An OOP Typed Intermediate Language with Support to Generics and Interfaces

Chiyan Chen

Introduction

- Types facilitate static reasoning of various kinds of program invariants and provide static safety assurance for programs.
- Conventional compiler intermediate languages are untyped
  - Types are only preserved until the type checking phase.
  - Code translation and optimization are performed over the untyped intermediate language.

Typed Intermediate Language

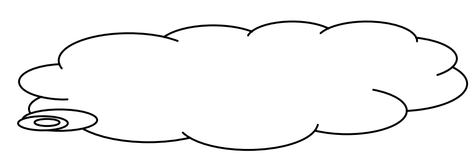
- Preserve types in the compiler intermediate representations
  - Help improving the correctness of compilers.
  - Facilitate type directed compilation. Typing information can be used to direct optimization.
  - Provide safety assurance on intermediate codes.

OOP

- Widely used in many realistic applications.
- Close to real world objects, intuitive in reasoning, easier for programming…
- Typed intermediate language for OOP is of particular interest.

Overview of Object Encoding

Most traditional object encodings are based on $F^{\omega}_{<:}$

Objects	⇒	Records
Method invacation	⇒	Field projection
Nominal subclassing	⇒	Structural subtyping
Nominal classes	⇒	
Dynamic type cast	⇒	?

Overview of Object Encoding

- Involve many advanced type features.
  - Impredicative bounded quantification
  - Higher order bounded quantification
  - F-bounded quantification
  - Equi-recursive types
  - Row polymorphism
  - Intersection types
- Decidability of type checking is always a concern.
- May incur runtime overheads.

Our approach

- Based On dependent types
  – A restricted form of dependent types (simple index types)
  – Predicative bounded quantification
  – No equi-recursive type needed
- Type checking is easy
- No runtime overhead involved

Dependent Types

- int: …, -2, -1, 0, 1, 2, …
- Refined int by "odd or even":
  - int(odd): …, -1, 1, 3, …
  - int(even): …, -2, 0, 2, …
- Refine int with "positive or negative":
  - int(+): 0, 1, 2, …
  - int(-): …, -3, -2, -1
- Type refinement
  - More refined types ➔
  - more information is known about the values ➔
  - more static reasoning is allowed

Dependent Types

- Typing objects
  - obj: o_1, o_2, …
- Refin the typing of objects
  - obj(c_1): o_1, …
  - obj(c_2): o_2, …

