Static single information form for abstract compilation

Davide Ancona and Giovanni Lagorio

DISI, University of Genova, Italy
{davide,lagorio}@disi.unige.it

Abstract. In previous work we have shown that more precise type analysis can be achieved by exploiting union types and static single assignment (SSA) intermediate representation (IR) of code.
In this paper we exploit static single information (SSI), an extension of SSA proposed in literature and adopted by some compilers, to allow assignments of more precise types to variables in conditional branches. In particular, SSI can be exploited rather easily and effectively to assign more precise static types in presence of explicit runtime typechecking, a necessity that occurs rather often in statically typed object-oriented languages, and even more than often in dynamically typed ones.
We show how the use of SSI form can be smoothly integrated with abstract compilation, our approach to static type analysis. In particular, we define abstract compilation based on union and nominal types for a simple dynamically typed Java-like language in SSI form with a runtime typechecking operator, to show how precise the proposed analysis can be.

1 Introduction

In previous work [6] we have shown that more precise type analysis can be achieved by exploiting union types and static single assignment (SSA) [7] intermediate representation (IR) of code. Most modern compilers (among others, GNU's GCC [13], the SUIF compiler system [12], and JIT compilers including Java HotSpot [11], and Java Jikes RVM [9]) implement efficient algorithms for translating code in SSA IR [8], therefore focusing on analysis of SSA IR not only allows more precise results, but also favors reuse of compiler technology, and better integration with existing compilers. In particular, we have studied how *abstract compilation* [5, 4, 6] can naturally take advantage of SSA IR when union types are considered. Abstract compilation aims to reconcile static type analysis and symbolic execution: one can check whether the execution of a certain expression e is type safe, when variable contents range over possibly infinite sets of values (represented by types), by solving a goal, obtained by abstract compilation of e, w.r.t. the coinductive[1] semantics of the constraint logic program automatically generated from the source program in which the expression is executed.

[1] Coinduction allows proper treatment of recursive types and methods [5].

Abstract compilation fosters *plug-and-play* static type analysis since one can provide several compilation schemes for the same language, each corresponding to a different kind of analysis, without changing the inference engine that implements the coinductive semantics of constraint logic programs [15, 14, 4]. For instance, in previous work we have defined compilation schemes for Java-like languages based on union and structural object types, to support parametric and data polymorphism, (that is, polymorphic methods and fields) that allow quite precise type analysis, and a smooth integration with the nominal type annotations contained in the programs [5]; other proposed compilation schemes aim to detect uncaught exceptions for Java-like languages [4], or to integrate SSA IR in presence of imperative features [6].

In this paper we exploit static single information (SSI), an extension of SSA proposed in literature [2, 17], to allow assignments of more precise types to variables in conditional branches. SSI has been already adopted by compilers as LLVM [18], PyPy [3], and SUIF [16], and proved to be more effective than SSA for performing data flow analysis, program slicing, and interprocedural analysis.

In particular, we show how SSI can be exploited rather easily and effectively by abstract compilation to assign more precise static types in presence of explicit runtime typechecking, a necessity that occurs rather often in statically typed object-oriented languages [19], and even more than often in dynamically typed ones.

To this aim, we formally define the operational semantics of a simple dynamically typed Java-like language in SSI form equipped with a runtime typechecking operator, and then provide an abstract compilation scheme based on union and nominal types supporting more precise type analysis of branches guarded by explicit runtime typechecks.

The paper is structured as follows: Section 2 introduces SSA and SSI IRs and motivates their usefulness for type analysis; Section 3 formally defines the SSI IR of a dynamically typed Java-like language equipped with an operator **instanceof** for runtime typechecking. Section 4 presents a compilation scheme for the defined IR, based on nominal and union types, and Section 5 concludes with some considerations on future work. Abstract compilation of the code examples in Section 2 together with the results of the resolution of some goals can be found in an extended version of this paper.[2]

2 Type analysis with SSA and SSI

In this section SSA and SSI IRs are introduced and their usefulness for type analysis is motivated.

Type analysis with static single assignment form

Method `read()` declared below, in a hypothetical dynamically typed Java-like language, creates and returns a shape which is read through method `nextLine()`

[2] Available at ftp://ftp.disi.unige.it/person/AnconaD/foveoos11long.pdf

that reads the next available string from some input source. The partially omitted methods `readCircle()` and `readSquare()` read the needed data from the input, create, and return a new corresponding instance of `Circle` or `Square`.

```
class ShapeReader {
   ...
   nextLine() {...}
   readCircle() { ... return new Circle(...); }
   readSquare() { ... return new Square(...); }
   read() {
      st = this.nextLine();
      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.print("Area = ");
      this.print(sh.area());
   }
}
```

Although method `read()` is type safe, no type can be inferred for `st` to correctly typecheck the method; indeed, when method `area()` is invoked, variable `st` may hold an instance of `Circle` or `Square`, therefore the most precise type that can be correctly assigned to `st` is `Circle` $\vee$ `Square`. However, if `st` has type `Circle` $\vee$ `Square`, then both `sh.getRadius()` and `sh.getSide()` do not typecheck.

There are two different kinds of approaches to solve the problem shown above. Either one defines a rather sophisticated flow-sensitive type system able to associate different types with different occurrences of the same variable, or one can typecheck the SSA IR in which the method is compiled.

In an SSA IR the value of each variable is determined by exactly one assignment statement [7]. To obtain this property, a flow graph is built, and a suitable renaming of variables is performed to keep track of the possibly different versions of the same variable; following Singer's terminology [17] we call these versions *virtual registers*. Conventionally, this is achieved by using a different subscript for each virtual register corresponding to the same variable. For instance, in the SSA IR of method `read()` (Figure 1) there are three virtual registers ($\texttt{sh}_0$, $\texttt{sh}_1$ and $\texttt{sh}_2$) for the variable `sh`.

