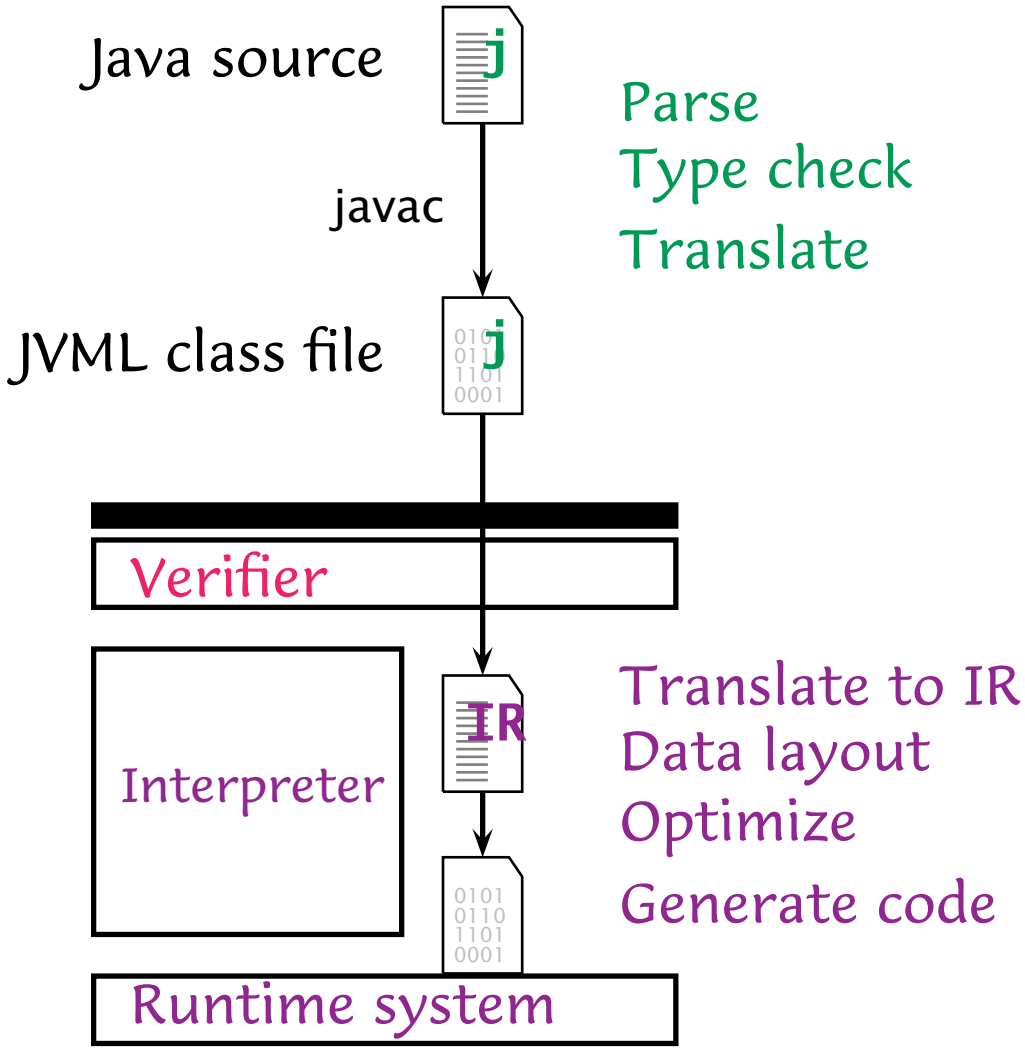
Type-Preserving Compilation of Featherweight Java

