Static single information form for abstract compilation

Davide Ancona Giovanni Lagorio

University of Genova

IFIP Theoretical Computer Science 2012, September 26-28, Amsterdam

Main motivations

- type analysis of dynamic object-oriented languages
- integration of type analysis (abstract compilation) and standard analysis techniques used in compilers
- reuse of already available frameworks (e.g. LLVM)
- abstract compilation [AnconaLagorio@ECOOP09]: program to be analyzed abstractly translated into a logic program

Language, types and subtyping

In this paper: approach applied to a specific example

- A simple dynamic o-o language, no type annotations, even though syntax is reminiscent of Java
- Types: class names, union types (hence, set of class names)
- Subtyping: set inclusion

  $\texttt{Square} \leq \texttt{SquareVColoredSquare}$

  $\texttt{ColoredSquare} \leq \texttt{SquareVColoredSquare}$

Language, types and subtyping

In this paper: approach applied to a specific example

- A simple dynamic o-o language, no type annotations, even though syntax is reminiscent of Java
- Types: class names, union types (hence, set of class names)
- Subtyping: set inclusion

  $\texttt{Square} \leq \texttt{SquareVColoredSquare}$

  $\texttt{ColoredSquare} \leq \texttt{SquareVColoredSquare}$

But the approach is applicable in general

- More expressive types: structural types, exceptions, ...
- More expressive subtyping relation

Inheritance is not subtyping

Example

```
class Square {
    ...
    equals(s){return this.side == s.side;}
}
class ColoredSquare extends Square {
    ...
    equals(cs){return this.side == cs.side &&
                      this.color == cs.color;}
}
```

Inheritance is not subtyping

Example

```
class Square {
    ...
    equals(s){return this.side == s.side;}
}
class ColoredSquare extends Square {
    ...
    equals(cs){return this.side == cs.side &&
                      this.color == cs.color;}
}
```

- `ColoredSquare` $\not\leq$ `Square` even if `ColoredSquare` extends `Square`

Inheritance is not subtyping

Example

```
class Square {
    ...
    equals(s){return this.side == s.side;}
}
class ColoredSquare extends Square {
    ...
    equals(cs){return this.side == cs.side &&
                      this.color == cs.color;}
}
```

- `ColoredSquare` $\not\leq$ `Square` even if `ColoredSquare` extends `Square`
- `s.equals(new Square(3))` is well typed if `s` has type `Square`, but not if `s` has type `ColoredSquare`

Static Single Assignment form and type analysis

Example

```
class ShapeReader {
   next() {...}
   readCircle() { ... return new Circle(...); }
   readSquare() { ... return new Square(...); }
   read() {
      st = this.next();
      if(st.equals("circle")) {
         sh = this.readCircle();
         this.print("A circle with radius ");
         this.print(sh.getRadius());
      }
      else if(st.equals("square")) {
         sh = this.readSquare();
         this.print("A square with side ");
         this.print(sh.getSide());
      }
      else throw new IOException();
      this.println("Area = "+sh.area());
   }}
```

Static Single Assignment form and type analysis

Example

```
class ShapeReader {
   next() {...}
   readCircle() { ... return new Circle(...); }
   readSquare() { ... return new Square(...); }
   read() {
      st = this.next();
      if(st.equals("circle")) {
         sh = this.readCircle();          // T_sh ≥ Circle
         this.print("A circle with radius ");
         this.print(sh.getRadius());      // T_sh ≤ Circle
      }
      else if(st.equals("square")) {
         sh = this.readSquare();          // T_sh ≥ Square
         this.print("A square with side ");
         this.print(sh.getSide());        // T_sh ≤ Square
      }
      else throw new IOException();
      this.println("Area = "+sh.area()); // T_sh ≤ Circle∨Square
   }}
```

SSA and type analysis

Problem

There is no single type T_{sh} that can be assigned to variable `sh`

Possible solutions

- type systems allowing assignments of different types to different occurrences of the same variable
- transform code in Single Static Assignment form: occurrences of the same variable but with different definition points are renamed

SSA form [CytronEtAl91,Singer05]

Each variable is determined by exactly one assignment statement

- a suitable renaming of variables is performed to keep track of the possibly different versions of the same variable (*virtual register*)
- pseudo-functions, conventionally called φ-functions, have to be inserted to correctly deal with merge points