To transform a program into SSA form, pseudo-functions, conventionally called φ-functions, have to be inserted to correctly deal with merge points. For instance, in block 5 the value of `sh` can be that of either $\texttt{sh}_0$ or $\texttt{sh}_1$, therefore a new virtual register $\texttt{sh}_2$ has to be introduced to preserve the SSA property. The

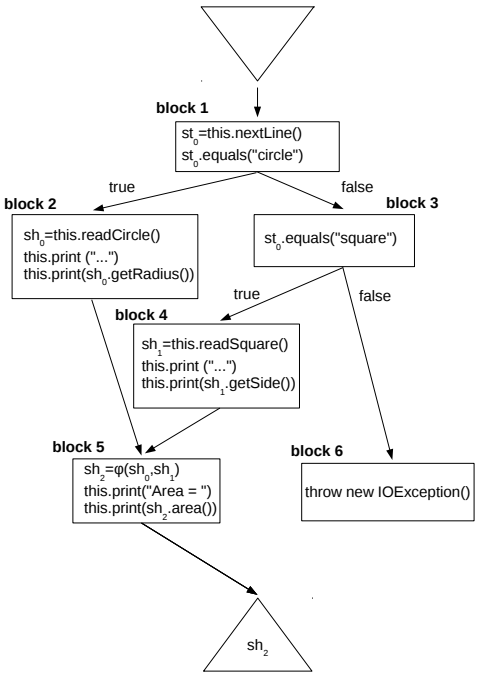

Fig. 1. Control flow graph corresponding to the body of method `read`

expression φ(`sh`$_0$,`sh`$_1$) keeps track of the fact that the value of `sh`$_2$ is determined either by `sh`$_0$ or `sh`$_1$.

At the level of types φ-functions naturally correspond to the union type constructor (Figure 2): arrows correspond to data flow and, as usual, to ensure soundness the type at the origin of an arrow must be a subtype of the type the arrow points to. In the figure, $\tau_0, \tau_1 \leq \tau_0 \vee \tau_1 \leq \tau_2$.

The transformation of a source program into its SSA IR is standard [7], and there exists a quite efficient algorithm to perform it [8], therefore it is more convenient to define abstract compilation for programs in SSA IR. While flow graphs are used for generating SSA IR, here we adopt a textual language more suitable for defining an abstract compilation scheme. For instance, the SSA IR of method `read()` is the following:

```
read() {
  b1:{st0 = this.nextLine();
      if(st0.equals("circle"))
         jump b2;
      else
         jump b3;}
  b2:{sh0=this.readCircle();
```

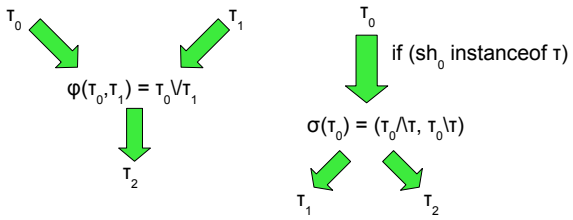

Fig. 2. Type theoretic interpretation of φ-functions and σ-functions

```
        this.print("A circle with radius ");
        this.print(sh0.getRadius());
        jump b5;}
    b3:{if(st0.equals("square"))
            jump b4;
        else
            jump b6;}
    b4:{sh1=this.readSquare();
        this.print("A square with side ");
        this.print(sh1.getSide());
        jump b5;}
    b5:{sh2=φ(sh0,sh1);
        this.print("Area = ");
        this.print(sh2.area());
        jump out;}
    b6:{throw new IOException();}
    out:{return sh2;}
}
```

The body of a method in IR is a sequence of uniquely labeled blocks; each block ends with either a conditional or unconditional `jump`, a **return** or a **throw**[3]. For simplicity, we require that only the last block[4] contains the **return** statement.

Type analysis with static single information form

Let us consider method `largerThan(sh)` of class `Square`, where **instanceof** is exploited to make the method more efficient in case the parameter `sh` contains an instance of (a subclass) of `Square`.

```
class Square {
...
   largerThan(sh) {
     if(sh instanceof Square)
```

[3] In the formal treatment that follows we omit exceptions for simplicity.

[4] This can be always obtained by introducing new virtual registers and inserting a φ-function in case of multiple returned values.

```
        return this.side > sh.side;
     else
        return this.area() > sh.area();
   }
}
```

The method is transformed in the following SSA IR:

```
  largerThan(sh0) {
    b1:{if(sh0 instanceof Square)
           jump b2;
        else
           jump b3;}
    b2:{r0=this.side > sh0.side;
        jump out;}
    b3:{r1=this.area() > sh0.area();
        jump out;}
    out:{r2=φ(r0,r1);
        return r2;}
}
```

Since variable `sh` is not updated, both blocks `b2` and `b3` refer to the same virtual register $\texttt{sh}_0$. As a consequence, the only possible type that can be correctly associated with $\texttt{sh}_0$ is `Square`, thus making the method of little use. However, this problem can be encompassed if one considers the SSI IR of the method [2, 17].

```
  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;}
}
```

SSI is an extension of SSA enforcing the additional constraint that all variables must have different virtual registers in the branches of conditional expressions; such a property is obtained by a suitable renaming and by the insertion of σ-functions at split points. As a consequence, suitable virtual registers and σ-functions have to be introduced also for the read-only pseudo-variable **this**.

The notion of σ-function is the dual of φ-function (Figure 2); the type theoretic interpretation of σ depends on the specific kind of conditional context. If such a context is of the form $(\texttt{sh}_0$ **instanceof** `Square`$)$ as in the example, then σ splits the type τ_0 of $\texttt{sh}_0$ in the type $\tau_0 \wedge$`Square`, assigned to $\texttt{sh}_1$, and in the

type $\tau_0\backslash$`Square`, assigned to $\texttt{sh}_2$, where the intersection and the complement operators have to be properly defined (see Section 4). For instance, if $\texttt{sh}_0$ has type `Square`$\vee$`Circle`, then $\texttt{sh}_1$ has type `(Square`$\vee$`Circle)`$\wedge$`Square`=`Square`, and $\texttt{sh}_2$ has type `(Square`$\vee$`Circle)`$\backslash$`Square`=`Circle`, therefore `Square`$\vee$`Circle` turns out to be a valid type for the parameter $\texttt{sh}_0$ of the method `largerThan`. For what concerns **this**, in this particular example no real split is performed: if we assume that $\textbf{this}_0$ has type `Square`, then `Square` is split in `(Square,Square)`, and both $\textbf{this}_1$ and $\textbf{this}_2$ have type `Square`.

3 Language definition

In this section we formally define an SSI IR for a simple dynamically typed Java-like language equipped with an **instanceof** operator for performing runtime typechecking.

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

Syntactic assumptions: inheritance is not cyclic, method bodies are in correct SSI form and are terminated with a unique `return` statement, method and class names are disjoint, no name conflicts in class, field, method and parameter declarations, `new` $c(\overline{e}^{\,n})$ allowed only if $c \neq \mathit{Bool}$, and declared parameters cannot be `this`.

