Structural resolution for Abstract Compilation of Object-Oriented Languages

Luca Franceschini ¹ Davide Ancona ¹
Ekaterina Komendantskaya ²

¹Department of Informatics, Bioengineering, Robotics and Systems Engineering
University of Genoa

²Department of Computer Science
Heriot-Watt University of Edinburgh

Workshop on Coalgebra, Horn Clause Logic Programming and Types, 29th November 2016, Edinburgh, UK
Abstract compilation

Abstract compilation: encoding of source code in a logic program $P$ and then write type-checking/inference queries to be solved w.r.t. to $P$ (Ancona and Lagorio, 2009).
Abstract compilation

Abstract compilation: encoding of source code in a logic program $P$ and then write type-checking/inference queries to be solved w.r.t. to $P$ (Ancona and Lagorio, 2009).

- types are terms
Abstract compilation

Abstract compilation: encoding of source code in a logic program $P$ and then write type-checking/inference queries to be solved w.r.t. to $P$ (Ancona and Lagorio, 2009).

- types are terms
- inductive semantics is not enough
Abstract compilation

Abstract compilation: encoding of source code in a logic program $P$ and then write type-checking/inference queries to be solved w.r.t. to $P$ (Ancona and Lagorio, 2009).

- types are terms
- inductive semantics is not enough
- the system is parametric w.r.t. the resolution method
Contents

Corecursion in abstract compilation
  Co-LP and abstract compilation
  S-resolution & abstract compilation

Productivity of logic programs

Conclusions
Induction and coinduction

Finite derivations

▶ SLD resolution

Infinite derivations

▶ Co-LP (Simon et al., 2006): terms and derivations can be cyclic (a.k.a. regular or rational)

▶ S-resolution (Komendantskaya and Johann, 2015): coinduction is not limited to cyclic terms and proofs but logic programs must be productive
Induction and coinduction

Finite derivations

- SLD resolution

Infinite derivations

- Co-LP (Simon et al., 2006): terms and derivations can be cyclic (a.k.a. regular or rational)
- S-resolution (Komendantskaya and Johann, 2015): coinduction is not limited to cyclic terms and proofs but logic programs must be productive

Two different (sound but not complete) implementations for the greatest complete fixed-point semantics. Which one to choose?
Co-LP & abstract compilation

class Factorial extends Object {
    Factorial() {
        super();}

    compute(int n) {
        // no return type annotation
        if (n <= 0) 1
        else n * this.compute(n-1) }
}
class Factorial extends Object {
    Factorial() { super(); }

    compute(int n) { // no return type annotation
        if (n <= 0) 1
        else n × this.compute(n−1) }
    }

new Factorial().compute(4)
class Factorial extends Object {
    Factorial() {
        super();
    }

    compute(int n) {
        // no return type annotation
        if (n <= 0) 1
        else n * this.compute(n - 1)
    }
}

new Factorial().compute(4)

new(factorial, [], F) ∧ invoke(F, compute, [int], T)
Co-LP & abstract compilation

\[ \text{new}(\text{factorial}, [], F) \land \text{invoke}(F, \text{compute}, [\text{int}], T) \]

\[ \vdots \]

\[ \text{invoke}(F, \text{compute}, [\text{int}], T) \]

\[ \vdots \]

\[ \text{invoke}(F, \text{compute}, [\text{int}], \text{int}) \]

\[ \vdots \]

\[ \text{invoke}(F, \text{compute}, [\text{int}], \text{int}) \]
Co-LP & abstract compilation

\[ \text{new(}\text{factorial, [], } F) \land \text{invoke}(F, \text{compute}, [\text{int}], T) \]

\[ \vdots \]

\[ \text{invoke}(F, \text{compute}, [\text{int}], T) \]

\[ \vdots \]

\[ \text{invoke}(F, \text{compute}, [\text{int}], \text{int}) \]

\[ \vdots \]

\[ \text{invoke}(F, \text{compute}, [\text{int}], \text{int}) \]