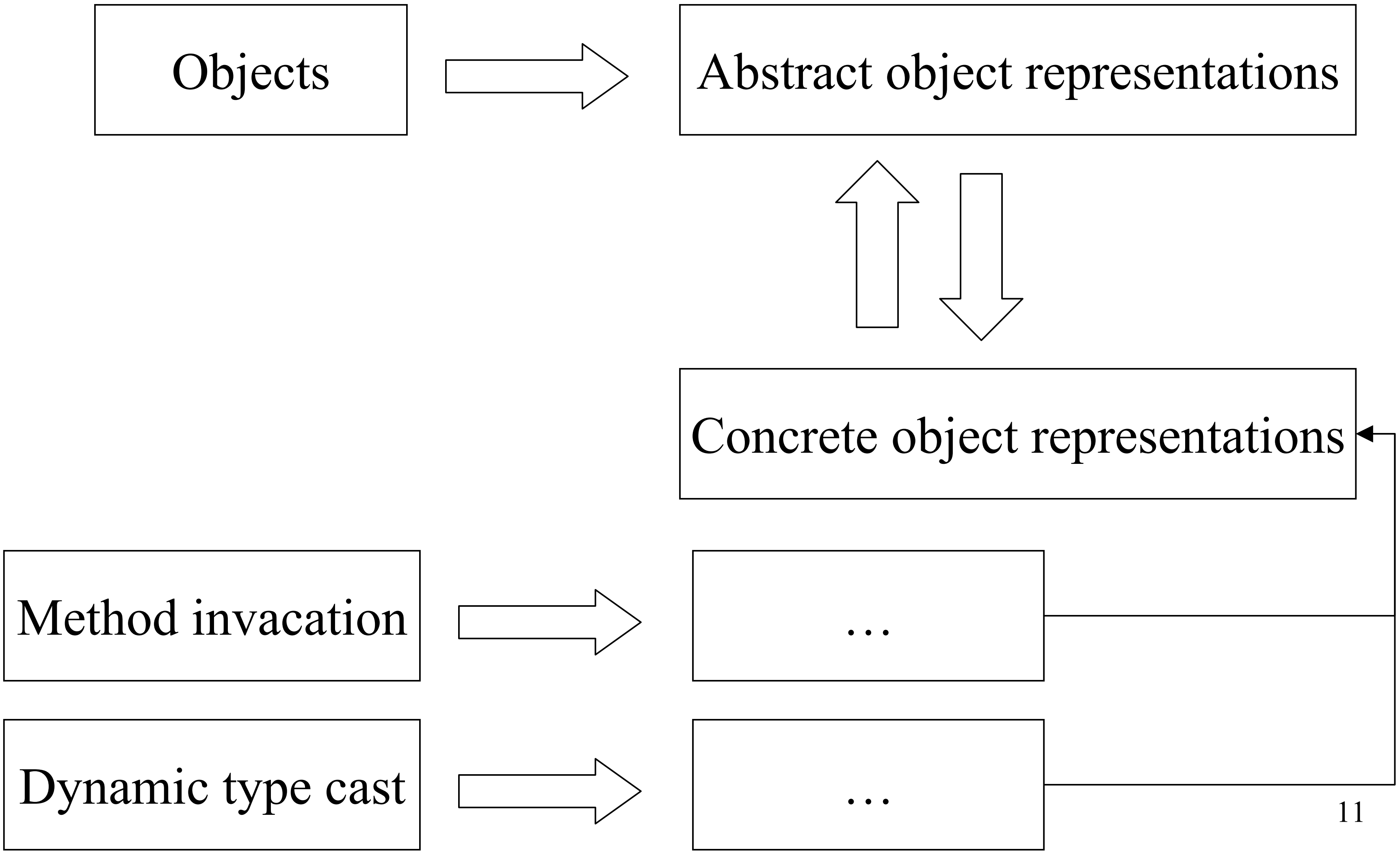
Our approach

Nominal classes → Static type indices

Nominal subclassing → An ordering on the indices

Totally separate from structural subtyping

Our approach



Highlighted Contribution

- Based on the previous work of LILc
  - Fit it into a standard dependent type framework
  - More general type refinement through constraint based typing.
  - Ease type checking and relieve the restriction on subtyping
- Add support of bounded generic classes
  - How to deal with static class indices and runtime class tags in the presence of generic classes
- Formalize the treatment of interfaces.

Statics of LILg

- σ ::= cls | *
- S ::= C: $\Pi \langle \alpha \mid P \rangle$. cls
  | $C(\alpha) \ll C'(c)$
- c ::= α | C(c)
- P ::= $\emptyset$ | $c \ll c'$ | $c = c'$ | P, P'
- τ ::= …| tag(c) | obj(c) | $\forall \langle \alpha{:}\ \sigma \mid P \rangle.\ \tau$
  | $\exists \langle \alpha{:}\ \sigma \mid P \rangle.\ \tau$ | $\{l_i{:}\ \tau_i\}$

C $\langle \alpha \ll A \rangle$ extends B {…}

C: $\Pi \langle \alpha \mid \alpha \ll A \rangle$. cls
$C(\alpha) \ll B$

Dynamics of LILg

- e ::= … | tag(C) (e) | (C(c)) e | c2r (c, e)

Runtime class tags

Coercion from concrete representation to abstract representation

Coercion from abstract representation to concrete representation

Typing of LILg

$$\frac{\Delta; P \vdash_{\mathcal{S}} C(c) : \Omega_c \quad \Delta; P; \Gamma \vdash_{\mathcal{S}} e : \mathsf{tag}(c)}{\Delta; P; \Gamma \vdash_{\mathcal{S}} \mathsf{tag}(C)(e) : \mathsf{tag}(C(c))}$$

$$\frac{\Delta; P; \Gamma \vdash_{\mathcal{S}} e : R(C(c))}{\Delta; P; \Gamma \vdash_{\mathcal{S}} (C(c))e : \mathsf{obj}(C(c))}$$

$$\frac{\Delta; P; \Gamma \vdash_{\mathcal{S}} e : \mathsf{obj}(c) \quad \Delta; P \models_{\mathcal{S}} c << C(c_0)}{\Delta; P; \Gamma \vdash_{\mathcal{S}} \mathsf{c2r}(C(c_0), e) : ApproxR(c, C(c_0))}$$

Reduction Semantics of LILg

- c2r(c_2, (c_1) v) is a redex. Its reduction is compc2r(c_2, (c_1) v), where compc2r is a meta operation to be defined accordingly to the concrete object representations.
- ……

Regularity Property

- Assume c_1 and c_2 are two class indices such that $\models_S c_1 << c_2$, and v is a value is such that $\vdash_S v: R(c_1)$ is derivable. Then $\vdash_S$ compc2r(c_2, (c_1) v): ApproxR(c_1, c_2) is derivable.