Fig. 3. SSI intermediate language

A program is a collection of class declarations followed by a main expression e with no free variables. The notation $\overline{cd}^{\,n}$ is a shortcut for $cd_1, \ldots, cd_n$. A class declares its direct superclass (only single inheritance is supported), its fields, and its methods. Two predefined classes are available: *Object*, the usual root class of the inheritance tree, and *Bool*, the class of the two literals `false` and `true`; such a class cannot be extended, and does not provide any constructor.

Every class, except *Bool*, comes equipped with the implicit constructor with parameters corresponding to all fields, in the same order as they are inherited and declared, but for simplicity no user declared constructors can be added.

Method bodies are sequences of uniquely labeled blocks that contain sequences of expressions. We assume that all blocks contain exactly one jump, necessarily placed at the end of the block. Three different kinds of jumps are considered: local unconditional and conditional jumps, and returns from methods. Method bodies are implicitly assumed to be in correct SSI IR: each virtual

register is determined by exactly one assignment statement, and all variables must have different virtual registers in the branches of conditional expressions. Finally, all method bodies contain exactly one return expression, which is always placed at the end of the body.

The receiver object can be referred inside method bodies with the special implicit parameter `this`; besides usual statements and expressions, we consider φ and σ pseudo-function assignments.

In $r = \varphi(\overline{r}^n)$, $n \geq 2$ and all virtual registers $\overline{r}^n$ occurring in φ are assumed to refer to the same variable, denoted by $var(r_1) = \ldots = var(r_n)$.

All σ-functions are used in association with conditional jumps; each virtual register r occurring in either branches has to be split into two new distinct versions used in the blocks labeled by $\mathtt{l}_1$, and $\mathtt{l}_2$, respectively. The conditions we consider are only of the form (r `instanceof` c) since our aim is studying how type analysis can be enhanced by SSI in the presence of explicit runtime typechecks; however, more elaborated forms of conditions can be expressed in terms of the more primitive form (r `instanceof` c) by simple transformations performed by the compiler front-end. For instance, the statement

if (x.m1() **instanceof** A && y.m2() **instanceof** B) e_1 **else** e_2

can be transformed in the equivalent SSI IR

```
b0 : {z0=xk.m1();
      if (z0 instanceof A) with ... jump b1 else jump b3;}
b1 : {w0=yn.m2();
      if (w0 instanceof B) with ... jump b2 else jump b3;}
b2 : {e'1}
b3 : {e'2}
```

where $\mathtt{z}_0$ and $\mathtt{w}_0$ are fresh, e'_1 and e'_2 are the SSI IRs of e_1 and e_2, respectively, and σ-functions assignments (that depend on e_1 and e_2) have been omitted. Depending on the types and abstract compilation scheme under consideration, there could be other kinds of conditions for which SSI would improve type analysis; for instance, if an abstract compilation scheme allows analysis of null references, such an analysis could be enhanced by SSI in the case of conditional expressions with conditions of the form (`x == null`). On the other hand, for conditions of the form (x_1 `<` x_2) SSI does not help refine type analysis as long as the abstract compilation scheme maps numeric values to the standard primitive types **int**, **float**, and **double**; in this case the type theoretic interpretation of σ-functions is the loosest one: $\sigma(\tau)=(\tau,\tau)$ (hence, no split is actually performed).

Semantics: To define the small step semantics of the language we first need to specify values v (see Figure 4), which are either the literals `false` and `true` (recall that they are predefined instances of the *Bool* class) or identities o of dynamically created objects. Furthermore, we add *frame expressions* $ec\{e\}$, where ec is an execution context; frame expressions are *runtime expressions*, that is, expressions that represent intermediate evaluation steps and that are needed for defining the small step semantics of method calls. An execution context ec is a pair

consisting of a stack frame fr and a code address a. A frame expression $\langle fr, a\rangle\{e\}$ corresponds to the execution of a call to a method m declared in class c, where e is the residual expression (yet to be evaluated) of the currently executed block, fr is the stack frame of the method call, $a = \mu.l$ is the address of the current block, where $\mu = c.m$ is the fully qualified name of the method, and l is the label of the current block.

Stack frames fr map variables and virtual registers to their corresponding values. These frames are represented by a pair of lists of associations, $x \mapsto v$ and $r \mapsto v$, where variables and virtual registers are all distinct. Keeping track of the values of both virtual registers and their associated variable allows for a simple semantics, as discussed below.

Heaps $\mathcal{H}$ map object identifiers o to objects, that is, pairs consisting of a class name c and the set of field names f with their corresponding value v.

Figure 5 shows the execution rules. The main evaluation judgment has shape $\mathcal{H} \vdash e \to \mathcal{H}', e'$, meaning that e rewrites to e' in $\mathcal{H}$, yielding the new heap $\mathcal{H}'$. Furthermore, rule (ctx-closure) uses the auxiliary judgment $\mathcal{H}, ec \vdash e \to \mathcal{H}', ec', e'$, meaning that redex e rewrites to e' in $\mathcal{H}$ and ec, yielding the new execution context ec' and heap $\mathcal{H}'$. Both judgments, and all auxiliary functions should be parametrized by the whole executing program, $\overline{cd}^n$, which is kept implicit. The execution of the main expression of a program starts from an empty heap $\epsilon_{\mathcal{H}}$ and an empty stack frame ϵ, annotated by the pair $\langle \bot, \bot \rangle$, since the main expression is not actually contained in any block/method.

The main evaluation judgment is defined by the three rules (meth-call) (a new frame is pushed on the stack), (ctx-closure) (evaluation continues in the currently active frame), and (return) (the current active frame is popped from the stack).

In rule (meth-call), the object referenced by o is retrieved in order to find its class, c. Then, the auxiliary functions *firstBlock* and *params* return the first block of the method and its parameter names, respectively. The result of the evaluation is a frame expression, where the new stack frame maps parameters to their corresponding passed arguments, and `this` to the reference o, and the code address is the fully qualified name of the invoked method, $c.m$, plus the label of its first block, l. Finally, the resulting expression is the context applied to the body of the first block.