Successful derivation in Co-LP!
S-resolution & abstract compilation

Not all infinite derivations are cyclic though...
S-resolution & abstract compilation

Not all infinite derivations are cyclic though...

class List extends Object {
    ...

    buildList(n, l) {
        if (n ≤ 0) l
        else this.buildList(n-1, new NEList(n, l))
    }
}
Not all infinite derivations are cyclic though...

```java
class List extends Object {
    ...

    buildList(n, l) {
        if (n <= 0) l
        else this.buildList(n-1, new NEList(n, l))
    }

    new List().buildList(n, new EList())
```
Not all infinite derivations are cyclic though…

class List extends Object {
  ...

  buildList(n, l) {
    if (n ≤ 0) l
    else this.buildList(n−1, new NEList(n, l))
  }

  new List().buildList(n, new EList())

  invoke(L, buildList, [int, E], R)
S-resolution & abstract compilation

```java
new List().buildList(n, new EList())
```

```
invoke(L, buildList, [int, E], R)
  :

invoke(L, buildList, [int, E'], R')
  :

invoke(L, buildList, [int, E''], R'')
  :
```
S-resolution & abstract compilation

\texttt{new List().buildList(n, new EList())}

\begin{align*}
\text{invoke}(L, buildList, [\textit{int}, E, R]) \\
\vdots \\
\text{invoke}(L, buildList, [\textit{int}, E', R']) \\
\vdots \\
\text{invoke}(L, buildList, [\textit{int}, E'', R'']) \\
\vdots
\end{align*}

Co-LP: infinite derivation...
S-resolution: success!

\[ R = E \lor \text{obj}(\text{nelist}, [\text{head} : \textit{int}, \text{tail} : R']) \]
Contents

Corecursion in abstract compilation
  Co-LP and abstract compilation
  S-resolution & abstract compilation

Productivity of logic programs

Conclusions
Productivity

S-resolution does not come for free: logic programs have to be *productive*.
Productivity

S-resolution does not come for free: logic programs have to be *productive*.

\[
p(f(X)) \leftarrow p(X) \\
p(X) \leftarrow p(X)
\]
Productivity

S-resolution does not come for free: logic programs have to be productive.

\[ p(f(X)) \leftarrow p(X) \]

\[ p(X) \leftarrow p(X) \]

S-resolution is strictly more powerful than Co-LP only for productive logic programs.
Productivity

Can we make arbitrary logic programs productive?
Productivity

Can we make arbitrary logic programs productive?

\[ p(X) \leftarrow p(X) \]

\[ p(X, f(Y)) \leftarrow p(X, Y) \]

More generally...

\[ p(\overline{t}^n) \leftarrow p_1(\overline{t_1}^{n_1}) \land \cdots \land p_m(\overline{t_m}^{n_m}) \]

\[ p(\overline{t}^n, f(X_1, \ldots, X_m)) \leftarrow p_1(\overline{t_1}^{n_1}, X_1) \land \cdots \land p_m(\overline{t_m}^{n_m}, X_m) \]
A simple transformation

A logic program $P$ is a set $\{C_1, \ldots, C_n\}$ of $n$ Horn clauses. The transformation $[\cdot]$ is inductively defined on the structure of logic programs (Fu and Komendantskaya, 2017).
A simple transformation

A logic program $P$ is a set $\{ C_1, \ldots, C_n \}$ of $n$ Horn clauses. The transformation $\llbracket \cdot \rrbracket$ is inductively defined on the structure of logic programs (Fu and Komendantskaya, 2017).

