Principal typings for Java-like languages

paper at POPL'04, Davide Ancona and Elena Zucca
DISI - University of Genova, Italy

+ ongoing work

MIKADO/MYTHS/DART meeting in S. Servolo, June, 13, 2004

Plan of the talk

Part I General framework for separate compilation and (sound and complete) inter-checking,
Relation with principal typings (Wells@ICALP02, previous talk)

Formalization of claim "Compositional analysis helps with separate compilation"

Part II Instantiation on Featherweight Java [IPW@OOPSLA99] (+ method overloading and field hiding)

Problem in compositional analysis of Java-like languages: code generation requires contextual information

Part I: Inter-checking

Basic notions adapted from Cardelli@POPL97:

separate compilation $\Gamma \vdash \mathtt{s}{:}\tau \leadsto \mathtt{b}$

- $\mathtt{s}$ source fragment = sequence of (class) declarations
  In this talk for simplicity one class declaration: $\mathtt{s} = \mathtt{class\ C}\{\ldots\}$

- τ type (in Java can be extracted from $\mathtt{s}$)

- $\mathtt{b}$ binary fragment

- Γ type environment = sequence of type assumptions $\gamma_1, \ldots, \gamma_n$
  on other classes needed for typechecking $\mathtt{s}$ generating $\mathtt{b}$

linkset $\mathtt{L} = \Gamma | \Gamma_i \vdash \mathtt{s}_i : \tau_i \leadsto \mathtt{b}_i{}^{i \in 1..n}$ valid judgments

inter-checking (informally) $\mathtt{L}$ inter-checks iff $\forall i \in 1..n$

assumptions Γ_i required by $\mathtt{s}_i$ are satisfied by other fragments

Formally (in Cardelli@POPL97 for assumptions of form $\mathtt{C} : \tau$)

$\forall i, j \in 1..n$

$\mathtt{s}_i = \mathtt{class}\ \mathtt{C}_i\{...\}$ and $\mathtt{C}_i : \tau$ in Γ_j implies $\tau = \tau_i$

Inter-checking (generalization)

Assumptions have arbitrary forms
e.g., $\mathtt{C}_1 \leq \mathtt{C}_2$, $\exists\,\mathtt{C}$, $\mathtt{C.m}(\overline{\mathtt{C}}) \stackrel{\mathtt{m\text{-}res}}{\longrightarrow} (\overline{\mathtt{C}}', \mathtt{C}')$, . . .

Assume entailment relation $\Gamma \vdash \Gamma'$

$\mathtt{L} = \Gamma \,|\, \overline{\Gamma_i \vdash \mathtt{s}_i : \tau_i \rightsquigarrow \mathtt{b}_i}^{i \in 1..n}$, $\mathtt{s}_i = \mathtt{\textbf{class}}\ \mathtt{C}_i\{...\}$

L inter-checks (written $\vdash \mathtt{L}\diamond$) iff for all $i \in 1..n$:

$$\Gamma, \overline{\mathtt{C}_j : \tau_j}^{j \in 1..n} \vdash \Gamma_i \text{ holds}$$

(Well-known) advantages (compositional analysis)

separate compilation + inter-checking versus global compilation:

- each fragment can be compiled in isolation
- a collection of fragments can be put together to form an executable application by only inspecting type information (type environment and type) of fragments without reinspecting code

BUT:
these advantages actually hold only if inter-checking satisfies some properties
(issue not considered in Cardelli@POPL97)

Soundness of inter-checking

For all $\mathtt{L} = \Gamma | \Gamma_i \vdash \mathtt{s}_i{:}\tau_i \leadsto \mathtt{b}_i{}^{i \in 1..n}$, $\mathtt{s}_i = \mathtt{class}\ \mathtt{C}_i\{\ldots\}$

$$\vdash \mathtt{L}\diamond \Rightarrow \Gamma, \mathtt{C}_j : \tau_j^{j \in 1..n} \vdash \mathtt{s}_i{:}\tau_i \leadsto \mathtt{b}_i{}^{i \in 1..n}$$

inter-checking successful $\Rightarrow$ compiling altogether $\mathtt{s}_1, \ldots, \mathtt{s}_n$ in Γ we successfully get the same binary fragments

Property always expected to hold

Sufficient condition: entailment sound $\Gamma_1 \vdash \Gamma_2 \Rightarrow \Gamma_1 \leq \Gamma_2$
$\Gamma_1 \leq \Gamma_2$ iff $\Gamma_2 \vdash \mathtt{s}{:}\tau \leadsto \mathtt{b} \Rightarrow \Gamma_1 \vdash \mathtt{s}{:}\tau \leadsto \mathtt{b}$

(Γ_1, Γ_2 consistent)

Completeness of inter-checking (intuition)

What can we conclude if inter-checking of $\mathtt{L} = \Gamma | \Gamma_i \vdash \mathtt{s}_i : \tau_i \leadsto \mathtt{b}_i{}^{i \in 1..n}$ fails?