Christopher League
Valery Trifonov
Zhong Shao

FOOL 8
20 January 2001

Java mobile code platform

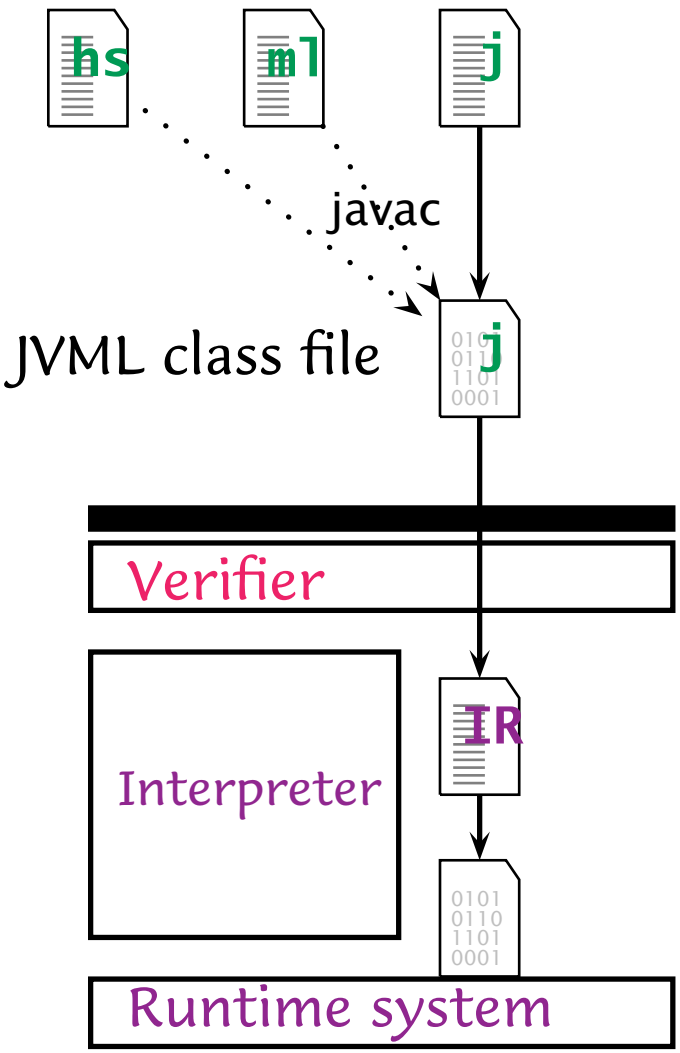


```
public class Example {
    public static void main (String[] args) {
        if (args.length == 0)

                System.out.println("None");



        else System.out.println(args.length);



    }
}
```

JVML is very high-level

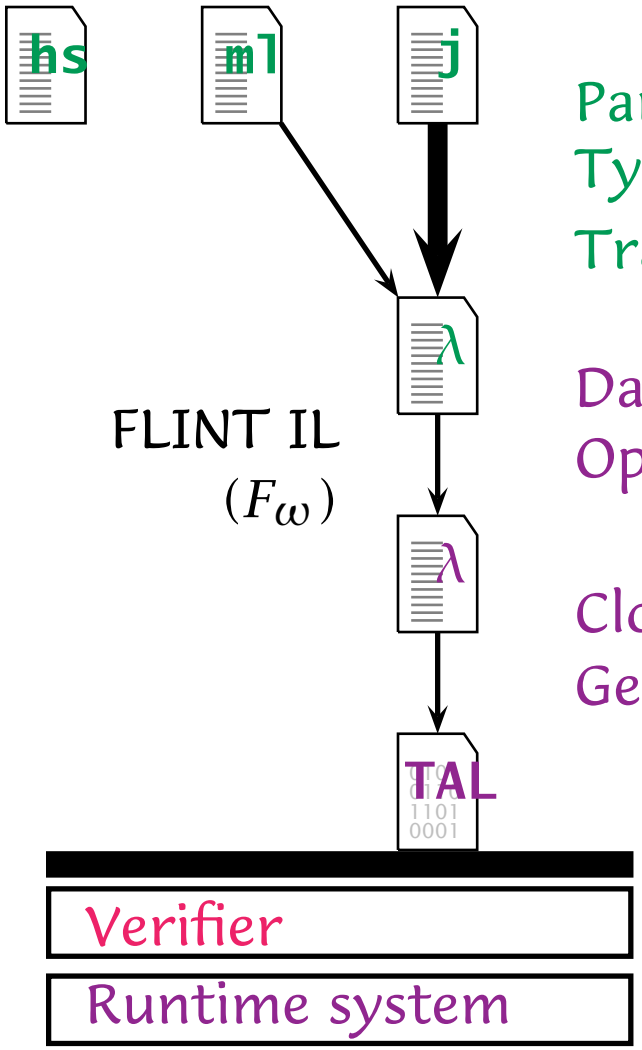
```
//
            0  aload_0
            1  arraylength
            2  ifne 16
//
            5  getstatic       ‹java.io.PrintStream out›
            8  ldc             ‹"None"›
           10  invokevirtual   ‹void println(java.lang.String)›
           13  goto 24
//
           16  getstatic       ‹java.io.PrintStream out›
           19  aload_0
           20  arraylength
           21  invokevirtual   ‹void println(int)›
           24  return
```

Shortcomings of JVML



1. Poor support for other source languages
2. Cannot express data layouts, optimizations
3. Enormous trusted computing base
4. Complex semantics