\[
\llbracket P \rrbracket = \llbracket \{ C_1, \ldots, C_n \} \rrbracket = \{ \llbracket C_1 \rrbracket_{\kappa_1}, \ldots, \llbracket C_n \rrbracket_{\kappa_n} \}
\]

\[
\llbracket C \rrbracket_{\kappa} = \llbracket A \leftarrow A_1 \land \cdots \land A_n \rrbracket_{\kappa} = \llbracket A \rrbracket_{\kappa}(x_1, \ldots, x_n) \leftarrow \llbracket A_1 \rrbracket_{x_1} \land \cdots \land \llbracket A_n \rrbracket_{x_n}
\]

\[
\llbracket A \rrbracket_\tau = \llbracket p(t_1, \ldots, t_n) \rrbracket_\tau = p(t_1, \ldots, t_n, \tau)
\]
A simple transformation

A logic program $P$ is a set $\{C_1, \ldots, C_n\}$ of $n$ Horn clauses. The transformation $[-]$ is inductively defined on the structure of logic programs (Fu and Komendantskaya, 2017).

$$[P] = [[\{C_1, \ldots, C_n\}] = \{[C_1]_{\kappa_1}, \ldots, [C_n]_{\kappa_n}\}

\lfloor C \rfloor_{\kappa} = \lfloor A \leftarrow A_1 \land \cdots \land A_n \rfloor_{\kappa} = \lfloor A \rfloor_{\kappa}(X_1, \ldots, X_n) \leftarrow \lfloor A_1 \rfloor_{X_1} \land \cdots \land \lfloor A_n \rfloor_{X_n}

\lfloor A \rfloor_{\tau} = \lfloor p(t_1, \ldots, t_n) \rfloor_{\tau} = p(t_1, \ldots, t_n, \tau)

\lfloor G \rfloor = \lfloor \leftarrow A_1 \land \cdots \land A_n \rfloor = \leftarrow \lfloor A_1 \rfloor_{X_1} \land \cdots \land \lfloor A_n \rfloor_{X_n}$$
Example of transformation

\[
\begin{cases}
\text{subclass}(A, A) \leftarrow \text{class}(A) \\
\text{subclass}(A, \text{object}) \leftarrow \text{class}(A) \\
\text{subclass}(A, C) \leftarrow \text{extends}(A, B) \land \text{subclass}(B, C)
\end{cases}
\]
Example of transformation

\[
\begin{bmatrix}
  \text{subclass}(A, A) & \leftarrow & \text{class}(A) \\
  \text{subclass}(A, \text{object}) & \leftarrow & \text{class}(A) \\
  \text{subclass}(A, C) & \leftarrow & \text{extends}(A, B) \land \text{subclass}(B, C)
\end{bmatrix}
\]

\[
\left[ \text{subclass}(A, A) \leftarrow \text{class}(A) \right]_{\kappa_1}
\]

\[
= \left[ \text{subclass}(A, \text{object}) \leftarrow \text{class}(A) \right]_{\kappa_2}
\]

\[
\left[ \text{subclass}(A, C) \leftarrow \text{extends}(A, B) \land \text{subclass}(B, C) \right]_{\kappa_3}
\]
Example of transformation

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>subclass(A, A) ← class(A)</code></td>
<td>Subclass of same class</td>
</tr>
<tr>
<td><code>subclass(A, object) ← class(A)</code></td>
<td>Subclass of object class</td>
</tr>
<tr>
<td><code>subclass(A, C) ← extends(A, B) ∧ subclass(B, C)</code></td>
<td>Subclass of extended class</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
  \text{[} & \text{subclass}(A, A) \leftarrow \text{class}(A) \text{]}_{\kappa_1} \\
  = & \text{[} \text{subclass}(A, object) \leftarrow \text{class}(A) \text{]}_{\kappa_2} \\
  \text{[} & \text{subclass}(A, C) \leftarrow \text{extends}(A, B) \land \text{subclass}(B, C) \text{]}_{\kappa_3}
\end{align*}
\]

\[
\begin{align*}
  \text{[} & \text{subclass}(A, A)]_{\kappa_1}(X) \leftarrow \text{[} \text{class}(A) \text{]}_X \\
  = & \text{[} \text{subclass}(A, object)]_{\kappa_2}(X) \leftarrow \text{[} \text{class}(A) \text{]}_X \\
  \text{[} & \text{subclass}(A, C)]_{\kappa_3}(X, Y) \leftarrow \text{[} \text{extends}(A, B) \text{]}_X \land \text{[} \text{subclass}(B, C) \text{]}_Y
\end{align*}
\]
Example of transformation

