Static Single Information Form for Abstract Compilation

Davide Ancona, Giovanni Lagorio

Static single information form for abstract compilation*

Davide Ancona and Giovanni Lagorio

DIBRIS, Università di Genova, Italy
{Davide.Ancona,Giovanni.Lagorio}@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 infer more precise types in dynamic object-oriented languages, where explicit runtime typechecking is frequently used.
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 dynamic object-oriented language in SSI form with a runtime typechecking operator, to show how precise type inference 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) [8] intermediate representation (IR) of code. Most modern compilers (among others, GNU's GCC [15], the SUIF compiler system [14], Java HotSpot [12], and Java Jikes RVM [10]) and formal software development tools implement efficient algorithms for translating code in advanced forms of IR particularly suitable for static analysis, thus offering the concrete opportunity of exploiting such IRs to obtain more precise type analysis and inference, and to fruitfully reuse those software components devoted to IR generation.

Abstract compilation [5,4,6] is a modular approach to static type analysis aiming to reconcile types and symbolic execution: an expression e is well-typed if the goal generated by compiling e succeeds w.r.t. the coinductive[1] model of the constraint logic program obtained by compiling the source program in which the expression is executed. In such a model terms are types representing possibly infinite sets of values, and goal resolution corresponds to symbolic execution.

* This work has been partially supported by MIUR DISCO - Distribution, Interaction, Specification, Composition for Object Systems.

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

Abstract compilation is particularly suited for implementing type inference and global type analysis of dynamic object-oriented languages in a modular way since one can provide several compilation schemes for the same language, each corresponding to a different kind of analysis, without changing the inference engine, which typically implements coinductive constraint logic programming [17,16,4]. For instance, in previous work we have defined compilation schemes based on union and structural object types, to support parametric and data polymorphism, (that is, polymorphic methods and fields) to obtain precise type analysis, and a smooth integration with the nominal type annotations contained in the programs and the inferred structural types [5]; other proposed compilation schemes aim to detect uncaught exceptions [4], or to integrate SSA IR in the presence of imperative features [6].

In this paper we exploit static single information (SSI), an extension of SSA proposed in literature [2,19], to allow more precise type inference in conditional branches guarded by runtime typechecks. SSI has been already adopted by compiler frameworks as LLVM [20], PyPy [3], and SUIF [18], and proved to be more effective than SSA for performing data flow analysis, program slicing, and interprocedural analysis. Similar IRs are adopted as well in formal software development tools.

We show how SSI can be exploited rather easily and effectively by abstract compilation to improve type inference of dynamic object-oriented languages, where explicit runtime typechecks are frequently used.

To this aim, we formally define the operational semantics of a simple dynamic object-oriented 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 inference 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 dynamic object-oriented 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.