A more principled & flexible platform

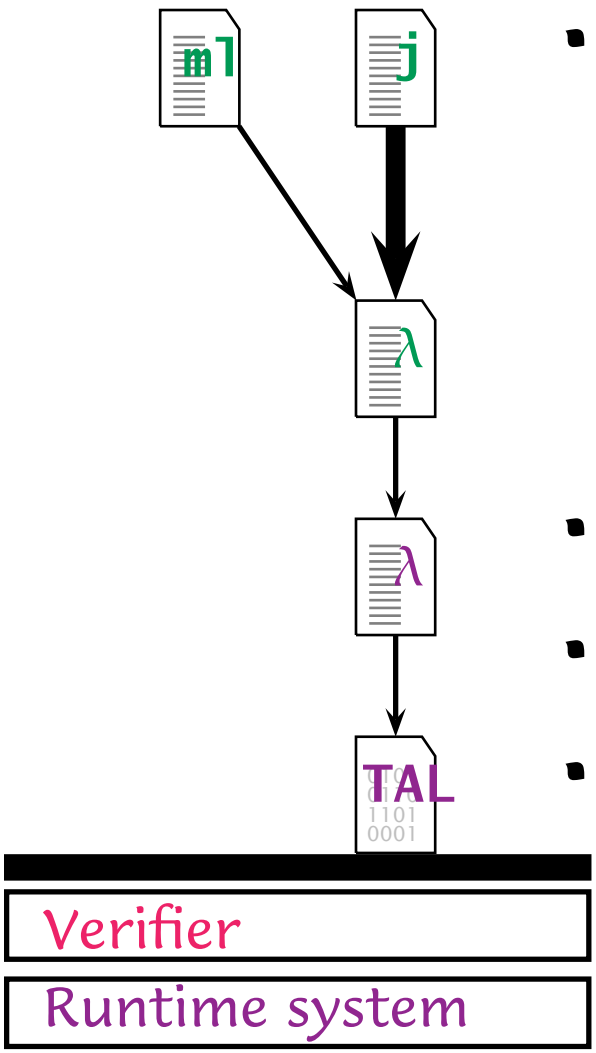


Parse
Type check
Translate

Data layout
Optimize

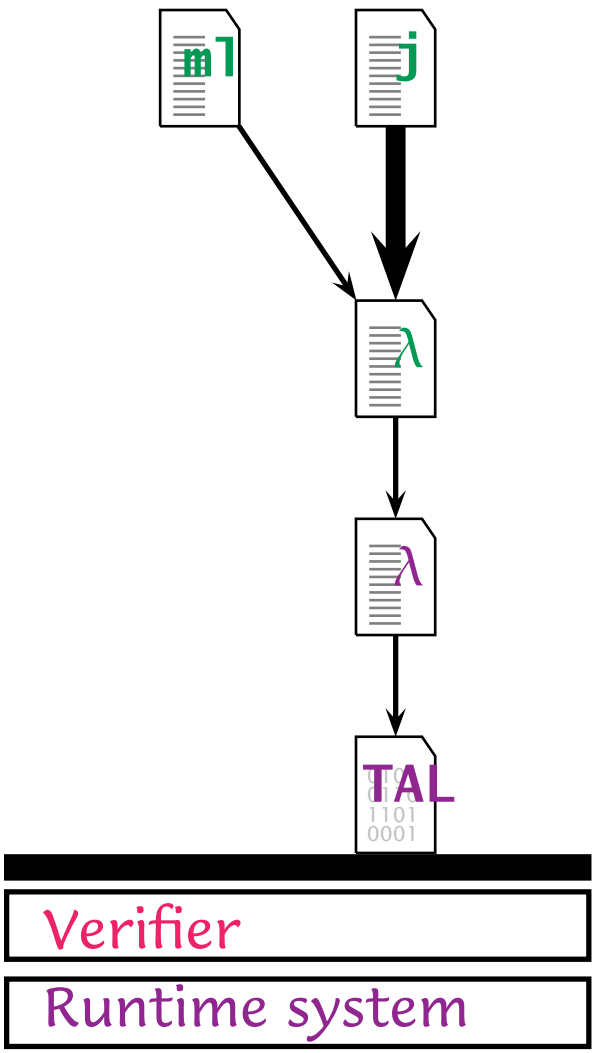
Closure-convert
Generate code

Challenges



- New features and abstractions
  - classes, inheritance, interfaces
  - privacy, mutual recursion
  - name equivalence, dynamic cast
  - constructors, super, static
- Sophisticated type systems
- Fast type checking
- No runtime overhead

Contribution



- Simple, sound, decidable target language
  - Supports SML
- Formal translation of Featherweight Java
  - Dynamic cast
  - Mutual recursion
  - Separate compilation
  - Extends naturally to a significant subset of Java
- Type preservation proof

Featherweight Java

$$CL ::= \text{class C extends C } \{\overline{\text{C f;}}\ K\ \overline{M}\}$$

$$K ::= \text{C}(\overline{\text{C f}})\ \{\text{super}(\overline{\text{f}});\ \overline{\text{this.f = f;}}\}$$

