Typed Compilation Against Non-Manifest Base Classes

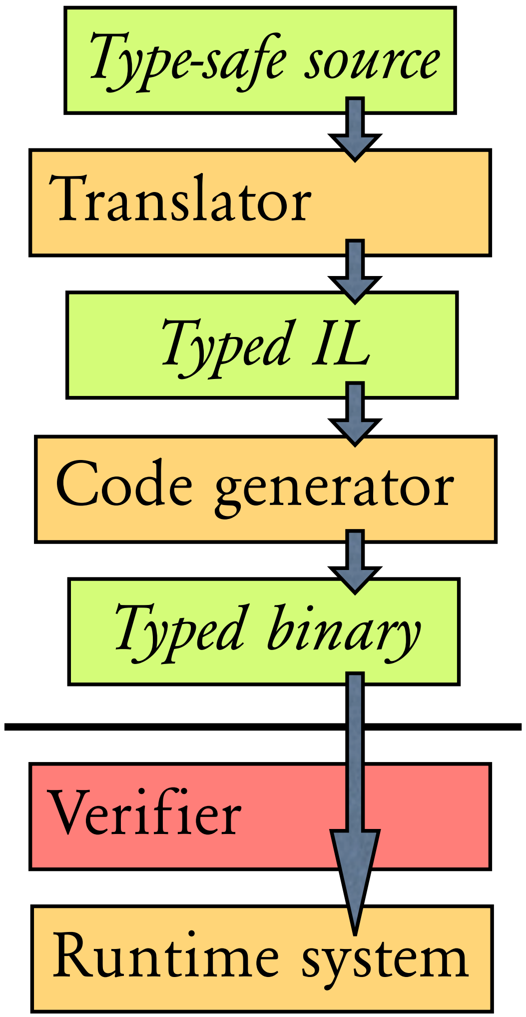
Christopher League
Long Island University
christopher.league@liu.edu

Stefan Monnier
Université de Montréal
monnier@iro.umontreal.ca

FTfJP workshop
26 July 2005

Goal: certifying compiler

…for Java-like languages



- Type safety is necessary for security
- Fairly well understood for core Java/C#
  [Colby et al., PLDI ’00]
  [League et al., TOPLAS ’02, CC ’03]
- What about *next generation* object-oriented features?

A characteristic of next-gen OO

- Looser coupling between classes in a hierarchy
  - ➞ Classes as module parameters
  - ➞ First-class classes
  - ➞ Mixins, Traits
  - ➞ Java's binary compatibility

Q: *How much information about a base class is needed to compile its derived class?*

LONG ISLAND UNIVERSITY

Non-manifest base classes

- Base class not available for inspection when derived class is compiled
- Implementations use *dictionary* to map method/field names to their locations in the object layout
  - → Dictionary lookup may occur at link time or run time

LONG ISLAND UNIVERSITY

Example in Objective Caml

```
module type CIRCLE =
sig type spec
    class circle : spec ->
      object method center : float*float
             method radius : float
      end
end
```

Example in Objective Caml

```
module CircleBBox =
  functor (C : CIRCLE) ->
  struct
    class bbox arg = object (self)
      inherit C.circle arg
      method bounds =
        let (x,y) = self#center in
        let r = self#radius in
        ((x-r,y-r), (x+r,y+r))
    end
  end
```

Where are these methods?

Non-manifest base class

Granularity of IL

- Method dispatch is *atomic* at source level: `x.m(…);`
- In IL, it should be decomposed into:
  - → null check (if applicable)
  - → dictionary lookup
  - → method dereference
  - → function call (indirect)

A generic IL for OO languages

- *Links* [Fisher et al., ESOP '00]
  - Its primitives express a "wide range of class-based OO features, including various forms of method dispatch."
  - But, it is not typed.

