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 $\text{F}^{\omega}_{<:}$:

- Objects $\leftrightarrow$ Records
- Method invocation $\rightarrow$ Field projection
- Nominal subclassing $\rightarrow$ Structural subtyping
- Nominal classes $\rightarrow$ ?
- Dynamic type cast $\rightarrow$ ?
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: o₁, o₂, …

• Refin the typing of objects
  – obj(c₁): o₁, …
  – obj(c₂): o₂, …
Our approach

Nominal classes $\rightarrow$ Static type indices

Nominal subclassing $\rightarrow$ An ordering on the indices

Totally separate from structural subtyping
Our approach

Objects → Abstract object representations

Concrete object representations

Method invocation

Dynamic type cast

…

…
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

- \( \sigma ::= \text{cls} | * \)
- \( S ::= C : \Pi \langle \alpha \mid P \rangle . \text{cls} \)
  \[ \mid C(\alpha) \ll C'(c) \]
- \( c ::= \alpha \mid C(c) \)
- \( P ::= \emptyset \mid c \ll c' \mid c = c' \mid P, P' \)
- \( \tau ::= \ldots \mid \text{tag}(c) \mid \text{obj}(c) \mid \forall \langle \alpha : \sigma \mid P \rangle . \tau \)
  \[ \mid \exists \langle \alpha : \sigma \mid P \rangle . \tau \mid \{l_i : \tau_i\} \]
Dynamics of LILg

- $e ::= \ldots | \text{tag}(C) \ (e) | (C(c)) \ e | c2r \ (c, \ e)$

- Runtime class tags
- Coercion from concrete representation to abstract representation
- Coercion from abstract representation to concrete representation
Typing of LILg

\[ \Delta; P \vdash_S C(c) : \Omega_c \quad \Delta; P; \Gamma \vdash_S e : \text{tag}(c) \]
\[ \Delta; P; \Gamma \vdash_S \text{tag}(C)(e) : \text{tag}(C(c)) \]

\[ \Delta; P; \Gamma \vdash_S e : R(C(c)) \]
\[ \Delta; P; \Gamma \vdash_S (C(c))e : \text{obj}(C(c)) \]

\[ \Delta; P; \Gamma \vdash_S e : \text{obj}(c) \quad \Delta; P \models_S c \ll C(c_0) \]
\[ \Delta; P; \Gamma \vdash_S c2r(C(c_0), e) : \text{Approx}R(c, C(c_0)) \]
Reduction Semantics of LILg

• \(c2r(c_2, (c_1) v)\) is a redex. Its reduction is \(\text{compc2r}(c_2, (c_1) v)\), where \(\text{compc2r}\) 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 \( S c_1 << c_2 \), and \( v \) is a value is such that \( S v: R(c_1) \) is derivable. Then \( S \) \( \text{compc2r}(c_2, (c_1) v): \text{ApproxR}(c_1, c_2) \) is derivable.
Type Soundness of LILg

- Type preservation
- Progress
- Well typed LILg programs do not go wrong.
A Concrete Object Representation

```
C ⟨ α << A ⟩ extends B {
    f: t₁
    m = (fun ⟨ β << D ⟩ (x: t₂): t₃ = e)
}
```

```
ApproxR (c, C(E)) =
{vtable: Vtable(c, C(E)),
    f: t₁[α → E]
}
R(C(E)) = Approx(C(E), C(E))
```

```
Vtable (c, C(E)) =
{tag: tag(c ),
    m: (∀ ⟨ β: cls | β << D ⟩. (object(c), tag(β), t₂→ t₃)[α → E]
}
object(c) = ∃ ⟨ α: cls | α << c ⟩. Obj(α)
```
An Concrete Object Representation

- Prop: If \( \models_S c_1 \ll c_2 \), then \( \models_S R(c_1) \leq \text{Approx}(c_1, c_2) \).