[2] Available at `ftp://ftp.disi.unige.it/person/AnconaD/tcs12long.pdf`

Type analysis with static single assignment form

Method `read()` declared below, in a dynamic object-oriented language, creates and returns a shape which is read through method `nextLine()` 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());
      return st;
   }
}
```

Although method `read()` is type safe, no type can be inferred for `sh` to correctly typecheck the method; indeed, when method `area()` is invoked, variable `sh` may hold an instance of `Circle` or `Square`, therefore the most precise type that can be correctly assigned to `sh` is `Circle` $\vee$ `Square`. However, if `sh` 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. One can either define a rather sophisticated flow-sensitive type system, where each occurrence of a single variable can be associated with a different type, or typecheck the SSA IR, in which the method can be compiled.

In an SSA IR the value of each variable is determined by exactly one assignment statement [8]. 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 [19] 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()` there are three virtual registers ($\texttt{sh}_0$, $\texttt{sh}_1$ and $\texttt{sh}_2$) for the variable `sh`.

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

To transform a program into SSA form, a pseudo-function, conventionally called φ-function, needs to be introduced to correctly deal with merge points. For instance, in block `b5` 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 expression $\varphi(\texttt{sh}_0,\texttt{sh}_1)$ simply keeps track of the fact that the value of $\texttt{sh}_2$ is determined either by $\texttt{sh}_0$ or $\texttt{sh}_1$.

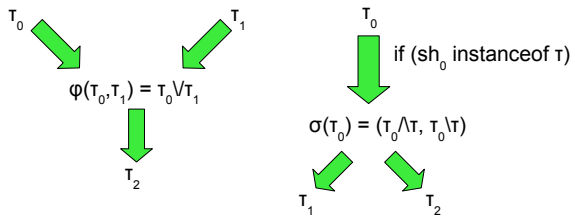

Fig. 1. Type theoretic interpretation of φ-function and σ-function

At the level of types, the φ-function naturally corresponds to the union type constructor (Figure 1): 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. That is, for the types shown in the figure, $\tau_0, \tau_1 \leq \tau_0 \vee \tau_1 \leq \tau_2$.

Thanks to pretty standard and efficient algorithms for transforming source programs into SSA IR [8,9], the flow analysis phase, where source code is transformed into IR, can be kept separate from the subsequent type analysis phase, favoring simplicity and reuse. Indeed, flow analysis and consequent transformation into IR is implemented by most compilers and formal software development tools. Abstract compilation makes such an approach even more modular, by dividing the overall process into three separate stages. First, the source code is transformed into a suitable IR. Then, the IR is compiled into a set of Horn clauses and a goal. Finally, the goal is resolved with an appropriate inference engine (typically, implementing coinductive constraint logic programming [4]).

This paper mainly focuses on the second stage of the overall process (that is, compilation from IR into Horn clauses), in the particular case when the adopted IR allows precise flow analysis, as happens with SSI, of dynamic object-oriented languages.

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

The method is transformed into 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 $\mathtt{sh}_0$. As a consequence, the only possible type that can be correctly associated with $\mathtt{sh}_0$ is `Square`, thus making the method of little use. However, this problem can be addressed if one considers the SSI IR of the method [2,19].

```
  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 a pseudo function, called σ-function. As a consequence, suitable virtual registers

and a σ-function have to be introduced also for the read-only pseudo-variable **this**.

The notion of σ-function is the dual of φ-function (Figure 1); the type theoretic interpretation of σ depends on the specific kind of guard. If it is a runtime typecheck (of the form `(sh`$_0$ **instanceof** `Square)` as in the example), then σ splits the type τ_0 of `sh`$_0$ in the type $\tau_0 \wedge$`Square`, assigned to `sh`$_1$, and in the type $\tau_0 \backslash$`Square`, assigned to `sh`$_2$, where the intersection and the complement operators have to be properly defined (see Section 4). For instance, if `sh`$_0$ has type `Square`$\vee$`Circle`, then `sh`$_1$ has type `(Square`$\vee$`Circle)`$\wedge$`Square=Square`, and `sh`$_2$ has type `(Square`$\vee$`Circle)`$\backslash$`Square=Circle`, therefore `Square`$\vee$`Circle` turns out to be a valid type for the parameter `sh`$_0$ of the method `largerThan`.

For what concerns **this**, in this particular example no real split would be necessary: **this**$_0$ has type `Square`, so `Square` is split into `(Square,Square)`, that is, both **this**$_1$ and **this**$_2$ have the same type `Square`.

3 Language definition

In this section we formally define an SSI IR for a simple dynamic object-oriented language equipped with an **instanceof** operator for performing runtime typechecking. Even though we have chosen a familiar Java-like syntax both for the IR and the source code used in the examples, the language is fully dynamic: code does not contain any type annotation, hence, under this point of view the language is quite different from Java.

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

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, main expression and declared parameters cannot be `this`.

Fig. 2. SSI intermediate language

A program is a collection of class declarations followed by an anonymous main method with no parameters and contained in an anonymous class (conventionally its fully qualified name is $\epsilon.\epsilon$), whose body is a sequence of blocks (see the comments on method bodies below).

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. *Object* is the usual predefined root class of the inheritance tree; every class comes equipped with the implicit constructor with parameters corresponding to all fields, in the same order as they are inherited and declared. For simplicity, no user constructors can be declared.

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.[3]

Virtual registers have the form x_i, where x is the corresponding variable. If r is a virtual register, then $var(r)$ returns the variable the register refers to, therefore if $r = x_i$, then $var(r) = x$. The receiver object can be referred inside method bodies with the special implicit parameter `this`, hence the IR contains virtual registers of the form $\texttt{this}_i$.