Code in SSA form

```
read() {
    st = this.next();
    if(st.equals("circle")) {
        sh0 = this.readCircle();
        this.print("A circle with radius ");
        this.print(sh0.getRadius());
    }
    else if(st.equals("square")) {
        sh1 = this.readSquare();
        this.print("A square with side ");
        this.print(sh1.getSide());
    }
    else throw new IOException();
    sh2=φ(sh0,sh1);
    this.println("Area = "+sh2.area());
}
```

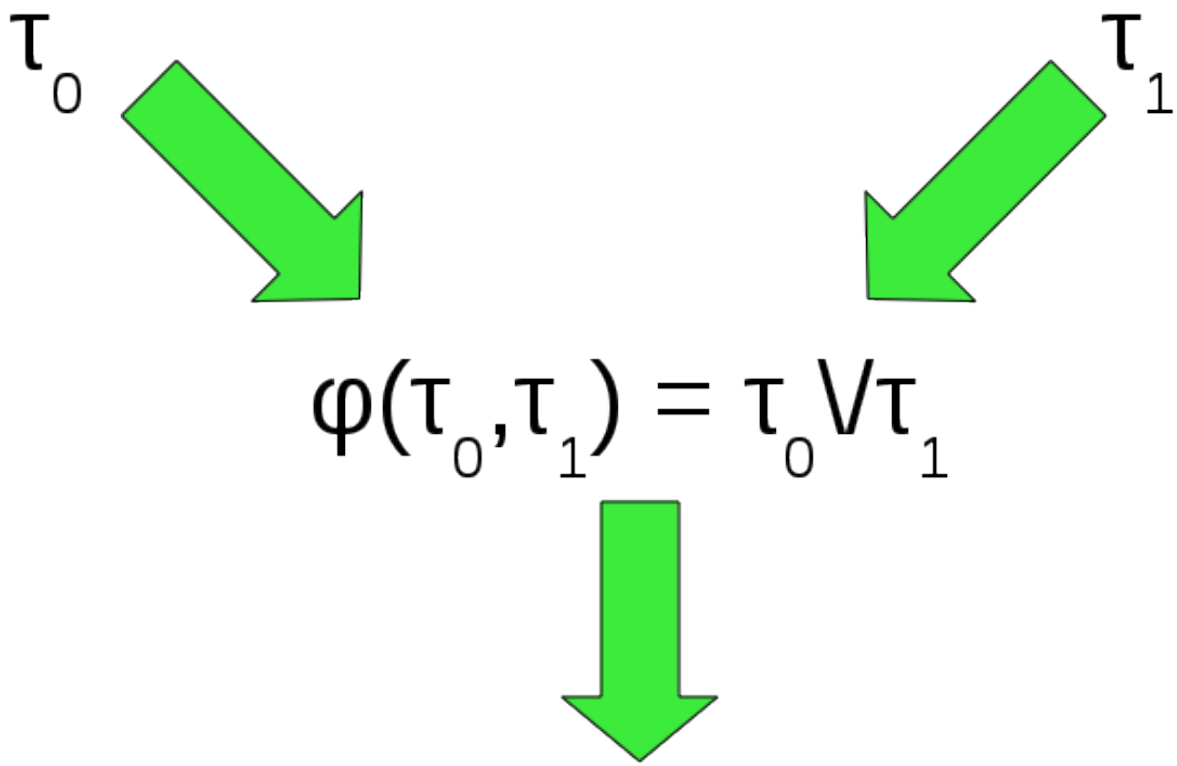
Code in SSA form

```
read() {
    st = this.next();
    if(st.equals("circle")) {
        sh0 = this.readCircle();          // T_sh0 ≥ Circle
        this.print("A circle with radius ");
        this.print(sh0.getRadius());      // T_sh0 ≤ Circle
    }
    else if(st.equals("square")) {
        sh1 = this.readSquare();          // T_sh1 ≥ Square
        this.print("A square with side ");
        this.print(sh1.getSide());        // T_sh1 ≤ Square
    }
    else throw new IOException();
    sh2=φ(sh0,sh1);                       // T_sh2 ≥ T_sh0 ∨ T_sh1
    this.println("Area = "+sh2.area()); // T_sh2 ≤ Circle∨Square
}
```

Valid type assignment:

sh_0:Circle, sh_1:Square, sh_2:Circle$\vee$Square

φ-function abstracted by the union type constructor



A more challenging example

Example 2

```
class Square {
...
   largerThan(sh) {
   if(sh instanceof Square)
      return this.side > sh.side;
   else
      return this.area() > sh.area();
   }
```

A more challenging example

Method `largerThan` in SSA form

```
class Square {
...
   largerThan(sh0) {
   if(sh0 instanceof Square)
      return this.side > sh0.side;
   else
      return this.area() > sh0.area();
   }
```