\[
\begin{align*}
\text{subclass}(A, A) & \leftarrow \text{class}(A) \\
\text{subclass}(A, \text{object}) & \leftarrow \text{class}(A) \\
\text{subclass}(A, C) & \leftarrow \text{extends}(A, B) \land \text{subclass}(B, C)
\end{align*}
\]
Properties of $\llbracket \neg \rrbracket$

- Logic programs can be made productive by construction (productivity checking is hard!)
Properties of $\llbracket \cdot \rrbracket$

- Logic programs can be made productive by construction (productivity checking is hard!)
- Simple, easy to implement, inductive and compositional
Properties of $[-]$:

- Logic programs can be made productive by construction (productivity checking is hard!)
- Simple, easy to implement, inductive and compositional
- *Sound and complete both inductively and coinductively!*

Given a logic program $P$ and an atom $A$, for some term $\tau$:

$$A \in \text{M}_P \iff J_A K \tau \in \text{M}_{J P K}$$

$\text{M}_P$ and $\text{M}_{co P}$ are the inductive and coinductive model of $P$, respectively.
Properties of $\mathbb{J}$−

- Logic programs can be made productive by construction (productivity checking is hard!)
- Simple, easy to implement, inductive and compositional
- **Sound and complete both inductively and coinductively!**
  Given a logic program $P$ and an atom $A$, for some term $\tau$:

  $A \in M_P \iff \llbracket A \rrbracket_\tau \in M_{[P]}$
  
  $A \in M^c_P \iff \llbracket A \rrbracket_\tau \in M^{co}_{[P]}$

  ($M_P$ and $M^c_P$ are the inductive and coinductive model of $P$, respectively)
Contents

Corecursion in abstract compilation
   Co-LP and abstract compilation
   S-resolution & abstract compilation

Productivity of logic programs

Conclusions
Conclusions

The system we propose is divided into three phases:

1. abstract compilation (object-oriented source $\Rightarrow$ logic program)
2. transformation ensuring productivity
3. S-resolution for type-checking/inference queries

Pros:
- very precise static type analysis for object-oriented languages
- support for parametric and data polymorphism
- modularity: the system is parametric w.r.t. the resolution method
- inference and type annotation work together seamlessly

Cons:
- sometimes inferred types may not have a finite regular representation
- (probably) undecidable: union and record types together are very expressive
Conclusions

The system we propose is divided into three phases:

1. abstract compilation (object-oriented source \(\mapsto\) logic program)
2. transformation ensuring productivity
3. S-resolution for type-checking/inference queries

Pros:

- very precise static type analysis for object-oriented languages
- support for parametric and data polymorphism
- modularity: the system is parametric w.r.t. the resolution method
- inference and type annotation work together seamlessly
Conclusions

The system we propose is divided into three phases:
1. abstract compilation (object-oriented source \(\leadsto\) logic program)
2. transformation ensuring productivity
3. S-resolution for type-checking/inference queries

Pros:
- very precise static type analysis for object-oriented languages
- support for parametric and data polymorphism
- modularity: the system is parametric w.r.t. the resolution method
- inference and type annotation work together seamlessly

Cons:
- sometimes inferred types may not have a finite regular representation
- (probably) undecidable: union and record types together are very expressive
Related work

D. Ancona and G. Lagorio.  
*Coinductive type systems for object-oriented languages.*  
ECOOP, 2009.

L. Simon, A. Mallya, A. Bansal and G. Gupta.  
*Coinductive logic programming.*  

E. Komendantskaya and P. Johann.  
Submitted to ACM Transactions in Computational Logic, 2015.

P. Fu and E. Komendantskaya.  
*Operational semantics of resolution and productivity in Horn clause logic.*  