Besides usual statements and expressions, we consider φ and σ pseudo-function assignments. Conditional jumps contain σ-functions which split each virtual register r occurring in either branches into two new distinct versions used in the blocks labeled by $\texttt{l}_1$, and $\texttt{l}_2$, respectively. The guard can only be of the form (r `instanceof` c); however, more elaborated guards can be easily expressed in terms of this primitive one by suitable transformations during the compilation from the source code to the IR. Depending on the types and abstract compilation scheme, there could be other kinds of guards for which SSI would improve type analysis; for instance, if one includes the type corresponding to the null references, then a guard of the form `(`r `==` **null**`)` would take advantage of SSI to enhance null reference analysis. For those guards for which no type refinement is possible the σ-function performs no split, that is, $\sigma\texttt{(}\tau\texttt{)=(}\tau\texttt{,}\tau\texttt{)}$.

Semantics: To define the small step semantics of the language we first need to specify values v (see Figure 3), which are just identities o of dynamically created objects. Furthermore, we add *frame expressions* $ec\{e\}$, where ec is an execution context; frame expressions are *runtime expressions* 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 fully qualified name μ. A frame expression $\langle fr, \mu\rangle\{e\}$ 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,

[3] Such a constraint does not imply any loss of generality, since it is always possible to add new virtual registers and to insert a φ-function when the source code contains multiple returns.

fr is the stack frame of the method call, and $\mu = c.m$ is the fully qualified name of the method.

$$
\begin{array}{rl}
v ::= & o \quad \text{(values)} \\
e ::= & v \mid ec\{e\} \mid \ldots \quad \text{(runtime expressions)} \\
ec ::= & \langle fr, \mu \rangle \quad \text{(execution context)} \\
fr ::= & \overline{r \mapsto_t v}^{k} \quad \text{(stack frames)} \\
\mu ::= & c.m \quad \text{(full method names)} \\
\mathcal{H} ::= & \overline{o \mapsto \langle c, \overline{f \mapsto v}^{j} \rangle}^{k} \quad \text{(heaps)} \\
\mathcal{C}[\cdot] ::= & [\cdot] \mid ec\{\mathcal{C}[\cdot]\} \mid \texttt{new}\ c(\overline{v}^{n}, \mathcal{C}[\cdot], \overline{e}^{j}) \mid \mathcal{C}[\cdot].f \mid \mathcal{C}[\cdot].m(\overline{e}^{k}) \mid v_0.m(\overline{v}^{j}, \mathcal{C}[\cdot], \overline{e}^{k}) \\
& \mid \mathcal{C}[\cdot]; e \mid x = \mathcal{C}[\cdot] \mid \mathcal{C}[\cdot].f = e \mid v.f = \mathcal{C}[\cdot] \\
& \mid \texttt{if}\ (\mathcal{C}[\cdot])\ \texttt{with}\ \overline{(x', x'') = \sigma(x''')}^{n}\ \texttt{jump}\ l_1\ \texttt{else jump}\ l_2
\end{array}
$$

Fig. 3. Syntactic definitions instrumental to the operational semantics

Stack frames fr map virtual registers to their corresponding values. Each association is labeled with a distinct time-stamp t, which specifies how recently the register has been updated (higher time-stamp values correspond to more recent updates). Such labels are used to define the semantics of φ-function assignments.

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 4 shows the execution rules. Three different judgments are defined: the main judgment $\overline{cd}^{n}\ \{\overline{b}^{n}\} \Rightarrow v$ states that the main method $\{\overline{b}^{n}\}$ of program $\overline{cd}^{n}$ evaluates to value v. Such a judgment directly depends on the auxiliary judgment $\mathcal{H} \vdash e \rightarrow \mathcal{H}', e'$, stating that e rewrites to e' in $\mathcal{H}$, yielding the new heap $\mathcal{H}'$, and whose definition uses the auxiliary judgment $\mathcal{H}, ec \vdash e \rightarrow \mathcal{H}', ec', e'$, having the meaning that redex e rewrites to e' in $\mathcal{H}$ and ec, yielding the new execution context ec' and heap $\mathcal{H}'$. All the auxiliary judgments and functions[4] should be parametrized by the whole executing program, $\overline{cd}^{n}$, which, however, is kept implicit to favor readability.

Rule (main) defines the main judgment; a value v is returned if the runtime expression $\langle \epsilon_{fr}, \epsilon.\epsilon \rangle \{e\}$, where e is the first block (retrieved by the auxiliary function *firstBlock*) of the main method, transitively rewrites to v (and a heap $\mathcal{H}$ which is discarded). The evaluation context $\langle \epsilon_{fr}, \epsilon.\epsilon \rangle$ of the frame expression specifies that initially the frame (ϵ_{fr}) is empty (neither `this`, nor parameters are accessible), and that execution starts in the main method, whose fully qualified name is $\epsilon.\epsilon$ (recall that, conventionally, the main method is anonymous and is contained in an anonymous class).