Rule (ctx-closure) performs a single computation step in the currently active frame (corresponding to the most nested frame expression). The execution context is extracted by *currentEC*;[5] then, if the redex e rewrites to e' yielding $\mathcal{H}'$ and ec' (see the other rules defining the auxiliary evaluation judgment), then the expression $\mathcal{C}[e]$ rewrites to $\mathcal{C}'[e']$, yielding the new heap $\mathcal{H}'$; context $\mathcal{C}'[\ ]$ is obtained from $\mathcal{C}[\ ]$ by updating the frame expression corresponding to the currently active frame with the new execution context ec'.

In rule (return) the currently active frame is removed, and the resulting expression is the context applied to the value associated with the returned virtual register r in the frame.

[5] The straightforward definitions of *currentEC* and *updateEC* have been omitted.

Rule (var) models the access to a virtual register (`this` is considered a special read-only local variable), by simply extracting the corresponding value from the stack frame fr; this works because of the way the assignment is handled in Rule (var-asn).

Variable and field assignments evaluate to their right values; rule (var-asn) models virtual register assignments, and has the side effect of updating, in the current stack frame fr, the values of both the virtual register r and its associated variable x. This implements a cache write-through strategy, where virtual registers cache values that are to be stored in the memory location corresponding to the variable to which virtual registers refer to; in this way evaluation of φ-function is simpler (see comments below).

Rule (fld-asn) models field assignments; in that case, the object referenced by o is retrieved from the heap, and its value updated.

Rule (seq) models the fact that a value is discarded when followed by another.

Rule (phi) models the assignment of a phi-function, which accesses to a (set $\overline{r}^n$ of different versions of the same) local variable x, by assigning the value contained in x (this is correct because of the write-through semantics of the Rule (var-asn)) to the virtual register r'.

Rule (new) models object creation; a new object, identified by a fresh reference o, is added to the heap $\mathcal{H}$. The fields $\overline{f}^n$ of the newly created object are initialized by the value passed to the constructor.

Rule (fld-acc) models field accesses; its evaluation is quite trivial: the object is retrieved from the heap, and the resulting expression is the value of the selected field.

Rules (jump) and (if) model unconditional and conditional jumps, respectively. These are the only rules that modify the label-part of the execution context. The evaluation of a jump, which, by construction, is known to be the last expression of a sequence, corresponds to replacing the jump expression itself with the expression e contained in the block labeled l' and updating the stack frame annotation accordingly.

The conditional jump (rule (if)) has to both choose which branch to execute and which virtual registers have to be updated, depending on whether the value of the register r (contained in $fr(r)$) is a reference to an object of a subclass of c. If the referenced object is indeed an instance of c, then the target label is l_1 and the virtual registers $\overline{r'}^n$ are updated; otherwise, the target label is l_2 and the virtual registers $\overline{r''}^n$ are updated.

4 Abstract compilation

In this section we define an abstract compilation scheme for programs in the SSI IR presented in Section 3. Programs are translated into a Horn formula Hf (that is, a logic program) and a goal B; type analysis amounts to resolve B w.r.t. the coinductive semantics (that is the greatest Herbrand model) of Hf [5].

In previous work we have defined compilation schemes based on union and structural object types, to support parametric and data polymorphism, (that

$$
\begin{array}{rl}
v ::= & \texttt{false} \mid \texttt{true} \mid o \quad \text{(values)} \\
e ::= & ec\{e\} \mid \ldots \quad \text{(runtime expressions; that is, frame expressions plus source expressions as defined in Figure 3)} \\
ec ::= & \langle fr, a \rangle \quad \text{(execution context)} \\
fr ::= & \overline{x \mapsto v}^{j} \overline{r \mapsto v}^{k} \quad \text{(stack frames)} \\
a ::= & \mu.l \quad \text{(block addresses, where } \mu \text{ are full method names } c.m\text{)} \\
\mathcal{H} ::= & \overline{o \mapsto \langle c, \overline{f \mapsto v}^{j} \rangle}^{k} \quad \text{(heaps)} \\
\mathcal{C}[\cdot] ::= & [\cdot] \mid ec\{\mathcal{C}[\cdot]\} \mid \texttt{new } c(\overline{v}^{n}, \mathcal{C}[\cdot], \overline{e}^{j}) \mid \mathcal{C}[\cdot].f \mid \mathcal{C}[\cdot].m(\overline{e}^{k}) \mid v_0.m(\overline{v}^{j}, \mathcal{C}[\cdot], \overline{e}^{k}) \\
& \mid \mathcal{C}[\cdot]; e \mid v; \mathcal{C}[\cdot] \mid x = \mathcal{C}[\cdot] \mid \mathcal{C}[\cdot].f = e \mid v.f = \mathcal{C}[\cdot] \\
& \mid \texttt{if } (\mathcal{C}[\cdot]) \texttt{ with } \overline{(x', x'') = \sigma(x''')}^{n} \texttt{ jump } l_1 \texttt{ else jump } l_2
\end{array}
$$

Fig. 4. Syntactic definitions instrumental to the operational semantics

$$
\text{(meth-call)}\frac{\begin{array}{l}\mathcal{H}(o) = \langle c, \_ \rangle \\ \mathit{firstBlock}(c.m) = l : e \\ \mathit{params}(c.m) = \overline{r}^{n} \\ fr = \overline{r \mapsto v}^{n}, \texttt{this}_0 \mapsto o\end{array}}{\begin{array}{l}\mathcal{H} \vdash \mathcal{C}[o.m(\overline{v}^{n})] \\ \quad \rightarrow \mathcal{H}, \mathcal{C}[\langle fr, c.m.l \rangle \{e\}]\end{array}} \qquad \text{(ctx-closure)}\frac{\begin{array}{c}\mathit{currentEC}(\mathcal{C}[\cdot]) = ec \\ \mathcal{H}, ec \vdash e \rightarrow \mathcal{H}', ec', e' \\ \mathcal{C}'[\cdot] = \mathit{updateEC}(\mathcal{C}[\cdot], ec')\end{array}}{\mathcal{H} \vdash \mathcal{C}[e] \rightarrow \mathcal{H}', \mathcal{C}'[e']}
$$

$$
\text{(return)}\frac{}{\begin{array}{l}\mathcal{H} \vdash \mathcal{C}[\langle fr, a \rangle \{\texttt{return } r\}] \\ \quad \rightarrow \mathcal{H}, \mathcal{C}[fr(r)]\end{array}} \qquad \text{(fld-acc)}\frac{\mathcal{H}(o) = \langle c, \overline{f \mapsto v}^{n} \rangle \quad f = f_j}{\mathcal{H}, ec \vdash o.f \rightarrow \mathcal{H}, ec, v_j}
$$