LONG ISLAND UNIVERSITY

Our contribution

- A sound & decidable type system for *Links*
  - ➙ using the *Type-Safe Certified Binaries* framework, "for explicitly representing complex propositions and proofs in typed ILs and object code" [Shao et al., POPL '02]
  - ➙ type and proof system based on *Calculus of Inductive Constructions* [Coquand & Paulin-Mohring, '90]

LONG ISLAND UNIVERSITY

Outline

1. Review of *Links* primitives
2. Introduction to *TSCB* framework
3. Our computation language: LITL
4. Encoding objects

Links syntax

$$
\begin{aligned}
e ::= {} & x \mid n \mid e_1 + e_2 \mid \lambda x.e \mid e_1\, e_2 \\
& \mid \langle e_1, \ldots, e_n \rangle \mid e_1 @ e_2 \\
& \mid e_1 @ e_2 \leftarrow e_3 \mid e; \langle e_1, \ldots, e_n \rangle \\
& \mid \{ l_1 = e_1, \ldots, l_n = e_n \} \mid e \# l
\end{aligned}
$$

Links syntax

$$
\begin{aligned}
e ::= {} & x \mid n \mid e_1 + e_2 \mid \boxed{\lambda x.e \mid e_1\ e_2} \\
& \mid \langle e_1, \ldots, e_n \rangle \mid e_1 @ e_2 \\
& \mid e_1 @ e_2 \leftarrow e_3 \mid e; \langle e_1, \ldots, e_n \rangle \\
& \mid \{ l_1 = e_1, \ldots, l_n = e_n \} \mid e \# l
\end{aligned}
$$

- Untyped lambda calculus

Links syntax

$$
\begin{aligned}
e ::= x \mid \boxed{n \mid e_1 + e_2} \mid \lambda x.e \mid e_1\, e_2 \\
\mid \langle e_1, \ldots, e_n \rangle \mid e_1 @ e_2 \\
\mid e_1 @ e_2 \leftarrow e_3 \mid e; \langle e_1, \ldots, e_n \rangle \\
\mid \{ l_1 = e_1, \ldots, l_n = e_n \} \mid e \# l
\end{aligned}
$$

- Natural numbers and addition
  - → to represent *offsets* of fields & methods

Links syntax

$$
\begin{aligned}
e ::= {} & x \mid n \mid e_1 + e_2 \mid \lambda x.e \mid e_1\, e_2 \\
\mid {} & \boxed{\langle e_1, \ldots, e_n \rangle \mid e_1 @ e_2} \\
\mid {} & e_1 @ e_2 \leftarrow e_3 \mid e; \langle e_1, \ldots, e_n \rangle \\
\mid {} & \{ l_1 = e_1, \ldots, l_n = e_n \} \mid e \# l
\end{aligned}
$$

- Tuple construction and projection
  - tuples represent objects and virtual function tables.
  - projection is *cheap*.

Links syntax

$$
\begin{aligned}
e ::= x \mid n \mid e_1 + e_2 \mid \lambda x.e \mid e_1\, e_2 \\
\mid \langle e_1, \ldots, e_n \rangle \mid e_1 @ e_2 \\
\mid \boxed{e_1 @ e_2 \leftarrow e_3 \mid e; \langle e_1, \ldots, e_n \rangle} \\
\mid \{ l_1 = e_1, \ldots, l_n = e_n \} \mid e \# l
\end{aligned}
$$

- Functional update and extension
  - ➙ needed for overriding and adding new methods

Links syntax

$$
\begin{aligned}
e ::= {} & x \mid n \mid e_1 + e_2 \mid \lambda x.e \mid e_1\ e_2 \\
& \mid \langle e_1, \ldots, e_n \rangle \mid e_1 @ e_2 \\
& \mid e_1 @ e_2 \leftarrow e_3 \mid e; \langle e_1, \ldots, e_n \rangle \\
& \mid \boxed{\{l_1 = e_1, \ldots, l_n = e_n\} \mid e \# l}
\end{aligned}
$$

- Dictionary construction and lookup
  - ➞ lookup involves search (can be expensive)
  - ➞ used to map method labels to offsets

Compiling method dispatch

If the vtable is at offset 0 of object x, and d maps method labels to vtable offsets, then the method dispatch

$$x.m(\ldots)$$

expands to

$$((x\,@\,0)\,@\,(d\,\#\,m))\ x\ \ldots$$

Typing this seems hopeless!

