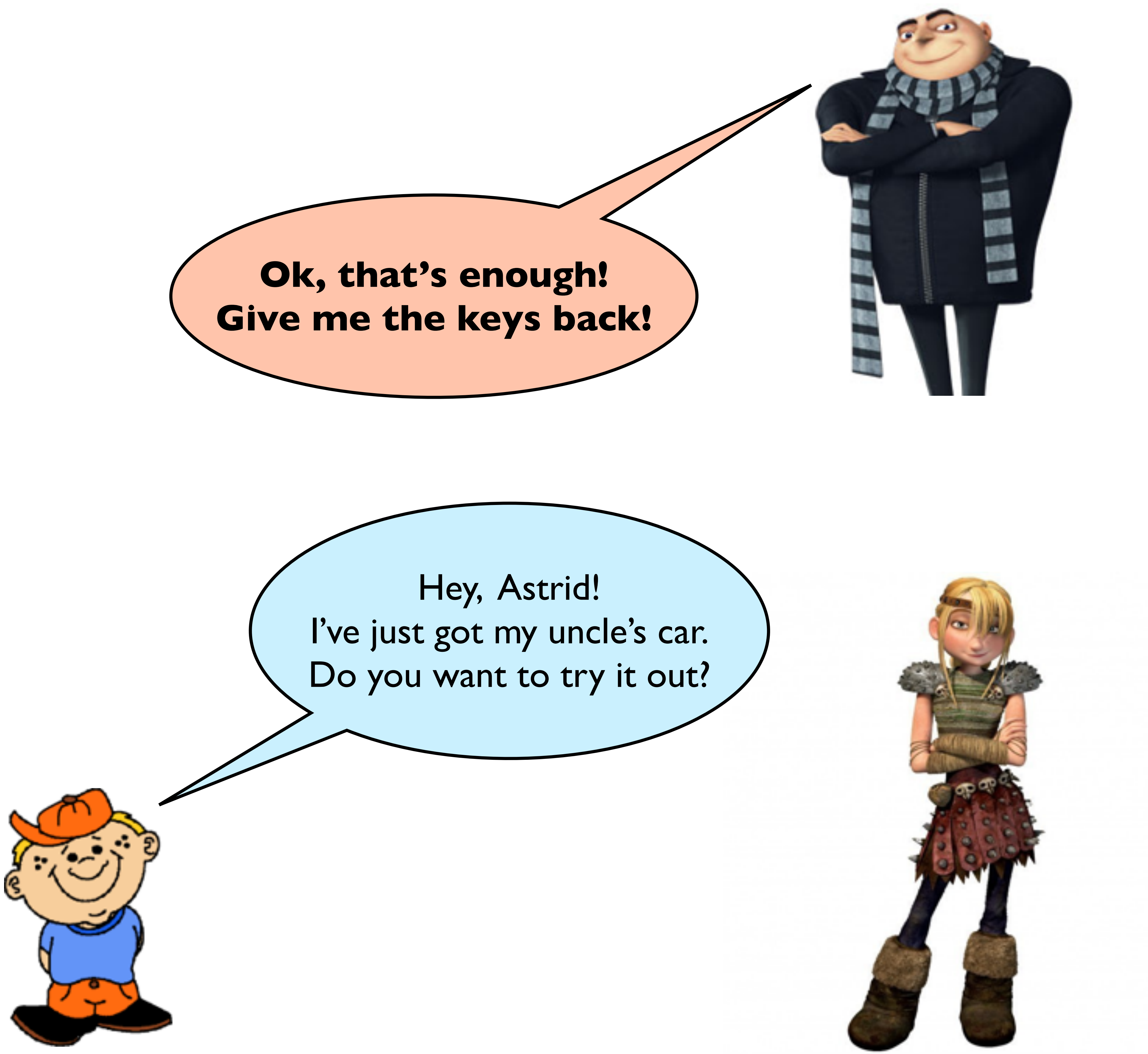

You convinced me that you're not going to give my car to **unknown** people, so I will not have to check it.

Type Safety Result

(informally)

Theorem 3:
A (statically) well-typed program does not violate the OAD invariant but might fail on a dynamic check.

Corollary:
A gradually well-typed program, being compiled, does not violate the OAD invariant.


Ok, that’s enough!
Give me the keys back!
Hey, Astrid!
I’ve just got my uncle’s car.
Do you want to try it out?

Implementation

- Implemented in JastAddJ [Ekman-Hedin:OOPSLA07]
  - Extended JastAddJ compiler for Java 1.4
- 2,600 LOC (not including tests and comments)
- Check insertion ⇒ compilation warning
- Source-to-source translation

Experience

- Java Collection Framework (JDK 1.4.2)
  - 46 source files, ~8,200 LOC
- Securing inner `Entries` of collections
- Questions addressed:
  - How many annotations are needed minimally?
  - What is the execution cost?
  - How many annotations for full static checking?