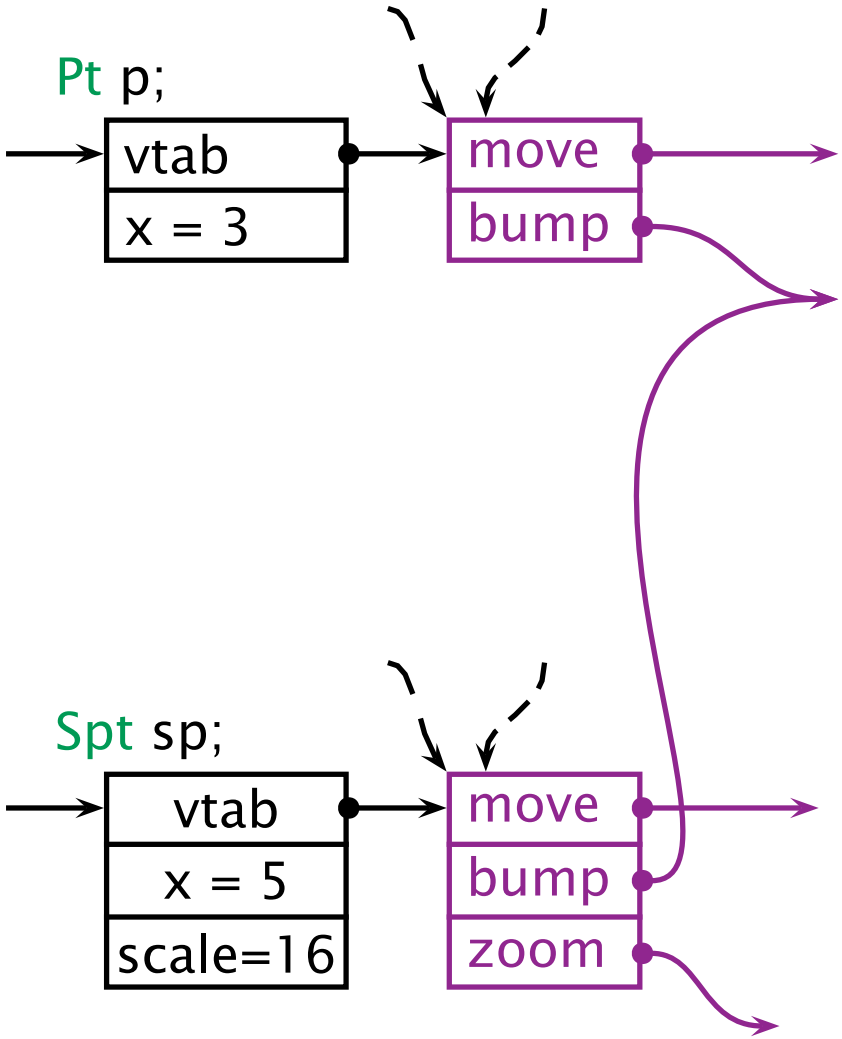
$$M ::= \text{C m}(\overline{\text{C } x})\ \{\uparrow e;\}$$

$$e ::= x \mid e.\text{f} \mid e.\text{m}(\overline{e}) \mid \text{new C}(\overline{e}) \mid (\text{C})e$$

[Igarashi, Pierce, Wadler 99]

```
class Pt {
  int x;
  Pt (int x)  { this.x = x; }
  Pt move (int dx)
    { ↑ new Pt (this.x + dx); }
  Pt bump ( )
    { ↑ this.move (1); }
}
class Spt extends Pt {
  int scale;
  Spt (int x, int s)
    { super(x); this.scale = s; }
  Pt move (int dx)
    { ↑ new Spt (this.x + this.scale * dx,
                 this.scale); }
  Spt zoom (int s)
  { ↑ new Spt (this.x, this.scale * s); }
}
```

Object layout = vtable + fields



Self-application using recursive records

$$\begin{array}{l}
\textbf{let}\ \mathsf{vtab_pt} = \{\mathsf{move} = \lambda(\mathsf{self}, \mathsf{dx}).\ _,\\
\qquad\qquad\qquad\qquad \mathsf{bump} = \lambda\mathsf{self}.\ _\}\\
\textbf{let}\ \mathsf{p} : \tau_{\mathsf{pt}} = \{\mathsf{vtab} = \mathsf{vtab_pt},\ \mathsf{x} = 3\}\\
(\textbf{unfold}\ \mathsf{p}).\mathsf{vtab}.\mathsf{bump}(\mathsf{p})
\end{array}$$

$$\begin{array}{l}
\tau_{\mathsf{pt}} = \mu\alpha.\ \{\mathsf{vtab} : \{\mathsf{move} : (\alpha, \mathsf{int}) \to \tau_{\mathsf{pt}},\\
\qquad\qquad\qquad\qquad \mathsf{bump} : \alpha \to \tau_{\mathsf{pt}}\},\\
\qquad\qquad\quad \mathsf{x} : \mathsf{int}\}
\end{array}$$