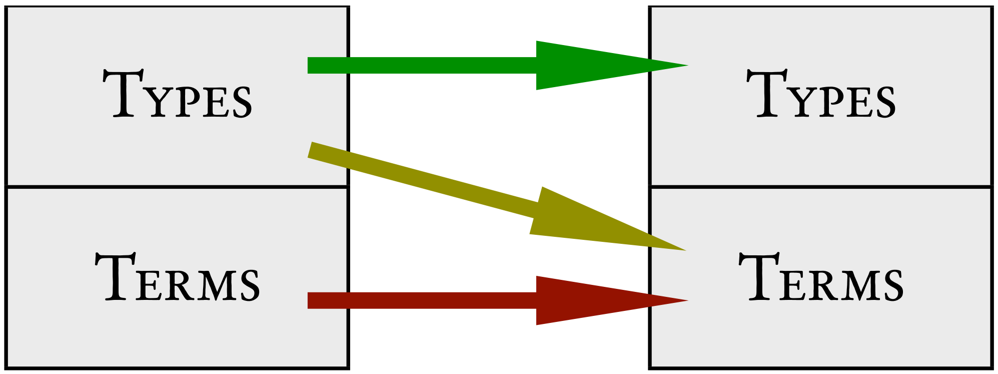
$$((x \,@\, 0) \,@\, (d \,\#\, m))\; x \;\ldots$$

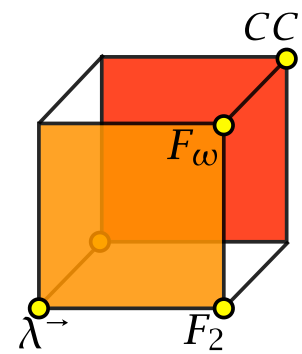
- What we don't know *could* hurt us:
  - → size/structure of vtable
  - → offset returned by dictionary
  - → is the index within bounds?
  - → does it return a function pointer?
- Subtle connection between *x* and *d*
- Typed IL must capture these invariants

Outline

1. Review of *Links* primitives
2. Introduction to *TSCB* framework
3. Our computation language: LITL
4. Encoding objects

Dependencies in the λ cube



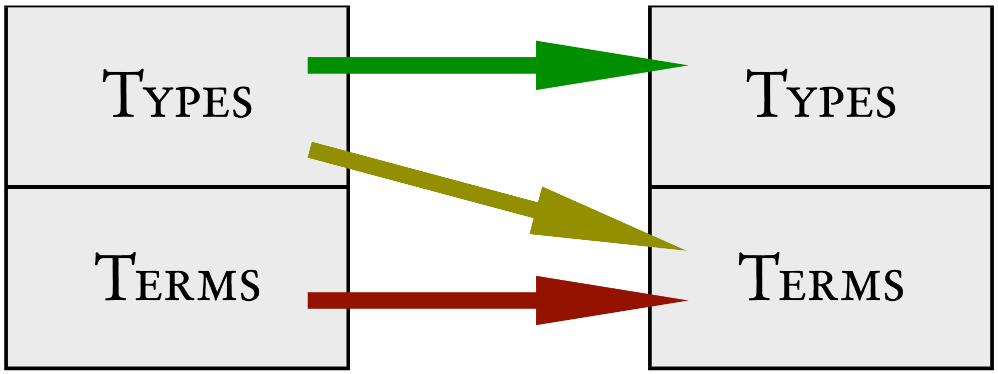


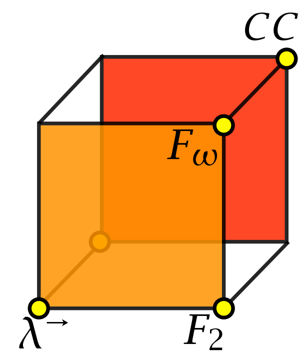
$$\mathit{twice} : \mathit{nat} \to \mathit{nat} \qquad \lambda^\to$$