Type Soundness of LILg

- Type preservation
- Progress
- Well typed LILg programs do not go wong.

A Concrete Object Representation

$C \langle \alpha << A \rangle$ extends B {
  f: t_1
  m = (fun $\langle \beta << D \rangle$ (x: t_2): t_3 = e)
}

ApproxR (c, C(E)) =
  {vtable: Vtable(c, C(E)),
   f: $t_1[\alpha \to E]$
  }
R(C(E)) = Approx(C(E), C(E))

Vtable (c, C(E)) =
  {tag: tag(c),
   m: $(\forall \langle \beta: cls \mid \beta << D \rangle. (object(c), tag(\beta), t_2) \to t_3) [\alpha \to E]$
  }
object(c) = $\exists \langle \alpha: cls \mid \alpha << c \rangle. Obj(\alpha)$

An Concrete Object Representation

- Prop: If $\models_S c_1 << c_2$, then $\models_S R(c_1) \leq Approx(c_1, c_2)$.

- compc2r $(c_1, (c_2)\ v) \rightarrow v$

- The regularity property is satisfied.
  - Assume c_1 and c_2 are two class indices such that $\models_S c_1 << c_2$, and v is a value is such that $\vdash_S v: R(c_1)$ is derivable. Then $\vdash_S$ compc2r(c2, (c1) v): ApproxR(c_1, c_2) is derivable.

Translation from Source Language

- $\text{object}(c) = \exists \langle \alpha\text{: cls} \mid \alpha \ll c \rangle.\ \text{obj}(\alpha)$

- $|\alpha| = \text{object}\ (\alpha)$
- $|C(c)| = \text{object}\ (C(c))$
- $|\tau_1 \to \tau_2| = |\tau_1| \to |\tau_2|$
- $|\forall \langle \alpha \ll c \rangle.\ \tau| = \forall \langle \alpha\text{: cls} \mid \alpha \ll c \rangle.\ \text{tag}(\alpha) \to |\tau|$

Dictionary class tag parameters

Translation from Source Language

- Dynamic type cast

$$downcast = fun\langle \alpha : \Omega_c \rangle(d_\alpha : \mathsf{TAG}(\alpha), o : \mathsf{object}^+(\mathsf{Topc})) : \mathsf{object}^+(\alpha) =$$
$$\mathsf{ifNull}(o) \text{ then pack } \langle \mathsf{null}, \text{pack } \langle \alpha, \mathsf{null}(\alpha) \rangle \text{ as } \mathsf{object}(\alpha) \rangle \text{ as } \mathsf{object}^+(\alpha) \text{ else bind } x \text{ in}$$
$$\text{open } x \text{ as } (\alpha', y) \text{ in } loopdc[\alpha, \alpha', \alpha'](d_\alpha, (c2r(\mathsf{Topc}, y)).vtable.tag, y)$$

$$loopdc = fun\langle \alpha_1 : \Omega_c, \alpha_2 : \Omega_c, \alpha_3 : \Omega_c | \alpha_3 \ll \alpha_2 \rangle$$
$$(d_{\alpha_1} : \mathsf{TAG}(\alpha_1), d_{\alpha_2} : \mathsf{TAG}(\alpha_2), o : \mathsf{obj}(\alpha_3, \mathsf{notnull})) : \mathsf{object}^+(\alpha_1) =$$
$$\mathsf{ifEqTag}(d_{\alpha_1}, d_{\alpha_2}) \text{ then pack } \langle \mathsf{notnull}, \text{pack } \langle \alpha_3, o \rangle \text{ as } \mathsf{object}(\alpha_1) \rangle \text{ as } \mathsf{object}^+(\alpha_1) \text{ else}$$
$$\mathsf{ifParent}(d_{\alpha_2}) \text{ then bind } (\alpha_4, d_{\alpha_4}) \text{ in } loopdc[\alpha_1, \alpha_4, \alpha_3](d_{\alpha_1}, d_{\alpha_4}, o) \text{ else } \bot$$

Translation from Source Language

- Type directed translation
  - Both type the source program and translate it.
  - $\Delta; P; \Gamma \vdash e: \tau \Rightarrow e^*$
- Details in technical report.