[Cardelli 84] [Reddy 88]

Core FLINT

Terms e	Types τ, σ	Kinds κ	
$\lambda x : \tau . e$ $e(e')$	$\tau \to \sigma$	Type	$\lambda^{\to}$
$\Lambda \alpha :: \kappa . e$ $e[\tau]$	$\forall \alpha :: \kappa . \tau$		F_2
	$\lambda \alpha :: \kappa . \tau$ $\tau\ \sigma$	$\kappa \to \kappa'$	F_ω
	$\{\!\{\overline{l = \tau}\}\!\}$ $\tau \cdot l$	$\{\!\{\overline{l :: \kappa}\}\!\}$	Type tuple
$\{\overline{l = e}\}$ $e.l$	$\{\overline{l : \tau}\}$		Record
$\mathbf{fold}_\tau\ e$ $\mathbf{unfold}_\tau\ e$	$\mu \alpha . \tau$		Iso-recursive

Core FLINT + rows + existentials

Terms e	Types τ, σ	Kinds κ	
$\lambda x:\tau.e$ $e(e')$	$\tau \to \sigma$	Type	$\lambda^{\to}$
$\Lambda\alpha::\kappa.e$ $e[\tau]$	$\forall\,\alpha::\kappa.\tau$		F_2
	$\lambda\alpha::\kappa.\tau$ $\tau\ \sigma$	$\kappa \to \kappa'$	F_ω
	$\{\{\overline{l = \tau}\}\}$ $\tau \cdot l$	$\{\{\overline{l :: \kappa}\}\}$	Type tuple
$\{\overline{l = e}\}$ $e.l$	$\{\tau\}$ Abs^L $l:\tau\,;\,\sigma$	R^L	Record Row [Rémy]
$\mathbf{fold}_\tau\ e$ $\mathbf{unfold}_\tau\ e$	$\mu\alpha.\tau$		Iso-recursive
$\langle\alpha = \sigma,\ e:\tau\rangle$ $\mathbf{open}\ \langle\alpha, x\rangle = e\ \mathbf{in}\ e'$	$\exists\alpha::\kappa.\tau$		Existential [M&P]

Core FLINT + mutually recursive types

Terms e	Types τ, σ	Kinds κ	
$\lambda x:\tau.e$ $e(e')$	$\tau \to \sigma$	Type	$\lambda^{\to}$
$\Lambda\alpha::\kappa.e$ $e[\tau]$	$\forall\,\alpha::\kappa.\tau$		F_2
	$\lambda\alpha::\kappa.\tau$ $\tau\ \sigma$	$\kappa \to \kappa'$	F_ω
	$\{\{\overline{l = \tau}\}\}$ $\tau \cdot l$	$\{\{\overline{l :: \kappa}\}\}$	Type tuple
$\{\overline{l = e}\}$	$\{\tau\}$		Record
$e.l$	Abs^L $l:\tau\,;\,\sigma$	R^L	Row [Rémy]
fold$_\tau$ e **at** l **unfold**$_\tau$ e **at** l	$\mu\alpha::\kappa.\tau$		Mutual Rec.
$\langle\alpha = \sigma,\ e:\tau\rangle$ **open** $\langle\alpha, x\rangle = e$ **in** e'	$\exists\alpha::\kappa.\tau$		Existential [M&P]

Records using row types

$\kappa ::= \ldots \mid \mathsf{R}^L$

$\tau ::= \ldots \mid \{\tau\} \mid \mathsf{Abs}^L \mid l : \tau \,;\, \sigma$

$$\frac{}{\Delta \vdash \mathsf{Abs}^L :: \mathsf{R}^L}$$

$$\frac{\Delta \vdash \tau :: \mathsf{R}^{\varnothing}}{\Delta \vdash \{\tau\} :: \mathsf{Type}}$$

$$\frac{\Delta \vdash \tau :: \mathsf{Type} \qquad \Delta \vdash \sigma :: \mathsf{R}^{L \cup \{l\}}}{\Delta \vdash l : \tau \,;\, \sigma :: \mathsf{R}^{L - \{l\}}}$$

Derived record type

$$\{l : \tau, m : \sigma\} \equiv \{l : \tau \,;\, m : \sigma \,;\, \mathsf{Abs}^{\{l,m\}}\}$$

Example

$$\begin{aligned} f &: \forall \rho :: \mathsf{R}^{\{l,m\}} . \{l : \tau \,;\, m : \sigma \,;\, \rho\} \to \tau \\ f[\mathsf{Abs}^{\{l,m\}}] &: \{l : \tau, m : \sigma\} \to \tau \\ f[n : \mathsf{int} \,;\, \mathsf{Abs}^{\{l,m,n\}}] &: \{l : \tau, m : \sigma, n : \mathsf{int}\} \to \tau \end{aligned}$$