The auxiliary judgment $\mathcal{H} \vdash e \rightarrow \mathcal{H}', e'$ is defined by the three rules (methcall) (a new execution context is created), (ctx) (evaluation continues in the currently active execution context), and (return) (the current execution context is closed).

[4] The straightforward definitions of the auxiliary functions have been omitted.

$$\text{(main)}\frac{\mathit{firstBlock}(\epsilon.\epsilon) = e \qquad \epsilon_{\mathcal{H}} \vdash \langle \epsilon_{fr}, \epsilon.\epsilon\rangle\{e\} \rightarrow^* \mathcal{H}, v}{\overline{cd}^n\ \{\overline{b}^n\} \Rightarrow v}$$

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

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

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

$$\text{(seq)}\frac{}{\mathcal{H}, ec \vdash v; e \rightarrow \mathcal{H}, ec, e} \qquad \text{(reg-asn)}\frac{}{\begin{array}{l}\mathcal{H}, \langle fr, \mu\rangle \vdash\ r = v \\ \qquad \rightarrow \mathcal{H}, \langle fr[v/r], \mu\rangle, v\end{array}}$$

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

$$\text{(phi)}\frac{}{\mathcal{H}, \langle fr, \mu\rangle \vdash r_0 = \varphi(\overline{r}^n) \rightarrow \mathcal{H}, \langle fr[\mathit{mru}(fr, \overline{r}^n)/r_0], \mu\rangle, v}$$

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

Fig. 4. Small-step semantics

In rule (meth-call), the object referenced by o is retrieved from the heap to find its class, c. Then, the auxiliary functions *firstBlock* and *params* return the first block of the method and its parameters, respectively. The result of the evaluation is a frame expression, where the new stack frame maps parameters to their corresponding arguments, and $\texttt{this}_0$ to o, the fully qualified name $c.m$ corresponds to the invoked method, and the expression is the first block of the method. Rule (ctx) deals with context closure. Contexts (the standard definition is in Figure 3) correspond to a deterministic call-by-value and left-to-right evaluation strategy. A single computation step in the current execution context (corresponding to the most nested frame expression) is performed. The active execution context is extracted by *currentEC*; then, if the redex e rewrites to e' yielding $\mathcal{H}'$ and ec' (see the other rules defining the auxiliary evaluation judgment), then the $\mathcal{C}[e]$ 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 active execution context with the new execution context ec'. In rule (return) the current execution context is closed, the heap is unaffected, and the result is the value associated with the returned virtual register r in the frame of the closing context.

The remaining rules define the auxiliary judgment $\mathcal{H}, ec \vdash e \rightarrow \mathcal{H}', ec', e'$, one for any distinct kind of redex. In rule (reg) a virtual register is accessed by extracting the corresponding value from the stack frame fr.Variable and field assignments evaluate to their right values; rule (reg-asn) has the side effect of updating, in the current stack frame fr, the value of the virtual register r and its associated time-stamp (this is implicit in the definition of $fr[v/r]$) since, after the assignment, r becomes the most recently updated register. Rule (fld-asn) deals with field assignments: the object referenced by o is retrieved from the heap, and its value updated. In rule (seq) the left-hand-side value in a sequence expression is discarded to allow evaluation to proceed with the next expression. In rule (phi), register r_0 is updated with the value (denoted by $mru(fr, \overline{r}^n)$) of the most recently updated register in the stack frame, between $\overline{r}^n$. In rule (new) 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 values passed to the constructor. In rule (fld-acc) field accesses are evaluated: the object is retrieved from the heap, and the resulting expression is the value of the selected field. Rules (jump) and (if) deal with unconditional and conditional jumps, respectively. The evaluation of a jump returns the expression e contained in the block labeled by l' in the method μ of the current execution context. The conditional jump (rule (if)) selects which branch to execute and which virtual registers have to be updated, depending on whether the value $fr(r)$ of the register r is a reference to an object of a subclass of c. If it is the case, then the returned expression is that labeled by l_1 and the virtual registers $\overline{r'}^n$ are updated; otherwise, the returned expression is that labeled by l_2 and the virtual registers $\overline{r''}^n$ are updated.