Experience

- Minimal amount of annotations
  - `LinkedList` - 17
  - `LinkedMap` - 15
- Performance overhead
  - ~1.5-2 times (for extensive updates)
- Full migration
  - `LinkedList` - yes, 34 annotations
  - `LinkedMap` - no, because of static factory methods
    - (best - 28 annotations)

Contributions II

1. A formalization of a gradual ownership type system and a type-directed compilation for a Java-like language
   - Proofs of safety result for type-directed compilation
2. An implementation of a translating compiler for gradual ownership types
   - Supports *full* Java 1.4
   - Available at `http://github.com/ilyasergey/Gradual-Ownership`
3. A report on program migration using gradual ownership types
   - Migrated several classes from Java Collection Framework 1.4.2
4. A discussion on gradualization of type systems for object ownership

Future Work II

1. Gradual ownership types in higher-order languages
   - Introduced notion of *dependent owners* is similar to *blame labels*
2. Gradual ownership types meet shape and pointer analysis
   - Imposed dynamic encapsulation invariant can be employed when inferring shape information of data structures
3. IDE Support
   - Gradual compiler emits warning messages that can be used to indicate invariant violations statically

The Thesis

KU LEUVEN

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science

Operational Aspects of Type Systems

Inter-Derivable Semantics of Type Checking
and Gradual Types for Object Ownership

Ilya SERGEY

Dissertation presented in partial
fulfilment of the requirements for
the degree of Doctor
in Engineering

November 2012

Thanks

Appendix

And also

1. Ilya Sergey, Jan Midtgaard and Dave Clarke
   Calculating Graph Algorithms for Dominance and Shortest Path
   In proceedings of MPC 2012, June 2012. Volume 7342 of LNCS, Springer.

   - Invited for publication in a journal special issue

2. Christopher Earl, Ilya Sergey, Matthew Might and David Van Horn
   Introspective Pushdown Analysis of Higher-Order Programs
   In Proceedings of ICFP 2012, September 2012. ACM.

   - Invited for publication in a journal special issue

3. Dominique Devriese, Ilya Sergey, Dave Clarke and Frank Piessens
   Fixing Idioms: a Recursion Primitive for Applicative DSLs
   Accepted to PEPM 2013.

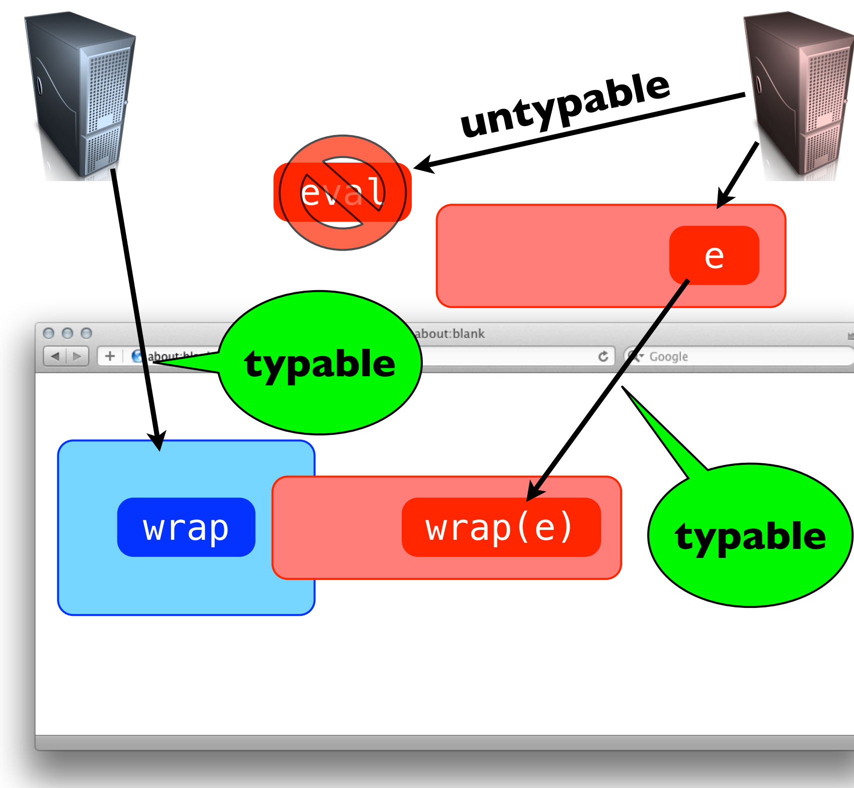
4. Ilya Sergey, Dave Clarke and Alexander Podkhalyuzin
   Automatic refactorings for Scala programs
   Scala Days 2010 Workshop. April 2010.

5. Dave Clarke and Ilya Sergey
   A semantics for context-oriented programming with layers
   In proceedings of Workshop on Context-Oriented Programming (COP 2009), June 2009. ACM.

Gradual
Types for Web Security

- Secure contexts for JavaScript evaluation are modeled by sandboxes
- Sandboxes can be modeled as a type system, resulting in static verification

Semantics and Types for Safe Web Programming
A. Guha, PhD Thesis, 2012



Untypable $\neq$ Forbidden

Gradual Ownership Types and Ownership Types Inference*

	Gradual Ownership Types	Ownership Types Inference
Straightforward correspondence to the TS	+	-
Modular	+	-
Effective debugging of type checking	+	-
Well-typed ~ full static safety	-	+
Minimal amount of annotations	required	optional
No runtime overhead	-	+

* Huang-Milanova:IWACO11