$$\mathit{id} : \forall \alpha{:}\Omega.\ \alpha \to \alpha \qquad F_2$$

$$\mathit{list} : \Omega \to \Omega \qquad F_\omega$$

Traditional typed ILs stop here



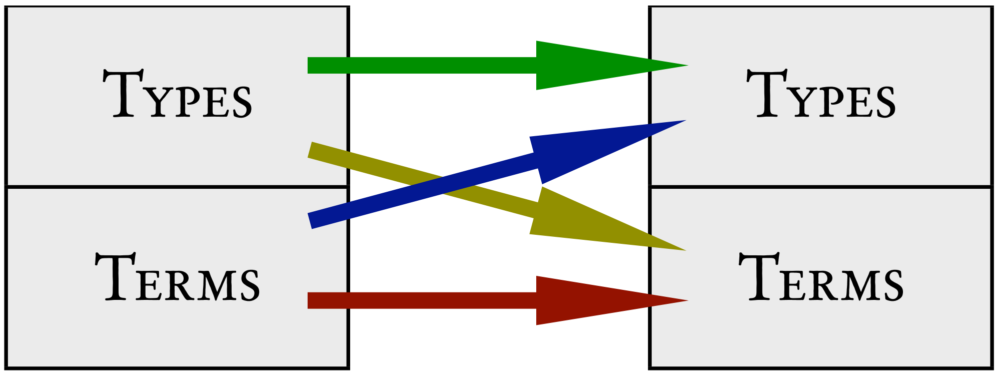


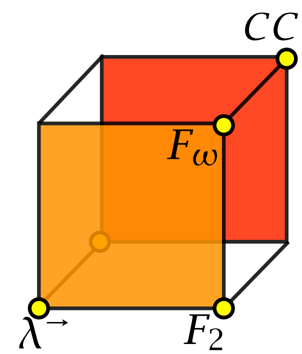
$\mathit{twice} : \mathit{nat} \rightarrow \mathit{nat}$	$\lambda^{\rightarrow}$
$\mathit{id} : \forall \alpha{:}\Omega.\ \alpha \rightarrow \alpha$	F_2
$\mathit{list} : \Omega \rightarrow \Omega$	F_ω

...and augment TERMS with *effects* (like non-termination)

LONG ISLAND UNIVERSITY

Dependencies in CC





$$\mathit{twice} : \mathit{nat} \rightarrow \mathit{nat}$$ $\lambda^{\rightarrow}$

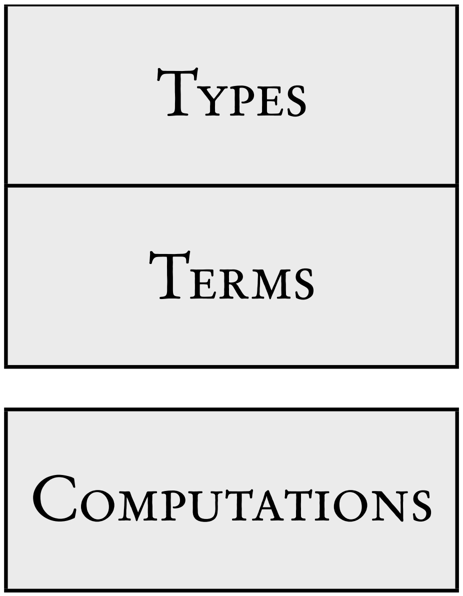
$$\mathit{id} : \forall \alpha{:}\Omega.\, \alpha \rightarrow \alpha$$ F_2

$$\mathit{list} : \Omega \rightarrow \Omega$$

$$\mathit{array} : \mathit{nat} \rightarrow \Omega \rightarrow \Omega$$

...now adding effects is dangerous!