As an example, let us consider the expression $\texttt{x}_0$**=new** C**();return** $\texttt{x}_0$ in a program where C is defined and has no fields.

Then $\epsilon_{\mathcal{H}} \vdash \langle \epsilon_{fr}, \epsilon.\epsilon \rangle \{\mathtt{x_0}\mathtt{=}\textbf{new } \mathtt{C()};\textbf{return } \mathtt{x_0}\} \rightarrow \mathcal{H}, \langle \epsilon_{fr}, \epsilon.\epsilon \rangle \{\mathtt{x_0}\mathtt{=}o;\textbf{return } \mathtt{x_0}\}$ by rules (ctx) (with context $\mathtt{x_0}\mathtt{=}[\cdot];\textbf{return } \mathtt{x_0}$) and (new), where $\mathcal{H} = o \mapsto \langle \mathtt{C}, \epsilon \rangle$;
$\mathcal{H} \vdash \langle \epsilon_{fr}, \epsilon.\epsilon \rangle \{\mathtt{x_0}\mathtt{=}o;\textbf{return } \mathtt{x_0}\} \rightarrow \mathcal{H}, \langle fr, \epsilon.\epsilon \rangle \{o;\textbf{return } \mathtt{x_0}\}$ by rules (ctx) (with context $[\cdot];\textbf{return } \mathtt{x_0}$) and (var-asn), where $fr = \mathtt{x} \mapsto o, \mathtt{x_0} \mapsto o$;
$\mathcal{H} \vdash \langle fr, \epsilon.\epsilon \rangle \{o;\textbf{return } \mathtt{x_0}\} \rightarrow \mathcal{H}, \langle fr, \epsilon.\epsilon \rangle \{\textbf{return } \mathtt{x_0}\}$ by rules (ctx) (with context $\langle fr, \epsilon.\epsilon \rangle \{[\cdot]\}$) and (seq); finally, $\mathcal{H} \vdash \langle fr, \epsilon.\epsilon \rangle \{\textbf{return } \mathtt{x_0}\} \rightarrow \mathcal{H}, o$ by rule (return) (with context $[\cdot]$).

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 and inference amounts to coinductive resolution of B, that is the greatest Herbrand model of *Hf* [5] is considered. The proof of soundness of such a translation is sketched in the Appendix.

In previous work [5,6] we have used expressive structural types; however, since SSI favors more precise type analysis, we have preferred to follow a simpler approach based on nominal types. Structural types could be used as well to allow data polymorphism, with the downside that subtyping relation becomes much more involved and termination issues must be addressed.

Subtyping is treated as an ordinary predicate, thus allowing only global analysis; compositional analysis can be obtained by considering subtyping as a constraint, and by using coinductive constraint logic programming [4].

The compilation of programs, class, and method declarations is defined in Figure 5. 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.

Each rule defines a different compilation judgment. The judgment $\overline{cd}^n \ e \rightsquigarrow (Hf^d \cup \overline{Hf}^n | B)$ states that the program $\overline{cd}^n \ e$ is compiled into the pair $(Hf^d \cup \overline{Hf}^n | B)$, where $Hf^d \cup \overline{Hf}^n$ is a Horn formula (that is, a set of Horn clauses), and B is a goal.[5] Type inference of the main expression is obtained by coinductive resolution of the goal B in $Hf^d \cup \overline{Hf}^n$. The Horn formula Hf^d contains all clauses that are independent of the program (Figure 7), whereas each Hf_i is obtained by compiling the class declaration cd_i (see below); the goal B is generated from the compilation of the main expression e (see below); the term t corresponds to the returned type of e; it is ignored here, but it is necessary for compiling expressions.

The compilation of a class declaration $\mathtt{class}\ c_1\ \mathtt{extends}\ c_2\ \{\ \overline{f}^n\ \ \overline{md}^k\ \}$ is a set of clauses, including each clause Hf_i obtained by compiling the method

[5] 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 $\mathit{false} \leftarrow B$.

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

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

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

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

Fig. 5. Compilation of programs, class, and method declarations and bodies.

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 $new(ce, c, [\overline{t}^n])$ 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 (Figure 6): it is a finite map (simply represented by a list) associating class names with field records (finite maps again simply represented by lists) assigning types to all fields of a class. 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 part h of the 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 7 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 also defined in Figure 7.