Translation Preserves Typing

- If $\Delta; P; \Gamma \vdash_S e: \tau \Rightarrow e^*$ is derivable, the so is $\Delta; P; \Gamma^* \vdash e^*: |\tau|$, where $\Delta; P \vdash \Gamma \Rightarrow \Gamma^*$.

$$\frac{}{\alpha_1, ..., \alpha_n; P \vdash_{\mathcal{S}} \emptyset \Rightarrow d_{\alpha_1} : \mathsf{tag}(\alpha_1), ..., d_{\alpha_n} : \mathsf{tag}(\alpha_n)}$$

$$\frac{\Delta; P \vdash_{\mathcal{S}} \Gamma \Rightarrow \Gamma^*}{\Delta; P \vdash_{\mathcal{S}} \Gamma, x : \tau \Rightarrow \Gamma^*, x : |\tau|}$$

Arrays

- Practically, A << B implies array(A) $\leq$ array(B). However, this breaks type soundness.
  - Well know examples
- Bridge the gap
  - $|\text{array}(c)| = \exists \langle \alpha: \text{cls} \mid \alpha \ll c \rangle. \{\text{tag}: \text{tag}(\alpha), \text{arr}: \text{array}(\text{object}(\alpha))\}$
  - A << B implies $|\text{array}(A)| \leq |\text{array}(B)|$
  - Dynamic type cast is involved in array element update, which patches up the potential unsoundness.

Interfaces

- τ ::= … | interface(c, i)
- e ::= … | (I(c)) e | i2r(e)

$$\frac{\Delta; P \models_{\mathcal{S}} c \ll I(\bar{t}) \quad \Sigma; \Delta; P; \Gamma \vdash_{\mathcal{S}} e : R_I(c, I(\bar{t}))}{\Sigma; \Delta; P; \Gamma \vdash_{\mathcal{S}} (I(\bar{t}))e : \mathsf{interface}(c, I(\bar{t}))}$$

$$\frac{\Sigma; \Delta; P; \Gamma \vdash_{\mathcal{S}} e : \mathsf{interface}(c, I(\bar{t}))}{\Sigma; \Delta; P; \Gamma \vdash_{\mathcal{S}} \mathsf{i2r}(e) : R_I(c, I(\bar{t}))}$$

Interfaces

$I\ \langle\ \alpha << A\ \rangle$ extends J {

  $m : \forall\ \langle\ \beta << D\ \rangle.\ t_1 \rightarrow t_2$

}

$R_I\ (c, I(B)) = \{$

  $m: (\forall\ \langle\ \beta\text{: cls} \mid \beta << D\ \rangle.\ (\text{object}(c), \text{tag}(\beta), t_1) \rightarrow t_2)\ [\alpha \rightarrow B]$

}

Interfaces

Vtable (c, C(E)) =

{tag: tag(c),

m: …

itable: array(ientry(c))

}

ientry(c) = $\exists \langle \alpha$: cls | c << $\alpha \rangle$. {tag: tag(α), intf: interface(c, α)}

Interfaces

$$o2i = fun\langle\alpha_i : \Omega_c, \alpha_c : \Omega_c\rangle(d_{\alpha_i} : \mathsf{TAG}(\alpha_i), o : \mathsf{obj}(\alpha_c, \mathsf{notnull}))$$

$$: \{obj^I : \mathsf{object}(\alpha_i), intf^I : \mathsf{interface}(\alpha_c, \alpha_i)\} =$$

$$\mathsf{let}\ itbl : Itable(\alpha_c) = \mathsf{c2r}(\mathsf{Topc}, o).vtable.itable\ \mathsf{in}$$

$$\mathsf{let}\ itbllen : \mathsf{int} = arraylength(itbl)\ \mathsf{in}$$

$$loopo2i[\alpha_i, \alpha_c](itbllen, d_{\alpha_i}, o, itbl)$$

$$loopo2i = fun\langle\alpha_i : \Omega_c, \alpha_c : \Omega_c\rangle(k : \mathsf{int}, d_{\alpha_i} : \mathsf{TAG}(\alpha_i), o : \mathsf{obj}(\alpha_c, \mathsf{notnull}), itbl : Itable(\alpha_c))$$