A more challenging example

Method `largerThan` in SSA form

```
class Square {
...
   largerThan(sh0) {
   if (sh0 instanceof Square)
      return this.side > sh0.side;          // T_sh0 ≤ Square
   else
      return this.area() > sh0.area();      // T_sh0 ≤ Square∨Circle
  }
```

Valid type assignment

$sh_0 : \texttt{Square}$

This prevents method `largerThan` from being used in most cases

Static Single Information form [Ananian99,Singer05]

Solution

Two different virtual registers need to be used in the branches

Method `largerThan` in SSI form

```
class Square {
...
   largerThan(sh0) {
   if(sh0 instanceof Square) with (sh1,sh2) = σ(sh0)
      return this.side > sh1.side;
   else
      return this.area() > sh2.area();
   }
```

Static Single Information form [Ananian99,Singer05]

Solution

Two different virtual registers need to be used in the branches

Method `largerThan` in SSI form

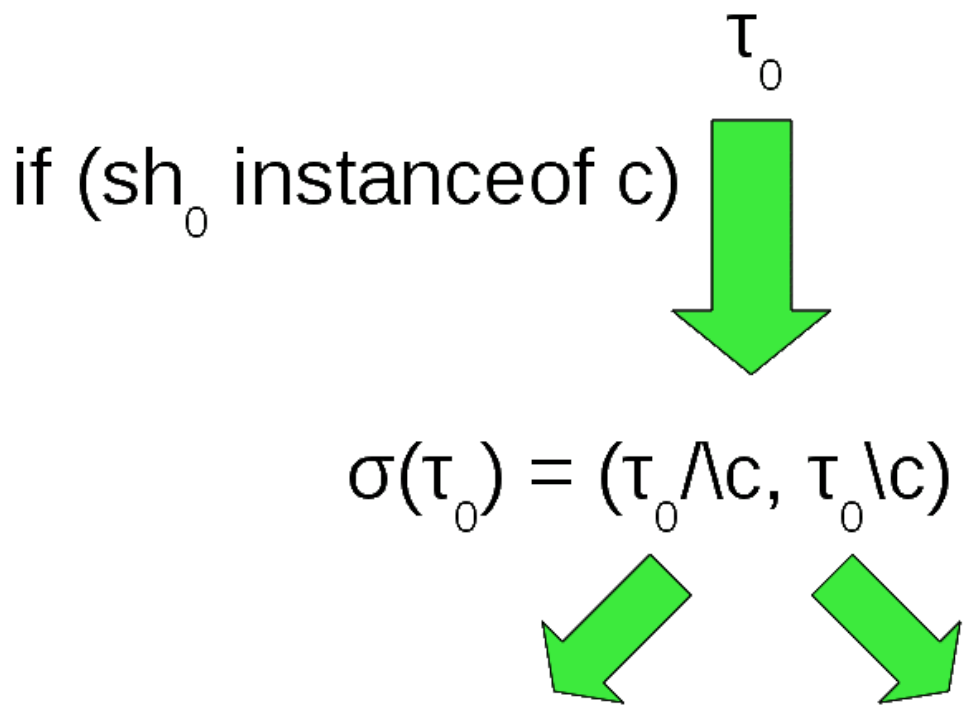
```
class Square {
...
   largerThan(sh0) {
   if(sh0 instanceof Square) with (sh1,sh2) = σ(sh0)
                                        // Tsh1 ≥ Tsh0∧Square
                                        // Tsh2 ≥ Tsh0\Square
      return this.side > sh1.side;       // Tsh1 ≤ Square
   else
      return this.area() > sh2.area();   // Tsh2 ≤ Square∨Circle
   }
```

Valid type assignment

sh_0:SquareVCircle, sh_1:Square, sh_2:Circle

σ-function abstraction

c subclass of c' = reflexive and transitive closure of **extends**
$\tau_0 \wedge c = \{c' \in \tau_0 \mid c' \text{ subclass of } c\}$
$\tau_0 \setminus c = \{c' \in \tau_0 \mid c' \text{ not subclass of } c\}$