$$
\text{(var)}\frac{}{\mathcal{H}, \langle fr, a \rangle \vdash r \rightarrow \mathcal{H}, \langle fr, a \rangle, fr(r)} \qquad \text{(new)}\frac{\begin{array}{c}o \text{ fresh in } \mathcal{H} \\ \mathit{fieldNames}(c) = \overline{f}^{n}\end{array}}{\begin{array}{l}\mathcal{H}, ec \vdash \texttt{new } c(\overline{v}^{n}) \\ \quad \rightarrow \mathcal{H}[\langle c, \overline{f \mapsto v}^{n} \rangle / o], ec, o\end{array}}
$$

$$
\text{(seq)}\frac{}{\mathcal{H}, ec \vdash v_1; v_2 \rightarrow \mathcal{H}, ec, v_2} \qquad \text{(var-asn)}\frac{x = \mathit{var}(r) \quad x \neq \texttt{this}}{\begin{array}{l}\mathcal{H}, \langle fr, a \rangle \vdash r = v \\ \quad \rightarrow \mathcal{H}, \langle fr[v/r, v/x], a \rangle, v\end{array}}
$$

$$
\text{(fld-asn)}\frac{\begin{array}{l}\mathcal{H}(o) = \langle c, \overline{f \mapsto v}^{n} \rangle \\ f = f_j \quad \text{if } i = j \text{ then } v_i' = v \\ \qquad\qquad \text{else } v_i' = v_i\end{array}}{\begin{array}{l}\mathcal{H}, ec \vdash o.f = v \\ \quad \rightarrow \mathcal{H}[\langle c, \overline{f \mapsto v'}^{n} \rangle / o], ec, v\end{array}} \qquad \text{(jump)}\frac{\mathit{block}(\mu.l') = l' : e}{\begin{array}{l}\mathcal{H}, \langle fr, \mu.l \rangle \vdash \texttt{jump } l' \\ \quad \rightarrow \mathcal{H}, \langle fr, \mu.l' \rangle, e\end{array}}
$$

$$
\text{(phi)}\frac{v = fr(\mathit{var}(r_1))}{\mathcal{H}, \langle fr, a \rangle \vdash r' = \varphi(\overline{r}^{n}) \rightarrow \mathcal{H}, \langle fr[v/r'], a \rangle, v}
$$

$$
\text{(if)}\frac{\begin{array}{l}\mathcal{H}(fr(r)) = \langle c', \_ \rangle \\ \text{if } c' \leq c \text{ then } l' = l_1, fr' = fr[\overline{fr(r''')/r'}^{n}] \\ \qquad\qquad \text{else } l' = l_2, fr' = fr[\overline{fr(r''')/r''}^{n}] \\ \mathit{block}(\mu.l') = l' : e\end{array}}{\begin{array}{l}\mathcal{H}, \langle fr, \mu.l \rangle \vdash \texttt{if } (r \texttt{ instanceof } c) \texttt{ with } \overline{(r', r'') = \sigma(r''')}^{n} \\ \quad \texttt{jump } l_1 \texttt{ else jump } l_2 \\ \quad \rightarrow \mathcal{H}, \langle fr', \mu.l' \rangle, e\end{array}}
$$

Fig. 5. Small-step semantics

is, polymorphic methods and fields) that allow quite precise type analysis, but pose problems in terms of termination of the resolved goals. In this paper we present a simpler compilation scheme based on union and nominal object types, that allows a less precise type analysis on the one hand (but still more precise than that obtained with the standard type systems of mainstream languages as Java and C#) but, on the other hand, avoids non termination problems with goal resolution, since the space of all possible valid types for a given program is always bounded. Therefore, while coinductive constraint logic programming [4] (where constraints are expressed in terms of the subtyping relation) is needed for structured types to ensure termination of the resolution of some goals, here would only allow more precise and efficient analysis; however, here subtyping is treated as an ordinary predicate to make the presentation simpler.

Summarizing, here we use nominal rather than structured types for the following reasons: SSI allows more precise type analysis even in the presence of less expressive types, the presentation is simpler, and, last but not least, we take advantage of the *plug-and-play* facility offered by the abstract compilation approach by providing yet another compilation scheme; in practice, more advanced compilations schemes could be adopted, including structural [5, 6] and exception types [4] (see further comments in the conclusion).

The compilation of programs, class, and method declarations is defined in Figure 6. We follow the usual syntactic conventions for logic programs: logical variable names begin with upper case, whereas predicate and functor names begin with lower case letters. Underscore denotes anonymous logical variables that occur only once in a clause; [ ] and $[e|l]$ respectively represent the empty list, and the list where e is the first element, and l is the rest of the list.

$$\text{(prog)}\frac{\forall\, i = 1..n \;\; cd_i \rightsquigarrow Hf_i \qquad e \rightsquigarrow (t \mid B)}{\overline{cd}^n \;\; e \rightsquigarrow (Hf^d \cup \overline{Hf}^n \mid B)}$$

$$\text{(class)}\frac{\forall\, i = 1..k \;\; md_i \;\; \textbf{in} \;\; c_1 \rightsquigarrow Hf_i \quad inhFields(c_1) = \overline{f'}^h}{\begin{array}{l}\texttt{class}\;\; c_1 \;\; \texttt{extends}\;\; c_2 \;\; \{\;\; \overline{f}^n \;\;\; \overline{md}^k \;\; \} \rightsquigarrow \overline{Hf}^k \cup \\ \left\{\begin{array}{l} class(c_1) \leftarrow true. \\ extends(c_1, c_2) \leftarrow true. \\ \overline{dec\_field(c_1, f)}^n \leftarrow true. \\ new(CE, c_1, [\overline{T'}^h, \overline{T}^n]) \leftarrow new(CE, c_2, [\overline{T'}^h]), \overline{field\_upd(CE, c_1, f, T)}^n. \end{array}\right\}\end{array}}$$

$$\text{(meth)}\frac{\overline{b}^n \rightsquigarrow (t \mid B)}{\begin{array}{l} m(\overline{r}^n)\{\overline{b}^n\} \;\; \textbf{in} \;\; c \rightsquigarrow \\ \left\{\begin{array}{l} dec\_meth(c, m) \leftarrow true. \\ has\_meth(CE, c, m, [This_0, \overline{r}^n], t) \leftarrow subclass(This_0, c), B. \end{array}\right\}\end{array}}$$

$$\text{(body)}\frac{\forall\, i = 1..n \;\; b_i \rightsquigarrow B_i}{\overline{b}^n \;\; l{:}\texttt{return}\;\; r \rightsquigarrow (r \mid \overline{B}^n)}$$

Fig. 6. Compilation of programs, class, and method declarations and bodies.