Solution: add another layer



- The 'types' that govern computations are inductively defined TERMS in CIC.
  - We can use the *Coq Proof Assistant* as a type checker for CIC.
  - Sorts: SET, PROP, TYPE

Outline

1. Review of *Links* primitives
2. Introduction to *TSCB* framework
3. Our computation language: LITL*
4. Encoding objects

*LITL Is Typed Links

Preliminary definitions

Inductive $nat : \textsc{Set} \equiv$
$\mid O : nat$
$\mid S : nat \to nat.$

Parameter $sym : \textsc{Set}.$ (* to represent labels *)

Inductive $option\ (A : \textsc{Set}) : \textsc{Set} \equiv$
$\mid Some : A \to option\ A$
$\mid None : option\ A.$

LONG ISLAND UNIVERSITY

Reasoning about things

$$\textbf{Inductive}\ lt : nat \to nat \to \textsc{Prop} \equiv$$
$$\mid ltzs : \Pi\, n : nat.\, lt\ O\ (S\ n)$$
$$\mid ltss : \Pi\, n\ m : nat.\, lt\ n\ m \to lt\ (S\ n)\ (S\ m).$$

$$\textbf{Inductive}\ eq\ (A : \textsc{Type})\ (x : A) : A \to \textsc{Prop} \equiv$$
$$refl\_equal : eq\ A\ x\ x.$$

Types for Links primitives

$$
\begin{aligned}
&\textbf{Inductive}\ \mathit{Ty} : \textsc{Set}\ \equiv \\
&\quad \mid \mathit{arw} : \mathit{Ty} \to \mathit{Ty} \to \mathit{Ty} \\
&\quad \mid \mathit{snat} : \mathit{nat} \to \mathit{Ty} \\
&\quad \mid \mathit{tup} : \mathit{nat} \to (\mathit{nat} \to \mathit{Ty}) \to \mathit{Ty} \\
&\quad \mid \mathit{dict} : (\mathit{sym} \to \mathit{option}\ \mathit{Ty}) \to \mathit{Ty} \\
&\quad \mid \mathit{mu} : \Pi\, k : \textsc{Set}.\ (k \to \mathit{Ty}) \to \mathit{Ty} \\
&\quad \mid \mathit{all} : \Pi\, k : \textsc{Set}.\ (k \to \mathit{Ty}) \to \mathit{Ty} \\
&\quad \mid \mathit{ex} : \Pi\, k : \textsc{Set}.\ (k \to \mathit{Ty}) \to \mathit{Ty}.
\end{aligned}
$$

Types for Links primitives

$$
\begin{aligned}
&\textbf{Inductive}\ \mathit{Ty} : \textsc{Set} \ \equiv \\
&\quad \mid \boxed{\mathit{arw} : \mathit{Ty} \to \mathit{Ty} \to \mathit{Ty}} \\
&\quad \mid \mathit{snat} : \mathit{nat} \to \ldots \\
&\quad \mid \mathit{tup} : \mathit{nat} \to \ldots \\
&\quad \mid \mathit{dict} : (\mathit{sym} \to \ldots \\
&\quad \mid \mathit{mu} : \Pi k : \textsc{Set}.\ (k \to \mathit{Ty}) \to \mathit{Ty} \\
&\quad \mid \mathit{all} : \Pi k : \textsc{Set}.\ (k \to \mathit{Ty}) \to \mathit{Ty} \\
&\quad \mid \mathit{ex} : \Pi k : \textsc{Set}.\ (k \to \mathit{Ty}) \to \mathit{Ty}.
\end{aligned}
$$

The type of functions (introduced by λ)

Types for Links primitives

$$
\begin{aligned}
&\textbf{Inductive } Ty : \textsc{Set} \equiv \\
&\mid arw : Ty \to Ty \to Ty \\
&\mid \boxed{snat : nat \to Ty} \\
&\mid tup : nat \to (nat \to Ty) \to Ty \\
&\mid dict : (sym \to \\
&\mid mu : \Pi k : \mathrm{S} \\
&\mid all : \Pi k : \mathrm{Se} \\
&\mid ex : \Pi k : \mathrm{Se}
\end{aligned}
$$