Definition of the language in SSI form

$$
\begin{array}{rcl}
prog & ::= & \overline{cd}^{n}\ e \\
cd & ::= & \textbf{class}\ c_1\ \textbf{extends}\ c_2\ \{\ \overline{f}^{n}\ \overline{md}^{k}\ \}\quad (c_1 \neq Object, Bool, c_2 \neq Bool) \\
md & ::= & m(\overline{x}^{n})\ \{\overline{b}^{n}\} \\
b & ::= & l{:}e \\
r & ::= & x_i \\
e & ::= & r \mid \textbf{false} \mid \textbf{true} \mid \textbf{new}\ c(\overline{e}^{n}) \mid e.f \mid e_0.m(\overline{e}^{n}) \mid e_1; e_2 \mid r = e \\
 & & \mid e_1.f = e_2 \mid \textbf{jump}\ l \mid r = \varphi(\overline{r}^{n}) \mid \textbf{return}\ r \\
 & & \mid \textbf{if}\ (r\ \textbf{instanceof}\ c)\ \textbf{with}\ \overline{(r', r'') = \sigma(r''')}^{n}\ \textbf{jump}\ l_1\ \textbf{else jump}\ l_2
\end{array}
$$

Example of method in SSI form

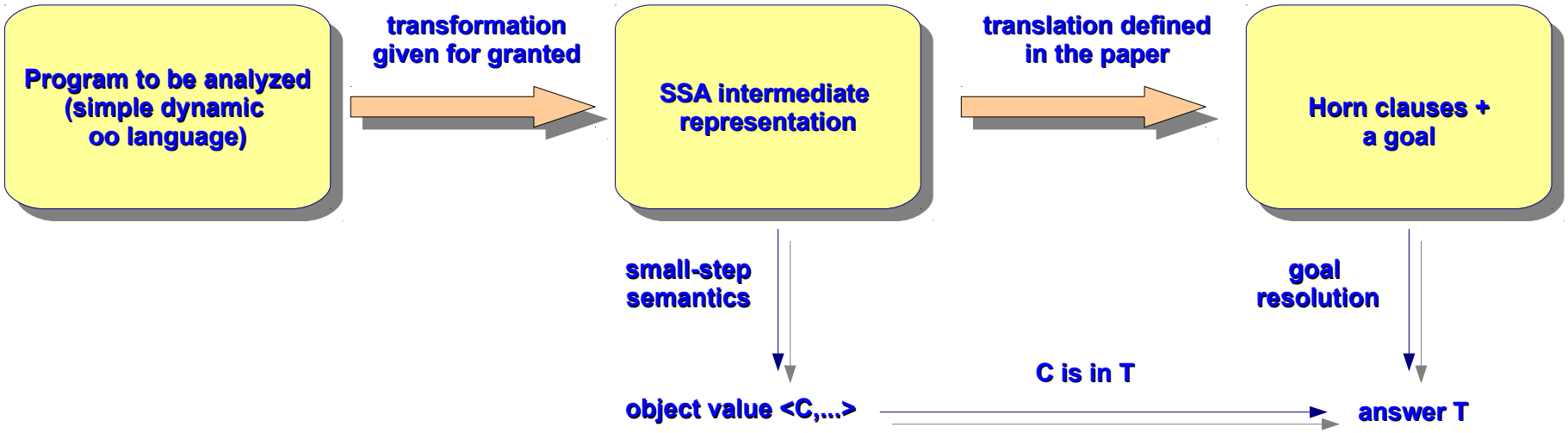
```
largerThan(sh0) {
  b1:{if(sh0 instanceof Square) with (sh1,sh2) = σ(sh0)
                                     (this1,this2) = σ(this0)
        jump b2;
      else
        jump b3;}
  b2:{r0=this1.side > sh1.side;
      jump out;}
  b3:{r1=this2.area() > sh2.area();
      jump out;}
  out:{r2=φ(r0,r1);
      return r2;}
}
```

Small-step semantics of SSI (outlined)

Main ingredients:

- runtime expressions
  - frame expression: $\langle fr, \mu \rangle \{e\}$ (execution of a method invocation)
  - execution context $\langle fr, \mu \rangle$: stack frame fr and fully qualified method name μ
  - stack frame fr: finite map from virtual registers to values and time-stamps
- $r{=}\varphi(r', r'')$: r updated with the most-recently updated register
- heap $\mathcal{H}$: finite map from object references o to object values
- object value: class name + finite map from fields to values
- basic judgment: $\mathcal{H}, \langle fr, \mu \rangle \vdash e \rightarrow \mathcal{H}', \langle fr', \mu' \rangle, e'$

More in the paper



Conclusion

Main contributions

- reuse compiler technology (SSA,SSI) for type analysis
- formal small step semantics of SSI
- type analysis by means of abstract compilation of a dynamic object-oriented language
- abstract compilation is sound

Future work

- null reference analysis: **if** `(x == null)` . . .

Thank you!