Each rule defines a different compilation judgment. The judgment $\overline{cd}^n \ e \rightsquigarrow (\mathit{Hf}^d \cup \overline{\mathit{Hf}}^n | B)$ means that the program $\overline{cd}^n \ e$ is compiled into the pair $(\mathit{Hf}^d \cup \overline{\mathit{Hf}}^n | B)$, where $\mathit{Hf}^d \cup \overline{\mathit{Hf}}^n$ is a Horn formula (that is, a set of Horn clauses), and B is a goal[6]. Static analysis of the program corresponds to resolving the goal B w.r.t. the coinductive semantics of $\mathit{Hf}^d \cup \overline{\mathit{Hf}}^n$. The Horn formula Hf^d contains all generated clauses that are invariant w.r.t. the program (see Figure 8), whereas each Hf_i is obtained by compiling the class declaration cd_i (see below); the goal B is obtained from the compilation of the main expression e (see below); the term t corresponding to the returned type of e is simply ignored here, but it is necessary for compiling expressions (see comments on Figure 7).

The compilation of a class declaration `class` c_1 `extends` c_2 $\{ \ \overline{f}^n \ \ \overline{md}^k \ \}$ is a set of clauses, including each clause Hf_i obtained by compiling the method md_i (see below), clauses asserting that class c_1 declares field f_i, for all $i = 1..n$, and three specific clauses for predicates *class*, *extends*, and *new*. The clause for *new* deserves some explanations: the atom $\mathit{new}(ce, c, [\overline{t}^n])$ succeeds iff the invocation of the implicit constructor of c with n arguments of type $\overline{t}^n$ is type safe in the global class type environment ce. The class environment ce is required for compiling field access and update expressions (see Figure 7): it is a finite map (simply represented by a list) associating class names with field records, which are finite maps (again simply represented by lists) associating all fields of a class with their corresponding types. Class environments are required because of nominal types: abstract compilation with structural types allows data polymorphism on a per-object basis, whereas here we obtain only a very limited form of data polymorphism on a per-class basis. Type safety of object creation is checked by ensuring that object creation for the direct superclass c_2 is correct, where only the first h arguments corresponding to the inherited fields (returned by the auxiliary function *inhFields* whose straightforward definition has been omitted) are passed; then, predicate *field_upd* defined in Figure 8 checks that all remaining n arguments, corresponding to the new fields declared in c_1, have types that are compatible with those specified in the class environment. The clause dealing with the base case for the root class *Object* is defined in Figure 8.

The judgment $m(\overline{r}^n)\{\overline{b}^n\}$ **in** $c \rightsquigarrow \mathit{Hf}$ means that the method declaration $m(\overline{r}^n)\{\overline{b}^n\}$ contained in class c compiles to Horn clauses Hf. Just two clauses are generated per method declaration: the first simply states that method m is declared in class c (and is needed to deal with inherited methods, see Figure 8), whereas the second is obtained by compiling the body of the method. The atom $\mathit{has\_meth}(ce, c, m, [t_0, \overline{t}^n], t)$ succeeds iff in class environment ce method m of class c can be safely invoked on target object of type t_0, with n arguments of type $\overline{t}^n$ and returned value of type t. The predicate *subclass* (defined in Figure 8) ensures that the method can be invoked only on objects that are instances of c or of one of its subclasses. For simplicity we assume that all names (including

[6] For simplicity we use the same meta-variable B to denote conjunctions of atoms (that is, clause bodies), and goals, even though more formally goals are special clauses of the form $\mathit{false} \leftarrow B$.

`this`) are translated to themselves, even though, in practice appropriate injective renaming should be applied [5].

The compilation of a method body $\overline{b}^n$ l:`return` r consists of the type of the returned virtual register r, and the conjunction of all the atoms generated by the compilation of blocks $\overline{b}^n$.

Figure 7 defines abstract compilation for blocks, and expressions.

$$\text{(block)}\frac{e \rightsquigarrow (t \mid B)}{l{:}e \rightsquigarrow B} \qquad \text{(seq)}\frac{e_1 \rightsquigarrow (t_1 \mid B_1) \qquad e_2 \rightsquigarrow (t_2 \mid B_2)}{e_1; e_2 \rightsquigarrow (t_2 \mid B_1, B_2)}$$

$$\text{(c-jmp)}\frac{\begin{array}{c}\text{if } var(r_i''') = var(r)\\ \text{then } t_i' = T, t_i'' = F\\ \text{else } t_i' = r_i''', t_i'' = r_i''' \quad T, F \text{ fresh}\end{array}}{\begin{array}{l}\texttt{if } (r \texttt{ instanceof } c) \texttt{ with } \overline{(r', r'') = \sigma(r''')}^n \texttt{ jump } l_1 \texttt{ else jump } l_2 \texttt{ in } \rightsquigarrow\\ \quad (\mathit{void} \mid \mathit{inter}(r, c, T), \mathit{diff}(r, c, F), \overline{\mathit{var\_upd}(r', t'), \mathit{var\_upd}(r'', t'')}^n)\end{array}}$$

$$\text{(var-upd)}\frac{e \rightsquigarrow (t \mid B)}{r = e \rightsquigarrow (t \mid B, \mathit{var\_upd}(r, t))} \qquad \text{(jmp)}\frac{}{\texttt{jump } l \rightsquigarrow (\mathit{void} \mid \mathit{true})}$$

$$\text{(field-upd)}\frac{e_1 \rightsquigarrow (t_1 \mid B_1) \qquad e_2 \rightsquigarrow (t_2 \mid B_2)}{e_1.f = e_2 \rightsquigarrow (t_2 \mid B_1, B_2, \mathit{field\_upd}(CE, t_1, f, t_2))}$$

$$\text{(phi)}\frac{}{r = \varphi(\overline{r}^n) \rightsquigarrow (\overline{\vee r}^n \mid \mathit{var\_upd}(r, \overline{\vee r}^n))}$$