Polymorphism without subsumption

let p : τ_{pt} = —
let sp : τ_{spt} = —
let f = $\lambda x : \tau_{pt}$. —
f(p)
f(sp) ✘

$$\tau_{pt} = \mu\alpha.\{\text{vtab} : \{\text{move} : (\alpha, \text{int}) \to \tau_{pt}, \text{bump} : \alpha \to \tau_{pt}\}, x : \text{int}\}$$

$$\tau_{spt} = \mu\alpha.\{\text{vtab} : \{\text{move} : (\alpha, \text{int}) \to \tau_{pt}, \text{bump} : \alpha \to \tau_{pt}, \text{zoom} : (\alpha, \text{int}) \to \tau_{spt}\}, x : \text{int}, \text{scale} : \text{int}\}$$

Polymorphism using existential rows

```
let p : τpt = —
let sp : τspt = —
let f = λx : τpt. —
f(p)
f(upcast[Spt, Pt, sp])
```

$$\tau_{pt} = \exists\{\!\{\rho :: R^{\{vtab,x\}},\ \delta :: Type \to R^{\{m,b\}}\}\!\}\ \mu\alpha.\{vtab : \{move : (\alpha, int) \to \tau_{pt}\ ;\ bump : \alpha \to \tau_{pt}\ ;\ \underline{\delta\ \alpha}\ \}\ ;\ x : int\ ;\ \underline{\rho}\ \}$$

$$\tau_{spt} = \exists\{\!\{\rho :: R^{\{vtab,x,s\}},\ \delta :: Type \to R^{\{m,b,z\}}\}\!\}\ \mu\alpha.\{vtab : \{move : (\alpha, int) \to \tau_{pt}\ ;\ bump : \alpha \to \tau_{pt}\ ;\ zoom : (\alpha, int) \to \tau_{spt}\ ;\ \underline{\delta\ \alpha}\ \}\ ;\ x : int\ ;\ scale : int\ ;\ \underline{\rho}\ \}$$

Recursive class references

$$
\begin{aligned}
\tau_{\mathsf{pt}} = \mu\beta.\ & \\
& \exists\{\!\{\rho :: \mathsf{R}^{\{\mathsf{vtab},\mathsf{x}\}},\ \delta :: \mathsf{Type}\to\mathsf{R}^{\{\mathsf{m},\mathsf{b}\}}\}\!\} \\
& \mu\alpha.\{\mathsf{vtab} : \{\mathsf{move} : (\alpha,\mathsf{int})\to\underline{\beta} \\
& \qquad\qquad ;\ \mathsf{bump} : \alpha\to\underline{\beta} \\
& \qquad\qquad ;\ \delta\ \alpha\ \} \\
& \qquad ;\ \mathsf{x} : \mathsf{int} \\
& \qquad ;\ \rho\ \}
\end{aligned}
$$

$$
\begin{aligned}
\tau_{\mathsf{spt}} = \mu\beta.\ & \\
& \exists\{\!\{\rho :: \mathsf{R}^{\{\mathsf{vtab},\mathsf{x},\mathsf{s}\}},\ \delta :: \mathsf{Type}\to\mathsf{R}^{\{\mathsf{m},\mathsf{b},\mathsf{z}\}}\}\!\} \\
& \mu\alpha.\{\mathsf{vtab} : \{\mathsf{move} : (\alpha,\mathsf{int})\to\tau_{\mathsf{pt}} \\
& \qquad\qquad ;\ \mathsf{bump} : \alpha\to\tau_{\mathsf{pt}} \\
& \qquad\qquad ;\ \mathsf{zoom} : (\alpha,\mathsf{int})\to\underline{\beta} \\
& \qquad\qquad ;\ \delta\ \alpha\ \} \\
& \qquad ;\ \mathsf{x} : \mathsf{int} \\
& \qquad ;\ \mathsf{scale} : \mathsf{int} \\
& \qquad ;\ \rho\ \}
\end{aligned}
$$

Mutually recursive types

$$\begin{aligned} \tau_{AB} \quad &:: \quad \{\!\{ A :: \mathsf{Type},\ B :: \mathsf{Type} \}\!\} \\ &= \quad \mu\gamma :: \{\!\{ A :: \mathsf{Type},\ B :: \mathsf{Type} \}\!\}.\ \{\!\{ A = \tau_A,\ B = \tau_B \}\!\} \end{aligned}$$

$$\frac{\Delta;\Gamma \vdash e \ : \ \tau_A}{\Delta;\Gamma \vdash \mathbf{fold}_{\tau_{AB}}\ e\ \mathbf{at}\ A \ : \ \tau_{AB}\cdot A}$$

$$\frac{\Delta;\Gamma \vdash e \ : \ \tau_{AB}\cdot B}{\Delta;\Gamma \vdash \mathbf{unfold}_{\tau_{AB}}\ e\ \mathbf{at}\ B \ : \ \tau_B}$$