$$: \{obj^I : \mathsf{object}(\alpha_i), intf^I : \mathsf{interface}(\alpha_c, \alpha_i)\} =$$

$$\mathsf{if}\ j = 0\ \mathsf{then}\ \bot\ \mathsf{else}$$

$$\mathsf{open}\ itbl[j-1]\ \mathsf{as}\ (\alpha, x)\ \mathsf{in}$$

$$\mathsf{ifEqTag}(d_{\alpha_i}, x.tag)\ \mathsf{then}\ \mathsf{new}\{obj = \mathsf{pack}\ \langle\alpha_c, o\rangle\ \mathsf{as}\ \mathsf{object}(\alpha_i), intf = x.intf\}$$

$$\mathsf{else}\ loopo2i[\alpha_i, \alpha_c](j-1, d_{\alpha_i}, itbl)$$

Two Approaches for Interface Translation

- Casting is no-op, interface method invocation requires a linear lookup of itable
  - $|I| = \text{object}(I)$
- Casting requires a linear lookup of itable, interface method invocation is simple record field projection.
  - $|I| = \exists \langle \alpha : \text{cls} \mid \alpha << I \rangle .\ \{\text{obj}: \text{object}(\alpha), \text{intf}: \text{interface}(\alpha, I)\}$

More contraints

- $P ::= \ldots \mid \bot \mid P \vee P' \mid \neg P$
- $\exists \langle \alpha: cls \mid \alpha = A \vee \alpha = B \rangle.\ obj(\alpha)$
- $\exists \langle \alpha: cls \mid \alpha << C, \neg (\alpha = D) \rangle.\ Obj(\alpha)$

Thanks to the lunch discussion with Bjarne and Juan for the inspiration.

Binary Methods

- Examples: equality comparison, list concatenation.
- $\tau ::= \text{thisclass} \mid @\tau$ — Exact type
- $|@\alpha| = obj(\alpha)$
- $|@C(c)| = obj(C(c))$
- Binary methods can only be invoked on object of exact type.

Binary Methods

```
C {
   m_1 = (fun (x: thisclass): t = e)
   m_2 = (fun (x: @thisclass): t = e)
}
```

The regularity property is preserved.

Vtable (c, C) =

{tag: tag(c),

m_1: (object(c), object(c))→ t)

m_1: (object(c), obj(c))→ t)

}

object(c) = ∃ ⟨ α: cls | α << c ⟩. Obj(α)

Other Concrete Object Representation

- Objects as functions (Xi, Chen and Chen 03)
- Multiple Inheritance (Chen, Shi and Xi 04)

Related Work

- Generics for .NET CLR (Yu et.al 04)
  - The target language remains at a high level, where method invocation and casting are primitives.
  - No bounded quantification for generics, no interfaces.
- Translating Pizza to Java (Odersky and Wadler 97)
  - Static type safety is lost in the translation.

Related Work

- An efficient class and object encoding (Glew 00)
  - F-bounded existentially quantified type + equi-recursive type
  - No generics, no interfaces
- Type preserving compilation of FJ (League et.al 02)
  - Row polymorphism + equi-recursive type + higher order quantification
  - No generics
- Simple, efficient object encoding using intersection types (Crary 99)
  - Intersection types + equi-recursive type
  - No generics, no interfaces

Related Work

- LILc (Chen and Tarditi 04)
  - Essentially employs dependent types for object encoding, however with a restrict form of type refinement which may potentially limit further extensions.
  - Does not deal with generic, no formalization for interfaces.
- Objects as functions (Xi et.al 03, Chen et.al 04)
  - Employ dependent types in object encoding (as functions)
  - Deal with generics and multiple inheritance
  - Not clear how to be deployed in realistic compliers.
- Applied Types Systems (ATS) (Xi 04)
  - A general framework for a wide range of highly expressive type systems.

Conclusion

- A simple, general and efficient typed intermediate language for OOP through the techniques of dependent types.
- First to formally addresses bounded generic classes and interfaces.
- Close to realistic implementation of compilers. Can be expected to be deployed in practical applications.
- Don’t be afraid of dependent types, they are our friend and they help!

Future Work

- Formal discussion of decidability of type checking
  - Fully exploit the typing of source programs in the type directed translation to annotate the target program
  - Easily decidable if every subexpression is annotated. But do we need such extensive annotation?
- Implementation in Bartok