$$\text{(new)}\frac{\forall\, i = 1..n \;\; e_i \rightsquigarrow (t_i \mid B_i)}{\texttt{new } c(\overline{e}^n) \rightsquigarrow (c \mid \overline{B}^n, \mathit{new}(CE, c, [\overline{t}^n]))}$$

$$\text{(field-acc)}\frac{e \rightsquigarrow (t \mid B) \quad R \text{ fresh}}{e.f \rightsquigarrow (R \mid B, \mathit{field}(CE, t, f, R))}$$

$$\text{(invk)}\frac{\forall\, i = 0..n \;\; e_i \rightsquigarrow (t_i \mid B_i) \quad R \text{ fresh}}{e_0.m(\overline{e}^n) \rightsquigarrow (R \mid B_0, \overline{B}^n, \mathit{invoke}(CE, t_0, m, [\overline{t}^n], R))}$$

$$\text{(var)}\frac{}{r \rightsquigarrow (r \mid \mathit{true})} \qquad \text{(bool)}\frac{v \in \{\texttt{true}, \texttt{false}\}}{v \rightsquigarrow (\mathit{bool} \mid \mathit{true})}$$

Fig. 7. Compilation of blocks and expressions

Compiling a block $l{:}e$ returns the conjunction of atoms obtained by compiling e; the type t of e is discarded.

The compilation of $e_1; e_2$ returns the type of e_2 and the conjunction of atoms generated from the compilation of e_1 and e_2.

The compilation of an unconditional jump generates the type *void* and the empty conjunction of atoms *true*. A conditional jump has type *void* as well, but a non empty sequence of predicates is generated to deal with the splitting performed by σ-functions; predicates *inter* and *diff* (defined in Figure 9) correspond to intersection T and difference F between the type of r and c, respectively, and

predicate *var_upd* (defined in Figure 8) ensures that the type of virtual registers r_i' and r_i'' are compatible with the pairs of types returned by the σ-functions. In case r_i''' refers to the same variable of r the types of such a pair are the computed intersection T and difference F, respectively, otherwise the pair (r_i''', r_i''') is returned (hence, no split is actually performed).

Compilation of assignments to virtual registers and fields yields the conjunction of the atoms generated from the corresponding sub-expressions, together with the atoms that ensure that the assignment is type compatible (with predicates *var_upd* and *field_upd* defined in Figure 8). The returned type is the type of the right-hand side expression.

Compilation of φ-function assignments to virtual registers is just an instantiation of rule (var-upd) where the type of the expression is the union of the types of the virtual registers passed as arguments to φ.

Compilation rules for object creation, field selection, and method invocation follow the same pattern: the type of the expression is a fresh logical variable (except for object creation) corresponding to the type returned by the specific predicate (*new*, *field*, and *invoke* defined in Figure 8). The generated atoms are those obtained from the compilation of the sub-expressions, together with the atom specific of the expression.

Rules (var) and (bool) are straightforward.

Figure 8 and Figure 9 define the set Hf^d used in compilation rule (prog) corresponding to all generated clauses that are invariant w.r.t. the compiled program.

Clauses in Figure 8 deserve some comments for what concerns the subtyping relation (predicate *subtype*); as expected, classes c_1 and c_2 are both subtypes of $c_1 \vee c_2$, but no subclass of c_1 or c_2 is a subtype of $c_1 \vee c_2$, because subclassing is not subtyping, since no rules are imposed on method overriding. Consider for instance the following source code snippet:

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

According to our compilation scheme, the expression `s1.equals(s2)` has type `Bool` if `s1` and `s2` have type `Square` and `Square`$\vee$`ColoredSquare`, respectively, but the same expression is not well-typed if `s1` has type `ColoredSquare` (hence, `ColoredSquare`$\not\leq$`Square`), since `s2` may contain an instance of `Square` for which field `color` is not defined. Subtyping is required for defining the predicates *var_upd* and *field_upd* for virtual register and field updates: the type of the source must be a subtype of the type of the destination.

$class(object) \leftarrow true.$
$class(bool) \leftarrow true.$
$extends(bool, object) \leftarrow true.$
$subclass(X, X) \leftarrow class(X).$
$subclass(X, Y) \leftarrow extends(X, Z), subclass(Z, Y).$
$subtype(T, T) \leftarrow true.$
$subtype(T1 \vee T2, T) \leftarrow subtype(T1, T), subtype(T2, T).$
$subtype(T, T1 \vee T2) \leftarrow subclass(T, T1).$
$subtype(T, T1 \vee T2) \leftarrow subclass(T, T2).$
$field(CE, C, F, T) \leftarrow has\_field(C, F), class\_fields(CE, C, R), field\_type(R, F, T).$
$field(CE, T1 \vee T2, F, FT1 \vee FT2) \leftarrow field(CE, T1, F, FT1),$
$\qquad field(CE, T2, F, FT2).$
$class\_fields([C : R|CE], C, R) \leftarrow no\_def(C, CE).$
$class\_fields([C1 : \_|CE], C2, R) \leftarrow class\_fields(CE, C2, R), C1 \neq C2.$
$field\_type([F{:}T|R], F, T) \leftarrow no\_def(F, R).$
$field\_type([F1 : \_|R], F2, T) \leftarrow field\_type(R, F2, T), F1 \neq F2.$
$no\_def(\_, [\,]) \leftarrow true.$
$no\_def(K1, [K2 : \_|Tl]) \leftarrow no\_def(K1, Tl), K1 \neq K2.$
$invoke(CE, C, M, A, RT) \leftarrow has\_meth(CE, C, M, [C|A], RT).$
$invoke(CE, T1 \vee T2, M, A, RT1 \vee RT2) \leftarrow invoke(CE, T1, M, A, RT1),$
$\qquad invoke(CE, T2, M, A, RT2).$
$new(\_, object, [\,]) \leftarrow true.$
$has\_field(C, F) \leftarrow dec\_field(C, F).$
$has\_field(C, F) \leftarrow extends(C, P), has\_field(P, F), \neg dec\_field(C, F).$
$has\_meth(CE, C, M, A, R) \leftarrow extends(C, P), has\_meth(CE, P, M, A, R),$
$\qquad \neg dec\_meth(C, M).$
$var\_upd(T1, T2) \leftarrow subtype(T2, T1).$
$field\_upd(CE, C, F, T2) \leftarrow field(CE, C, F, T1), subtype(T2, T1).$

Fig. 8. Clauses defining the predicates used by the abstract compilation

Predicate *field* looks up the type of a field in the global class environment, and is defined in terms of the auxiliary predicates *has_field*, *class_fields*, *field_type*, and *no_def*. In particular, predicate *has_field* checks that a class has actually a certain field, either declared or inherited. The definitions of *class_fields*, *field_type*, and *no_def* are straightforward (*no_def* ensures that a map does not contain multiple entries for a key), whereas the clause for *has_field* dealing with inherited fields is similar to the corresponding one for *invoke* (see below).