This does not mean that the fragments cannot be safely linked!

For some fragment we could have chosen a too restrictive type environment (that is, containing unnecessary type assumptions)

Inter-checking is complete iff we can choose for each fragment Γ (and τ) s.t. this cannot happen

Definition of complete inter-checking

For all typable $(\mathtt{s},\mathtt{b})$ we can choose a "canonical" typing s.t.

for all $\mathtt{L} = \Gamma | {\Gamma_i \vdash \mathtt{s}_i{:}\tau_i \rightsquigarrow \mathtt{b}_i}^{i\in 1..n}$, with (Γ_i, τ_i) canonical typing for $(\mathtt{s}_i, \mathtt{b}_i)$

$$\overline{\Gamma, {\mathtt{C}_j : \tau_j}^{j\in 1..n} \vdash \mathtt{s}_i{:}\tau_i \rightsquigarrow \mathtt{b}_i}^{i\in 1..n} \Rightarrow \vdash \mathtt{L}\diamond$$

global compilation successful $\Rightarrow$ inter-checking successful

global compilation would either fail or produce different binaries $\Leftarrow$ inter-checking fails

Sufficient conditions for completeness

Theorem If
- the type system has principal typings
- $\Gamma_1 \leq \Gamma_2 \Rightarrow \Gamma_1 \vdash \Gamma_2$ (entailment complete)

then, inter-checking is complete w.r.t. global compilation.

NB: in Wells@ICALP02 (previous talk)

$(\Gamma_1, \tau_1) \leq (\Gamma_2, \tau_2)$ iff $\Gamma_1 \vdash \mathtt{s}{:}\tau_1 \leadsto \mathtt{b} \Rightarrow \Gamma_2 \vdash \mathtt{s}{:}\tau_2 \leadsto \mathtt{b}$

Here τ extracted from code, hence

$(\Gamma_1, \tau_1) \leq (\Gamma_2, \tau_2)$ iff $\Gamma_2 \leq \Gamma_1$ and $\tau_1 = \tau_2$

Part II: Instantiation on Featherweight Java

Problem: compositional analysis hard in Java, C-sharp: code generation requires contextual information

I will outline three approaches:
- standard
- compositional for (s, b) (instance of previous framework, AZ@POPL04)
- compositional for s (work in progress)

FJ Syntax – source and binary

$$
\begin{array}{rcl}
\texttt{s} & ::= & \texttt{CD}^s_1 \ldots \ \texttt{CD}^s_n \\
\texttt{CD}^s & ::= & \texttt{class C extends C}' \ \{ \ \texttt{FDS MDS}^s \ \} \\
\texttt{FDS} & ::= & \texttt{FD}_1 \ldots \ \texttt{FD}_n \\
\texttt{FD} & ::= & \texttt{C f;} \\
\texttt{MDS}^s & ::= & \texttt{MD}^s_1 \ldots \ \texttt{MD}^s_n \\
\texttt{MD}^s & ::= & \texttt{MH} \ \{\texttt{return E}^s\texttt{;}\} \\
\texttt{MH} & ::= & \texttt{C}_0 \ \texttt{m}(\texttt{C}_1 \ \texttt{x}_1, \ldots, \texttt{C}_n \ \texttt{x}_n) \\
\texttt{E}^s & ::= & \texttt{x} \mid \texttt{E}^s\texttt{.f} \mid \texttt{E}^s_0\texttt{.m}(\texttt{E}^s_1, \ldots, \texttt{E}^s_n) \\
 & & \mid \texttt{new C}(\texttt{E}^s_1, \ldots, \texttt{E}^s_n) \mid (\texttt{C})\texttt{E}^s \\
 & & \\
\texttt{b} & ::= & \texttt{CD}^b_1 \ldots \ \texttt{CD}^b_n \\
\texttt{CD}^b & ::= & \texttt{class C extends C}' \ \{ \ \texttt{FDS MDS}^b \ \} \\
\texttt{MDS}^b & ::= & \texttt{MD}^b_1 \ldots \ \texttt{MD}^b_n \\
\texttt{MD}^b & ::= & \texttt{MH} \ \{\texttt{return E}^b\texttt{;}\} \\
\texttt{E}^b & ::= & \texttt{x} \mid \texttt{E}^b \ll \texttt{C.f} \ \texttt{C}' \gg \\
 & & \texttt{E}^b_0 \ll \texttt{C.m}(\bar{\texttt{C}})\texttt{C}' \gg (\texttt{E}^b_1, \ldots, \texttt{E}^b_n) \\
 & & \mid \texttt{new} \ll \texttt{C} \ \bar{\texttt{C}} \gg (\texttt{E}^b_1, \ldots, \texttt{E}^b_n) \mid (\texttt{C})\texttt{E}^b \\
\bar{\texttt{C}} & ::= & \texttt{C}_1, \ldots, \texttt{C}_n
\end{array}
$$