Singleton type for natural numbers:

$$\vdash 0 : snat\ O$$
$$\vdash 1 : snat\ (S\ O)$$
$$\vdash 2 : snat\ (S\ (S\ O))$$

LONG ISLAND UNIVERSITY

Types for Links primitives

Inductive $Ty : \textsc{Set} \;\equiv$

$\mid \mathit{arw} : Ty \to Ty \to Ty$

$\mid \mathit{snat} : \mathit{nat} \to Ty$

$\mid \boxed{\mathit{tup} : \mathit{nat} \to (\mathit{nat} \to Ty) \to Ty}$

$\mid \mathit{dict} : (\mathit{sym} \to$

$\mid \mathit{mu} : \Pi k : \mathrm{S}$

$\mid \mathit{all} : \Pi k : \mathrm{S}$

$\mid \mathit{ex} : \Pi k : \mathrm{S}$

The type of tuples:

$$\frac{\Delta ; \Gamma \vdash e_1 : \mathit{tup}\; n\; f \qquad \Delta ; \Gamma \vdash e_2 : \mathit{snat}\; i \qquad \Delta \vdash \sigma : \mathit{lt}\; i\; n}{\Delta ; \Gamma \vdash e_1 @ e_2 [\sigma] : f\; i}$$

Types for Links primitives

$$
\begin{array}{l}
\textbf{Inductive}\ Ty : \textsc{Set}\ \equiv \\
\quad \mid arw : Ty \to Ty \to Ty \\
\quad \mid snat : nat \to Ty \\
\quad \mid tup : nat \to (nat \to Ty) \to Ty \\
\quad \mid \boxed{dict : (sym \to option\ Ty) \to Ty} \\
\quad \mid mu : \Pi k : \textsc{Set}.\ (k \to Ty) \to Ty \\
\quad \mid all : \Pi k : \textsc{Se} \\
\quad \mid ex : \Pi k : \textsc{Se}
\end{array}
$$

The type of dictionaries:

$$
\frac{\Delta ; \Gamma \vdash e \ : \ dict\ g \qquad \Delta \vdash \sigma \ : \ eq\ (g\ l)\ (Some\ \tau)}{\Delta ; \Gamma \vdash e \# l\,[\sigma] \ : \ \tau}
$$

Long Island University

Types for Links primitives

$$
\begin{aligned}
&\textbf{Inductive}\ \mathit{Ty} : \textsc{Set}\ \equiv \\
&\quad \mid \mathit{arw} : \mathit{Ty} \to \mathit{Ty} \to \mathit{Ty} \\
&\quad \mid \mathit{snat} : \mathit{nat} \to \\
&\quad \mid \mathit{tup} : \mathit{nat} \to \\
&\quad \mid \mathit{dict} : (\mathit{sym} \to \mathit{option}\ \mathit{Ty}) \to \mathit{Ty} \\
&\quad \mid \boxed{\mathit{mu} : \Pi k : \textsc{Set}.\ (k \to \mathit{Ty}) \to \mathit{Ty}} \\
&\quad \mid \boxed{\mathit{all} : \Pi k : \textsc{Set}.\ (k \to \mathit{Ty}) \to \mathit{Ty}} \\
&\quad \mid \boxed{\mathit{ex} : \Pi k : \textsc{Set}.\ (k \to \mathit{Ty}) \to \mathit{Ty}.}
\end{aligned}
$$

Recursive types, universal and existential quantifiers.