- \( \text{comp}_{c_2 r}(c_1, (c_2) v) \rightarrow v \)

- The regularity property is satisfied.
  - Assume \( c_1 \) and \( c_2 \) are two class indices such that \( \models_S c_1 \ll c_2 \), and \( v \) is a value such that \( \models_S v: R(c_1) \) is derivable. Then \( \models_S \text{comp}_{c_2 r}(c_2, (c_1) v): \text{Approx}_R(c_1, c_2) \) is derivable.
Translation from Source Language

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

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

Dictionary class

tag parameters
Translation from Source Language

- Dynamic type cast

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

\[
\text{loopdc} = \text{fun}\langle \alpha_1 : \Omega_c, \alpha_2 : \Omega_c, \alpha_3 : \Omega_c \mid \alpha_3 \ll \alpha_2 \rangle \\
\hspace{1cm} (d_{\alpha_1} : \text{TAG}(\alpha_1), d_{\alpha_2} : \text{TAG}(\alpha_2), o : \text{obj}(\alpha_3, \text{notnull})) : \text{object}^{+}(\alpha_1) = \begin{align*}
&\text{ifEqTag}(d_{\alpha_1}, d_{\alpha_2}) \text{ then pack } \langle \text{notnull}, \text{pack } \langle \alpha_3, o \rangle \text{ as object}(\alpha_1) \rangle \text{ as object}^{+}(\alpha_1) \text{ else} \\
&\text{ifParent}(d_{\alpha_2}) \text{ then bind } (\alpha_4, d_{\alpha_4}) \text{ in } \text{loopdc}[\alpha_1, \alpha_4, \alpha_3](d_{\alpha_1}, d_{\alpha_4}, o) \text{ else } \bot
\end{align*}
\]
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 $\Delta; P; \Gamma \vdash_S e : \tau \Rightarrow e^* \text{ is derivable, then so is } \Delta; P; \Gamma^* \vdash e^*: |\tau|$, where $\Delta; P \vdash \Gamma \Rightarrow \Gamma^*$.

\[
\alpha_1, \ldots, \alpha_n; P \vdash_S \emptyset \Rightarrow d_{\alpha_1} : \text{tag}(\alpha_1), \ldots, d_{\alpha_n} : \text{tag}(\alpha_n)
\]

\[
\Delta; P \vdash_S \Gamma \Rightarrow \Gamma^*
\]

\[
\Delta; P \vdash_S \Gamma, x : \tau \Rightarrow \Gamma^*, x : |\tau|
\]
Arrays

• Practically, $A \ll B$ implies $\text{array}(A) \leq \text{array}(B)$. However, this breaks type soundness.
  – Well know examples

• Bridge the gap
  – $|\text{array}(c)| = \exists \langle \alpha : \text{cls} \mid \alpha \ll c \rangle. \{\text{tag: tag}(\alpha), \text{arr: } \text{array}(\text{object}(\alpha))\}$
  – $A \ll B$ implies $|\text{array}(A)| \leq |\text{array}(B)|$
  – Dynamic type cast is involved in array element update, which patches up the potential unsoundness.
Interfaces

- $\tau ::= \ldots \mid \text{interface}(c, i)$
- $e ::= \ldots \mid (I(c)) \; e \mid \text{i2r}(e)$

$$
\begin{align*}
\Delta; P \models_S c \ll I(t) & \quad \Sigma; \Delta; P; \Gamma \vdash_S e : R_I(c, I(t)) \\
\Sigma; \Delta; P; \Gamma \vdash_S (I(t))e : \text{interface}(c, I(t)) \\
\Sigma; \Delta; P; \Gamma \vdash_S e : \text{interface}(c, I(t)) \\
\Sigma; \Delta; P; \Gamma \vdash_S \text{i2r}(e) : R_I(c, I(t))
\end{align*}
$$
Interfaces

\[
I \langle \alpha \ll A \rangle \text{ extends } J \{ \\
\quad m : \forall \langle \beta \ll D \rangle. t_1 \rightarrow t_2 \\
\}
\]