An example

```
class C extends Parent {
 Type1 m (Type2 x) { return new Used().g(x);}
}
```

Approach 1 standard Java type systems use type environments extracted from current contexts, e.g.

```
class Parent { }
class Type1 {}
class Type2 extends Type 3 {}
class Used {
 Type1 g(Type 3)
}
```

```
class Parent { Type1 m (Type2)}
class Type1 extends Parent1{}
class Type2 extends Parent2 {}
class Parent2 extends Type 3 {}
class Used {
 Type1 g(Type 3)
}
```

(and infinitely many others)

generate `new Used()` $\ll$ Used.g(Type3)Type1 $\gg$ (x)

no principal typing (no minimal type environment)

```
class C extends Parent {
 Type1 m (Type2 x) { return new Used().g(x);}
}
```

Approach 2: Which is the minimal type information on other classes needed for typechecking class generating a given byte-code?

$\exists$ `Type1`, $\exists$ `Type2`
`Parent`☺`Type1 m(Type2)`
$\texttt{Used.g(Type2)} \stackrel{\texttt{m-res}}{\longrightarrow} (\texttt{Type3}, \texttt{Type1})$
$\texttt{Type2} \leq \texttt{Type3}$

Principal typing (minimal type environment) for pair $(\texttt{s}, \texttt{b})$
(for generating $\texttt{new Used()} \ll \texttt{Used.g(Type3)Type1} \gg \texttt{(x)}$)

Formally:

In AZ@POPL04:

- We define a type system T^{FJ} which is an instance of previous general framework

- Entailment in T^{FJ} is sound $\Rightarrow$ inter-checking is sound w.r.t. global compilation

- T^{FJ} has principal typings + entailment in T^{FJ} is complete $\Rightarrow$ inter-checking is complete w.r.t. global compilation

Work in progress

(preliminary DART paper ADDZ@FTfJP04 - ECOOP workshop)

```
class C extends Parent {
 Type1 m (Type2 x) { return new Used().g(x);}
}
```

Approach 3: Which is the minimal type information on other classes needed for typechecking class regardless of which bytecode is generated?

$\exists$ Type1, $\exists$ Type2
Parent☺Type1 m(Type2)
Used.g(Type2) $\xrightarrow{\texttt{m-res}}$ (α, β)
Type2 $\leq \alpha$
$\beta \leq$ Type1

generates new Used() $\ll$ Used.g$(\alpha)\beta \gg$ (x)
principal typing (minimal type environment) for s

With this approach:

- type inference is possible
- polymorphic types, polymorphic bytecode
- standard bytecode can be generated either at inter-checking time by solving type constraints (in ADDZ@FTfJP04), or at dynamic linking time (DART paper by Drossopoulou&Buckley, also presented at FTfJP04)

Summary

General framework for separate compilation and inter-checking, relation with principal typings
Result: we have exported notions to a different context
Here less restrictive type environment rather than more general type

Application to Java: stream of work

ALZ@PPDP02 first definition of alternative type system for Java-like languages
AZ@POPL04 this talk (proof of principality)
AL@FTfJP03, AL@JOT04 application to selective recompilation
Lagorio@ICTCS03,Lagorio@SAC04 first step toward application to full Java and development of smart compiler
ADDZ@FTfJP04 polymorphic bytecode (in progress)

Thank you!