Mutually recursive class references

```
class A {
    B f (A x) {—}
}
class B extends A {
    B f (A x) {—}
    int g () {—}
}
```

$$\tau_{AB} = \mu\gamma{::}\{\!\{A :: \mathsf{Type},\ B :: \mathsf{Type}\}\!\}.$$
$$\{\!\{A = \exists\delta{::}\mathsf{Type}\to R^{\{f\}}.\,\mu\alpha.\,\{f : (\alpha, \gamma\cdot A)\to\gamma\cdot B;\ \delta\ \alpha\},$$
$$B = \exists\delta{::}\mathsf{Type}\to R^{\{f,g\}}.\,\mu\alpha.\,\{f : (\alpha, \gamma\cdot A)\to\gamma\cdot B;$$
$$g : \alpha\to\mathsf{int};\ \delta\ \alpha\}\ \}\!\}$$

Each class declaration ...

is a separately-compiled module
which imports:

- other classes

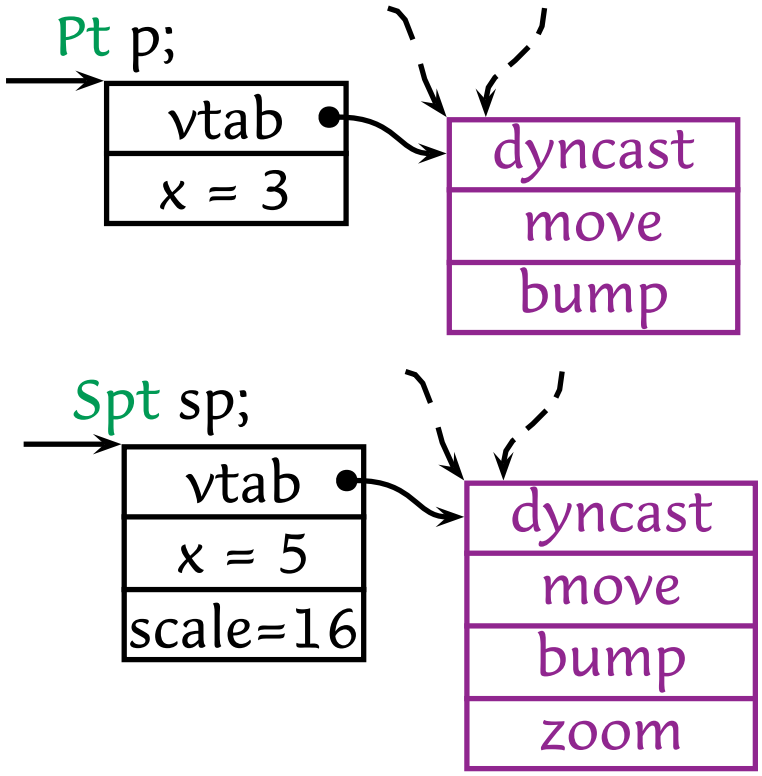
and exports:

- dictionary
- constructor
- tag

$$CL ::= \text{class C extends C } \{\overline{\text{C f;}}\ K\ \overline{M}\}$$

$$e ::= x \mid e.\text{f} \mid e.\text{m}(\overline{e})$$
$$\mid \text{new C}(\overline{e}) \mid (\text{C})e$$

Dynamic cast—observations



1. vtable identifies dynamic class

2. extensible sum can provide one tag per class

datatype Tagged =
T_{pt} **of** τ_{pt}
| T_{spt} **of** τ_{spt}
| ...

dyncast : $\forall\,\alpha.\,(\text{Tagged} \rightarrow \textbf{maybe}\ \alpha) \rightarrow \textbf{maybe}\ \alpha$

Dynamic cast—solution

Implementation

$\Lambda\alpha.\,\lambda(\text{self},\ \text{proj} : \text{Tagged}\to\textbf{maybe}\ \alpha).$

case proj$(T_C(\text{self}))$

of some x $\Rightarrow$ **some** x

| **none** $\Rightarrow$ super.dyncast[α](self, proj)

Invocation

let $\text{proj}_{C'} = \lambda t.$ **case** t **of** $T_{C'}\ x$ $\Rightarrow$ **some** x | _ $\Rightarrow$ **none**

p.vtab.dyncast[$\tau_{C'}$](p, $\text{proj}_{C'}$)