\[
R_1 (c, I(B)) = \{ \\
\quad m : (\forall \langle \beta : \text{cls} | \beta \ll D \rangle. \text{object(c), tag(\beta), } t_1 \rightarrow t_2) [\alpha \rightarrow B] \\
\}
\]
Interfaces

\[
\text{Vtable } (c, C(E)) = \\
\{\text{tag: tag}(c), \\
m: \ldots \\
\text{itable: array}(\text{ientry}(c))\}
\]

\[
\text{ientry}(c) = \exists \langle \alpha: \text{cls} \mid c \ll \alpha \rangle. \{\text{tag: tag}(\alpha), \text{intf: interface}(c, \alpha)\}
\]
Interfaces

\[ o2i = \text{fun}\{\alpha_i : \Omega_c, \alpha_c : \Omega_c\}(d_{\alpha_i} : \text{TAG}(\alpha_i), o : \text{obj}(\alpha_c, \text{nonnull})) \]
\[ : \{\text{obj}^I : \text{object}(\alpha_i), \text{intf}^I : \text{interface}(\alpha_c, \alpha_i)\} = \]
\[ \text{let } \text{itbl} : \text{Itable}(\alpha_c) = \text{c2r(Topc, o).vtable}.\text{itable} \text{ in} \]
\[ \text{let } \text{itblen} : \text{int} = \text{arraylength(itbl)} \text{ in} \]
\[ \text{loopo2i}[\alpha_i, \alpha_c](\text{itblen}, d_{\alpha_i}, o, \text{itbl}) \]

\[ \text{loopo2i} = \text{fun}\{\alpha_i : \Omega_c, \alpha_c : \Omega_c\}(k : \text{int}, d_{\alpha_i} : \text{TAG}(\alpha_i), o : \text{obj}(\alpha_c, \text{nonnull}), \text{itbl} : \text{Itable}(\alpha_c)) \]
\[ : \{\text{obj}^I : \text{object}(\alpha_i), \text{intf}^I : \text{interface}(\alpha_c, \alpha_i)\} = \]
\[ \text{if } j = 0 \text{ then } \perp \text{ else} \]
\[ \text{open } \text{itbl}[j - 1] \text{ as } (\alpha, x) \text{ in} \]
\[ \text{ifEqTag}(d_{\alpha_i}, x.\text{tag}) \text{ then new}\{\text{obj} = \text{pack } \langle \alpha_c, o \rangle \text{ as object}(\alpha_i), \text{intf} = x.\text{intf}\} \]
\[ \text{else } \text{loopo2i}[\alpha_i, \alpha_c](j - 1, d_{\alpha_i}, \text{itbl}) \]
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.
  – $|I| = \exists \langle \alpha: \text{cls} | \alpha \ll I \rangle. \{\text{obj: object}(\alpha), \text{intf: interface}(\alpha, I)\}$
More contraints

- $P ::= \ldots \mid \bot \mid P \lor P' \mid \neg P$
- $\exists \langle \alpha: \text{cls} \mid \alpha = A \lor \alpha = B \rangle. \text{obj}(\alpha)$
- $\exists \langle \alpha: \text{cls} \mid \alpha \ll C, \neg (\alpha = D) \rangle. \text{Obj}(\alpha)$

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

- Examples: equality comparison, list concatenation.

- \( \tau :: = \text{thisclass} \mid @\tau \quad \text{Exact type} \)

- \( @\alpha = \text{obj}(\alpha) \)
- \( @C(c) = \text{obj}(C(c)) \)

- Binary methods can only be invoked on object of exact type.
Binary Methods

C {
    \( m_1 = (\text{fun } (x: \text{thisclass}): t = e) \)
    \( m_2 = (\text{fun } (x: \text{@thisclass}): t = e) \)
}\n
The regularity property is preserved.

Vtable (c, C) =
{tag: tag(c ),
    m_1: (object(c), object(c))\rightarrow t)
    m_1: (object(c), obj(c))\rightarrow t)
}\n
object(c) = \exists (\alpha: \text{cls} | \alpha << c). \text{Obj}(\alpha)
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