The judgment $m(\overline{r}^n)\{\overline{b}^n\}$ **in** $c \rightsquigarrow Hf$ states that the method declaration $m(\overline{r}^n)\{\overline{b}^n\}$ contained in class c compiles to Horn clauses Hf. 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 Figure 7), whereas the second is obtained by compiling the body of the method. The atom $has\_meth(ce, c, m, [t_0, \overline{t}^n], t)$ 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 7) ensures that the method can be invoked only on objects that are instances of c or one of its subclasses. For simplicity we assume that all names (including `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 6 defines abstract compilation for blocks, and expressions.

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

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

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

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

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

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

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

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

$$\text{(var)}\frac{}{r \rightsquigarrow (r \mid \mathit{true})}$$

Fig. 6. 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 the σ-functions; predicates *inter* and *diff* (defined in Figure 7) compute the

intersection T and the difference F between the type of r and c, respectively, and predicate *var_upd* (defined in Figure 7) 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 7). 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 7). The generated atoms are those obtained from the compilation of the sub-expressions, together with the atom specific of the expression. Rules (var) is straightforward.

The clauses (see Hf^d in (prog)) that do not depend on the specific program are defined in Figure 7 in the Appendix.

Predicate *subtype* defining the subtyping relation deserves some comments: as expected, classes c_1 and c_2 are both subtypes of $c_1 \vee c_2$, but c_1 is not a subtype of c_2 when C_1 is a proper subclass of c_2: since no rule is imposed on method overriding, subclassing is not subtyping. 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` cannot 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.

Type *empty* is the bottom of the subtyping relation. The predicates *inter* and *diff* (see below) can generate *empty* when a branch is unreachable. Of course, field accesses and method invocations on type *empty* are correct (in practice they can only occur in dead code).

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.

Predicates *inter* and *diff* define type splitting for σ-functions; both predicates never fail, but the type *empty* is returned when a branch is not reachable. Their definition is straightforward: the former keeps all classes that satisfies the runtime typechecking, the latter all classes that do not satisfy it.

5 Conclusion

We have shown how SSI IR can be exploited by abstract compilation for static global type analysis of programs written in a dynamic object-oriented language equipped with an **instanceof** operator for runtime typechecks. The approach allows a rather precise analysis with just nominal and union types.

We have stated soundness of type inference and sketched the proof; to do that, a small step operational semantics of the SSI IR language has been formally defined; this is also an original contribution, since, to our knowledge, no prior formal definition of the semantics of an SSI IR language can be found in literature.

There already exists interesting work on type inference of dynamic object-oriented languages, and several papers have defined and studied flow sensitive type systems [7,1,11,13] (to mention just a few). The main distinguishing feature of abstract compilation, when compared with all other approaches, is its modularity. Abstract compilation allows one to implement different kinds of analysis, for different languages with a suitable compilation scheme, by using the same inference engine. Furthermore, abstract compilation is particularly suited for directly compiling IR languages, as SSI, to greatly improve the precision of the analysis.

Guarded type promotion [21] for Java allows more precise type analysis of branches guarded by dynamic type checks in a very similar way as in our approach. However, here we consider the more challenging problem of inferring types for a dynamic language where no type annotations are provided by the programmers.

References

1. J. An, A. Chaudhuri, J.S. Foster, and M. Hicks. Dynamic inference of static types for Ruby. In *POPL*, pages 459–472, 2011.

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. C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *LNCS*, pages 428–452. Springer Verlag, 2005.
8. 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.
9. D. Das and U. Ramakrishna. A practical and fast iterative algorithm for phi-function computation using DJ graphs. *TOPLAS*, 27(3):426–440, 2005.
10. B. Alpern et. al. The jalapeño virtual machine. *IBM Systems Journal*, 39, 2000.
11. J.S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI*, pages 1–12, 2002.
12. 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.
13. P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *ECOOP 2010 - Object-Oriented Programming*, volume 6183 of *LNCS*, pages 200–224. Springer Verlag, 2010.
14. G. Holloway. The machine-SUIF static single assignment library. Technical report, Harvard School of Engineering and Applied Sciences, 2001.
15. D. Novillo. Tree SSA - a new optimization infrastructure for GCC. In *GCC Developers' Summit*, pages 181–193, 2003.
16. L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP 2007*, pages 472–483, 2007.
17. L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In *ICLP 2006*, pages 330–345, 2006.
18. J. Singer. Static single information form in machine SUIF. Technical report, University of Cambridge Computer Laboratory, UK, 2004.
19. J. Singer. *Static Program Analysis based on Virtual Register Renaming.* PhD thesis, Christs College, 2005.
20. A. Tavares, F.M. Pereira, M. Bigonha, and R. Bigonha. Efficient SSI conversion. In *SBLP 2010*, 2010.
21. J. Winther. Guarded type promotion (eliminating redundant casts in Java). In *FTfJP 2011*. ACM, 2011.