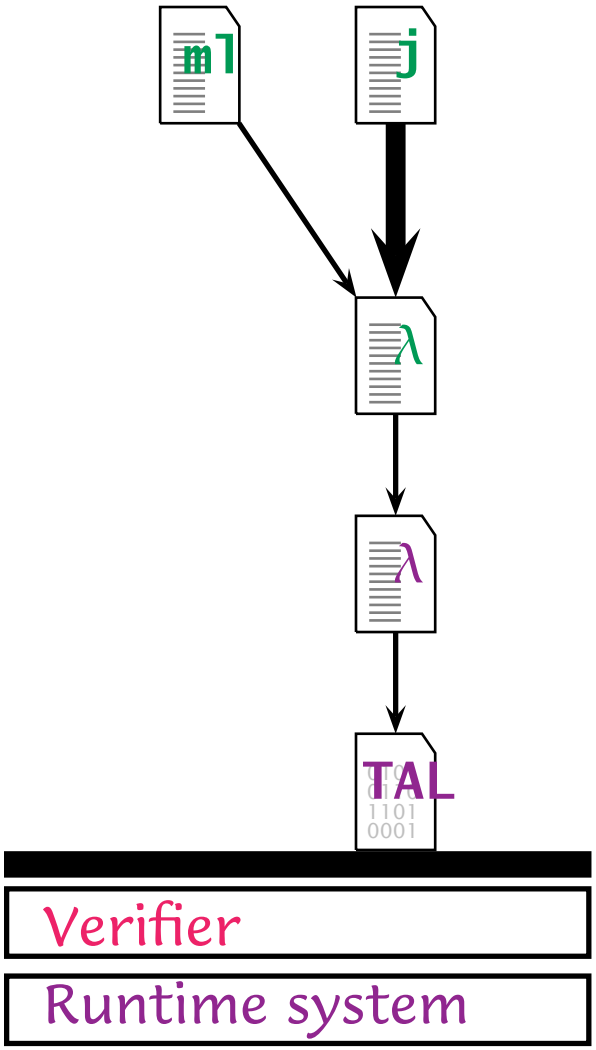
Comparing object encodings (Round II)

[PT 94] [HP 95]	$\exists\alpha.$	$\alpha \times (I\ \alpha\ \alpha)$
[Bruce 94]	$\mu\beta.\ \exists\alpha.$	$\alpha \times (I\ \beta\ \alpha)$
[ACV 96]	$\mu\beta.\ \exists\alpha \le \beta.$	$\alpha \times (I\ \alpha\ \alpha)$
[Crary 99]	$\mu\beta.\ \exists\alpha.$	$\alpha \wedge (I\ \beta\ \alpha)$
[Glew 00]	$\mu\beta.\ \exists\alpha \le (I\ \beta\ \alpha).$	α
	$\mu\beta.\ \exists\delta \le (I\ \beta).$	$\mu\,\delta$
[LST]	$\mu\beta.\ \exists\delta :: \mathsf{Type}\to\mathsf{R}^{m(I)}.$	$\mu\ (I'\ \beta\ \delta)$

where, for example,

$$I_P = \lambda\beta.\lambda\alpha.\{\mathsf{getx}:\alpha\to\mathsf{int},\ \mathsf{eq}:\alpha\to\beta\to\mathsf{bool}\}$$
$$I'_P = \lambda\beta.\lambda\delta{::}\mathsf{Type}\to\mathsf{R}^{\{\mathsf{getx},\mathsf{eq}\}}.\ \lambda\alpha.\{\mathsf{getx}:\alpha\to\mathsf{int};\ \mathsf{eq}:\alpha\to\beta\to\mathsf{bool};\ \delta\ \alpha\}$$

Conclusion



A simple, decidable typed IL can efficiently support both SML and a significant subset of Java.

- classes, inheritance, interfaces
- privacy, mutual recursion
- name equivalence, dynamic cast
- constructors, super, static

Implementation is in progress.

Dictionary is polymorphic ...

over additional fields and methods
in the self type.

$$
\begin{array}{l}
\textbf{let}\ \mathsf{pt\_dict} = \Lambda\{\!\{\rho :: \mathsf{R}^{\{\mathsf{vtab},\mathsf{x}\}},\ \delta :: \mathsf{Type}\to\mathsf{R}^{\{\mathsf{m},\mathsf{b}\}}\}\!\}. \\
\quad \{\mathsf{move} = \text{—}, \\
\quad\ \mathsf{bump} = \lambda \mathsf{self} : \mu\alpha.\ \{\mathsf{vtab} : \{\mathsf{move} : (\alpha, \mathsf{int})\to\tau_{\mathsf{pt}};\ \} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{bump} : \alpha\to\tau_{\mathsf{pt}};\ \delta\ \alpha\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \mathsf{x} : \mathsf{int};\ \rho\}. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\textbf{unfold}\ \mathsf{self}).\mathsf{vtab}.\mathsf{move}(\mathsf{self},\ 1)
\end{array}
$$