Type-annotated Links syntax

$$
\begin{aligned}
e ::= {} & x \mid n \mid e_1 + e_2 \mid \lambda x{:}\tau.e \\
& \mid \Lambda\alpha{:}\ldots \\
& \mid \langle e_1, \ldots \\
& \mid e_1 \mathbin{@} \ldots \\
& \mid \{ l_1 = \ldots \\
& \mid \boxed{\mathsf{cast}\,[\,\sigma\,]\,e} \mid [\tau_1, e \rhd \tau_2] \\
& \mid \mathsf{open}\, e_1 \;\mathsf{as}\, [\alpha, x]\, \mathsf{in}\, e_2 \\
& \mid \mathsf{fold}\, e \;\mathsf{as}\; \tau \mid \mathsf{unfold}\, e
\end{aligned}
$$

$$
\frac{\Delta \vdash \sigma : eq\ \tau_1\ \tau_2 \qquad \Delta;\Gamma \vdash e : \tau_1}{\Delta;\Gamma \vdash \mathsf{cast}\,[\,\sigma\,]\,e : \tau_2}
$$

Outline

1. Review of *Links* primitives
2. Introduction to *TSCB* framework
3. Our computation language: LITL
4. Encoding objects

Recall: method dispatch in Links

$$(x\,@\,(d\,\#\,m))\;x\;\ldots$$

Suppose that:

$$\Delta;\Gamma \vdash x : \mathit{tup}\; n\; f$$
$$\Delta;\Gamma \vdash d : \mathit{dict}\; g$$

These are unknown

The constraints on n, f, g are:

$$\Delta;\Gamma \vdash \sigma_1 : \mathit{eq}\;(g\;m)\;(\mathit{Some}\;(\mathit{snat}\;i))$$
$$\Delta;\Gamma \vdash \sigma_2 : \mathit{lt}\;i\;n$$
$$\Delta;\Gamma \vdash \sigma_3 : \forall\beta{:}\mathit{Ty}.\,\mathit{eq}\;(f\;\beta\;i)\;(\mathit{arw}\;\beta\;\tau)$$

These say “object has a method m at offset i returning type τ”

Encapsulating the object rep.

Definition $Rep : \textsc{Set} \;\equiv$
$(nat \;\times\; (Ty \to nat \to Ty) \times$
$(sym \to option\ Ty))$.

Definition $size \;\equiv\; \lambda\, r : Rep$.
match r **with** $(n, \_, \_) \Rightarrow n$ **end**.
Definition $tupfn \;\equiv\; \lambda\, r : Rep$.
match r **with** $(\_, f, \_) \Rightarrow f$ **end**.
Definition $dictfn \;\equiv\; \lambda\, r : Rep$.
match r **with** $(\_, \_, g) \Rightarrow g$ **end**.

Expressing the constraints

$$
\begin{array}{l}
\textbf{Inductive } \mathit{HasMethod} \\
\qquad (r : \mathit{Rep})\ (m : \mathit{sym})\ (t : \mathit{Ty}) : \textsc{Set}\ \equiv \\
\quad \mathit{method} : \Pi\, i : \mathit{nat}.\ \mathit{lt}\ i\ (\mathit{size}\ r) \rightarrow \\
\qquad \mathit{eq}\ (\mathit{dictfn}\ r\ m)\ (\mathit{Some}\ (\mathit{snat}\ i)) \rightarrow \\
\qquad (\Pi\, \mathit{self}.\ \mathit{eq}\ (\mathit{tupfn}\ r\ \mathit{self}\ i)\ (\mathit{arw}\ \mathit{self}\ t)) \rightarrow \\
\qquad \mathit{HasMethod}\ r\ m\ t.
\end{array}
$$

The type of an object

$$
\begin{aligned}
&\exists r : Rep. \\
&\exists p : HasMethod\ r\ m\ t. \\
&\mu self : Ty. \\
&tup\ (size\ r)\ (tupfn\ r\ self) \times dict\ (dictfn\ r)
\end{aligned}
$$

Conclusions

- *LITL* = sound, low-level typed IL with dictionaries & tuples
  - ➙ efficient target for various OO languages
  - ➙ even when layout of base class unknown
- *TSCB* is a powerful framework!
- A separate issue: encoding the subtype relationships of the source language
  - ➙ works in many cases; more work needed