$class(object) \leftarrow true.$
$subclass(X, X) \leftarrow class(X).$
$subclass(X, Y) \leftarrow extends(X, Z), subclass(Z, Y).$
$subtype(empty, \_) \leftarrow true.$
$subtype(T, T) \leftarrow true.$
$subtype(T1 \vee T2, T) \leftarrow subtype(T1, T), subtype(T2, T).$
$subtype(T, T1 \vee \_) \leftarrow subtype(T, T1).$
$subtype(T, \_ \vee T2) \leftarrow subtype(T, T2).$
$field(CE, empty, \_, \_) \leftarrow true.$
$field(CE, C, F, T) \leftarrow has\_field(C, F), class\_fields(CE, C, R), field\_type(R, F, T).$
$field(CE, T1 \vee T2, F, FT1 \vee FT2) \leftarrow field(CE, T1, F, FT1),$
$\qquad field(CE, T2, F, FT2).$
$class\_fields([C : R|CE], C, R) \leftarrow no\_def(C, CE).$
$class\_fields([C1 : \_|CE], C2, R) \leftarrow class\_fields(CE, C2, R), C1 \neq C2.$
$field\_type([F{:}T|R], F, T) \leftarrow no\_def(F, R).$
$field\_type([F1 : \_|R], F2, T) \leftarrow field\_type(R, F2, T), F1 \neq F2.$
$no\_def(\_, [\,]) \leftarrow true.$
$no\_def(K1, [K2 : \_|Tl]) \leftarrow no\_def(K1, Tl), K1 \neq K2.$
$invoke(\_, empty, \_, \_, \_) \leftarrow true.$
$invoke(CE, C, M, A, RT) \leftarrow has\_meth(CE, C, M, [C|A], RT).$
$invoke(CE, T1 \vee T2, M, A, RT1 \vee RT2) \leftarrow invoke(CE, T1, M, A, RT1),$
$\qquad invoke(CE, T2, M, A, RT2).$
$new(\_, object, [\,]) \leftarrow true.$
$has\_field(C, F) \leftarrow dec\_field(C, F).$
$has\_field(C, F) \leftarrow extends(C, P), has\_field(P, F), \neg dec\_field(C, F).$
$has\_meth(CE, C, M, A, R) \leftarrow extends(C, P), has\_meth(CE, P, M, A, R),$
$\qquad \neg dec\_meth(C, M).$
$var\_upd(T1, T2) \leftarrow subtype(T2, T1).$
$field\_upd(CE, C, F, T2) \leftarrow field(CE, C, F, T1), subtype(T2, T1).$
$inter(C1, C2, C1) \leftarrow subclass(C1, C2).$
$inter(C1, C2, empty) \leftarrow \neg subclass(C1, C2).$
$inter(T1 \vee T2, C, IT1 \vee IT2) \leftarrow inter(T1, C, IT1), inter(T2, C, IT2).$
$diff(C1, C2, C1) \leftarrow class(C1), \neg subclass(C1, C2).$
$diff(C1, C2, empty) \leftarrow class(C1), subclass(C1, C2).$
$diff(T1 \vee T2, C, IT1 \vee IT2) \leftarrow diff(T1, C, IT1), diff(T2, C, IT2).$