If the target object has a class type c, then the correctness of method invocation is checked with predicate *has_meth* applied to class c and to the same list of arguments where, however, the type c of **this** is added at the beginning. If the target object has a union type, predicate *invoke* checks that method invocation is correct for both types of the union, and then merges the types of the results into a single union type.

Finally, the clause for *has_meth* deals with the inherited methods: if class c does not declare method m, then *has_meth* must hold on the direct superclass of c.

$inter(C1, C2, C1) \leftarrow subclass(C1, C2).$
$inter(T1 \vee T2, C, IT1 \vee IT2) \leftarrow inter(T1, C, IT1), inter(T2, C, IT2).$
$inter(T1 \vee T2, C, IT1) \leftarrow inter(T1, C, IT1), \neg inter(T2, C, \_).$
$inter(T1 \vee T2, C, IT2) \leftarrow inter(T2, C, IT2), \neg inter(T1, C, \_).$
$diff(C1, C2, C1) \leftarrow class(C1), \neg subclass(C1, C2).$
$diff(T1 \vee T2, C, IT1 \vee IT2) \leftarrow diff(T1, C, IT1), diff(T2, C, IT2).$
$diff(T1 \vee T2, C, IT1) \leftarrow diff(T1, C, IT1), \neg diff(T2, C, \_).$
$diff(T1 \vee T2, C, IT2) \leftarrow diff(T2, C, IT2), \neg diff(T1, C, \_).$

Fig. 9. Clauses defining type intersection and difference

Predicates *inter* and *diff* define type splitting for σ-functions; the asymmetric definition for *inter* is due to the fact that subclass is not subtyping: if r has type c_1, then it means that it contains an object that is an instance of c_1, therefore the condition `(`r **instanceof** c_2`)` is false when c_2 is a proper subclass of c_1. Both predicates fail if the returned type is empty, therefore a conditional jump is not considered correct if either branches are dead (that is, not reachable). In practice, it would be better to avoid this kind of failures by introducing an explicit empty type constant to be able to detect dead code without any failure. Such an alternative option does not pose any technical issue, but since it is more verbose (a new clause dealing with the empty type must be added for most predicates) has not been considered here, just for space limitations.

5 Conclusion

We have defined the small step operational semantics of a simple Java-like language in SSI IR, equipped with an **instanceof** operator for runtime typechecks, and shown how precise type analysis of branches guarded by runtime typechecks can be achieved by abstract compilation in the presence of union and nominal types, and by suitably defining two predicates *inter* and *diff* that provide the type theoretic interpretation of σ-functions.

Despite the use of nominal types, the analysis is more precise than that we would get from the type system of a statically typed language as Java and C#; however, using structural types to trace the type of each field object leads to a more precise analysis, as already shown in previous work [5, 6]. Here we have preferred to keep the presentation simpler, but we envisage no problems in extending the compilation scheme and the definition of the predicates *inter* and *diff* to accommodate structural types.

For what concerns future developments, we are planning to extend the abstract compilation scheme proposed here to support subtyping constraints, to make the analysis more precise and efficient; we do not expect major problems in implementating in CHR [10] a constraint solver for subtyping between set of nominal types [1].

The approach proposed here seems promising also for other kinds of conditions occurring often in object-oriented programs; for instance, SSI can enhance

static analysis of null pointer references when branches are guarded by conditions of the form `(`r `== null)`.

References

1. A. Aiken. Introduction to set constraint-based program analysis. *SCP*, 35:79–111, 1999.
2. C. S. Ananian. The static single information form. Technical Report MITLCS-TR-801, MIT, 1999.
3. D. Ancona, M. Ancona, A Cuni, and N. Matsakis. RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *DLS'07*, pages 53–64. ACM, 2007.
4. D. Ancona, A. Corradi, G. Lagorio, and F. Damiani. Abstract compilation of object-oriented languages into coinductive CLP(X): can type inference meet verification? In *FoVeOOS 2010, Revised Selected Papers*, volume 6528 of *LNCS*. Springer Verlag, 2011.
5. D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In *ECOOP 2009*, volume 5653 of *LNCS*, pages 2–26. Springer Verlag, 2009. Best paper prize.
6. D. Ancona and G. Lagorio. Idealized coinductive type systems for imperative object-oriented programs. *RAIRO - Theoretical Informatics and Applications*, 45(1):3–33, 2011.
7. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13:451–490, 1991.
8. D. Das and U. Ramakrishna. A practical and fast iterative algorithm for phi-function computation using DJ graphs. *TOPLAS*, 27(3):426–440, 2005.
9. B. Alpern et. al. The jalapeño virtual machine. *IBM Systems Journal*, 39, 2000.
10. T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.
11. R. Griesemer and S. Mitrovic. A compiler for the java hotspottm virtual machine. In *The School of Niklaus Wirth, "The Art of Simplicity"*, pages 133–152, 2000.
12. G. Holloway. The machine-SUIF static single assignment library. Technical report, Harvard School of Engineering and Applied Sciences, 2001.
13. D. Novillo. Tree SSA - a new optimization infrastructure for GCC. In *GCC Developers' Summit*, pages 181–193, 2003.
14. L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP 2007*, pages 472–483, 2007.
15. L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In *ICLP 2006*, pages 330–345, 2006.
16. J. Singer. Static single information form in machine SUIF. Technical report, University of Cambridge Computer Laboratory, UK, 2004.
17. J. Singer. *Static Program Analysis based on Virtual Register Renaming*. PhD thesis, Christs College, 2005.
18. A. Tavares, F.M. Pereira, M. Bigonha, and R. Bigonha. Efficient SSI conversion. In *SBLP 2010*, 2010.
19. J. Winther. Guarded type promotion (eliminating redundant casts in Java). In *FTfJP 2011*. ACM, 2011.