Fig. 7. Clauses defining the predicates used by the abstract compilation

A Proof of soundness

To sketch the proof of soundness of abstract compilation, abstract compilation of expressions has to be extended to cover also runtime expressions, hence we define the new judgment $\mathcal{H}, fr, e \rightsquigarrow (t \mid B)$ (defined in Figure 8) stating that the runtime expression e compiles to the type t and the conjunction of atoms B in the heap $\mathcal{H}$ and stack frame fr. Heaps and stack frames are needed for compiling object values and virtual registers. All rules obtained as straightforward extension of the corresponding rules in Figure 6 have been omitted. In the sequel, all

$$(\text{obj})\frac{\mathcal{H}(o) = \langle c, \_\rangle}{\mathcal{H}, fr, o \rightsquigarrow (c \mid true)} \qquad (\text{frame})\frac{\mathcal{H}, fr', e \rightsquigarrow (t \mid B)}{\mathcal{H}, fr, \langle fr', \mu\rangle\{e\} \rightsquigarrow (t \mid B)}$$

$$(\text{c-jmp-then})\frac{fr(r) = c' \quad c' \leq c \quad \forall\, i = 1..n\ fr(r'_i) = t'_i \quad \begin{array}{l}\text{if } var(r'''_i) = var(r)\\ \text{then } t''_i = c' \text{ else } t''_i = fr(r'''_i)\end{array}}{\mathcal{H}, fr, \texttt{if } (r \texttt{ instanceof } c) \texttt{ with } \overline{(r', r'') = \sigma(r''')}^{n} \texttt{ jump } l_1 \texttt{ else jump } l_2 \textbf{ in } \rightsquigarrow (void \mid \overline{var\_upd(t', t'')}^{n})}$$

$$(\text{c-jmp-else})\frac{fr(r) = c' \quad c' \not\leq c \quad \forall\, i = 1..n\ fr(r''_i) = t'_i \quad \begin{array}{l}\text{if } var(r'''_i) = var(r)\\ \text{then } t''_i = c' \text{ else } t''_i = fr(r'''_i)\end{array}}{\mathcal{H}, fr, \texttt{if } (r \texttt{ instanceof } c) \texttt{ with } \overline{(r', r'') = \sigma(r''')}^{n} \texttt{ jump } l_1 \texttt{ else jump } l_2 \textbf{ in } \rightsquigarrow (void \mid \overline{var\_upd(t', t'')}^{n})}$$

$$(\text{var-upd})\frac{\mathcal{H}, fr, e \rightsquigarrow (t \mid B) \quad fr(r) = t'}{\mathcal{H}, fr, r = e \rightsquigarrow (t \mid B, var\_upd(t', t))}$$

$$(\text{phi})\frac{\forall\, i = 0..n\ fr(r_i) = t_i}{\mathcal{H}, fr, r_0 = \varphi(\overline{r}^{n}) \rightsquigarrow (\overline{\vee t}^{n} \mid var\_upd(t_0, \overline{\vee t}^{n}))} \qquad (\text{var})\frac{fr(r) = t}{\mathcal{H}, fr, r \rightsquigarrow (t \mid true)}$$

Fig. 8. Compilation of runtime expressions

statements depend on a particular program $\overline{cd}^{n}$; furthermore, we state that the coinductive resolution of a goal succeeds to mean that it succeeds w.r.t. the abstract compilation of $\overline{cd}^{n}$.

Lemma 1 (Progress). *If $\mathcal{H}, \epsilon_{fr}, e \rightsquigarrow (t \mid B)$, and the coinductive resolution of B succeeds, then either e is a value, or there exist $\mathcal{H}'$ and e' s.t. $\mathcal{H} \vdash e \to \mathcal{H}', e'$.*

Lemma 2 (Subject Reduction). *If $\mathcal{H} \vdash e \to \mathcal{H}', e'$ and $\mathcal{H}, \epsilon_{fr}, e \rightsquigarrow (t \mid B)$ and the coinductive resolution of B succeeds with grounding substitution θ, then there exist t' and B' s.t. $\mathcal{H}, \epsilon_{fr}, e' \rightsquigarrow (t' \mid B')$, the coinductive resolution of B' succeeds with grounding substitution θ', and subtype$(\theta' t', \theta t)$ succeeds.*

Theorem 1 (Soundness). *If $\mathcal{H}, \epsilon_{fr}, e \rightsquigarrow (t \mid B)$ and the coinductive resolution of B succeeds with grounding substitution θ, and $\mathcal{H} \vdash e \to^{*} \mathcal{H}', e'$, and there exist no $\mathcal{H}''$ and e'' s.t. $\mathcal{H}' \vdash e' \to \mathcal{H}'', e''$, then e' is an object value o s.t. $\mathcal{H}'(o) = \langle c, \_\rangle$, and subtype$(c, \theta t)$ succeeds.*