**Theoretical Informatics and Applications** Will be set by the publisher
Informatique Théorique et Applications

# IDEALIZED COINDUCTIVE TYPE SYSTEMS FOR IMPERATIVE OBJECT-ORIENTED PROGRAMS

Davide Ancona[1] and Giovanni Lagorio[1]

**Abstract**. In recent work we have proposed a novel approach to define idealized type systems for object-oriented languages, based on *abstract compilation* of programs into Horn formulas which are interpreted w.r.t. the coinductive (that is, the greatest) Herbrand model.

In this paper we investigate how this approach can be applied also in the presence of imperative features. This is made possible by considering a natural translation of *Static Single Assignment* intermediate form programs into Horn formulas, where $\varphi$ functions correspond to union types.

**1991 Mathematics Subject Classification.** 03B70, 03B15.

## 1. Introduction

Precise type inference of object-oriented programs relies on the ability of allowing assignments of values of unrelated types to a field (*data polymorphism*) and invocations of a method where arguments of unrelated types can be passed to the same parameter (*parametric polymorphism*) [1].

While most proposed solutions to type inference of object-oriented programs [1,7,17,24,27,34,35] support parametric polymorphism well, only few of them are able to deal properly with data polymorphism; moreover, such solutions turn out to be quite complex and specific, cannot be easily specified in an abstract way, and lack a common logical framework and inferential engine.

In recent work [4,8] we have proposed a novel approach to define idealized coinductive type systems for object-oriented languages, where programs are encoded into Horn formulas which are interpreted w.r.t. the coinductive (that is, the greatest) Herbrand model. Coinduction is needed since types (which are terms of the

*Keywords and phrases:* Imperative object-oriented languages, type analysis, coinduction, SSA intermediate form.

[1] DISI, University of Genova, Italy; e-mail: {davide,lagorio}@disi.unige.it

Herbrand coinductive universe) can be infinite terms; in this way, by using union types [10], it is possible to represent infinite sets of polymorphic values. The resulting type system is idealized, since terms are not required to be regular, that is, they are infinite (finitely branching) trees which may contain infinite different subtrees, and, therefore, cannot be finitely represented as happens with regular trees [12]. As a consequence, any implementation can only be sound, but not complete w.r.t. the defined type system which, in a sense, pushes to the extreme the theoretical limits of static analysis.

The encoding we have studied [4,8] defines an idealized type system for a small functional object-oriented language similar to Featherweight Java [20], where type annotations may be omitted, and parametric and data polymorphism are fully supported. The main contribution of this paper is to define an idealized type system for an imperative version of the language. This is made possible by considering a natural encoding of *Static Single Assignment* (SSA) intermediate form programs [15] into Horn formulas, where $\varphi$ functions correspond to union types.

The paper is structured as follows. Section 2 contains a gentle introduction to abstract compilation and coinductive types, and introduces the basic definitions. In Section 3 we show how abstract compilation can take advantage of the SSA intermediate form to perform a more precise type inference in the presence of imperative features such as local variable updates and loops. Abstract compilation of field updates is more challenging, and therefore its treatment is postponed to Section 4. The formalization of abstract compilation for the intermediate SSA form of a simple imperative Java-like language can be found in Section 5, whereas Section 6 is devoted to the definition of the small-step operational semantics of the language and to the proof of soundness of abstract compilation. Finally, Section 7 concludes with an analysis of the related work, and a discussion on future research directions.

## 2. A gentle introduction to coinductive type systems

This section contains an overview on abstract compilation and coinductive types based on previous work [4,8] on type inference of object-oriented languages in a functional setting.

Abstract compilation allows static analysis, and, more specifically, type inference, by translating the program to be analyzed into a Horn formula *Hf* (that is, a conjunction of Horn clauses) which corresponds to a more abstract representation of the program, and by resolving a certain goal w.r.t. the conductive (that is, the greatest) Herbrand model of *Hf*.

As an example, let us assume that we would like to perform type inference for the following Java-like program, where type annotations have been deliberately omitted.

```
class EList extends Object {
 EList(){super();}
}
```

```
class NEList extends Object {
 el, next;
 NEList(e,n){super();el=e;next=n;}
}

class Fact {
  iter(i) {
    if(i<=0) return new EList();
    else return new NEList(i,this.iter(i-1));
  }
}
```

Classes `EList` and `NEList` implement empty and non empty linked lists,[1] respectively. Method `iter` of the factory class `Fact` builds a linked list of length `i` containing integer values as elements.

### 2.1. Abstract compilation of method declarations

Each method declared above can be abstractly compiled in a Horn clause corresponding to a more abstract (hence, approximate) semantics of the method. This is achieved by considering an Herbrand universe where terms are types (hence, a term does not represent a single value, but rather a set of values), and by introducing and defining predicates for each language constructs, together with auxiliary predicates needed for expressing the abstract semantics of the language.

More precisely, terms are either constants corresponding to field, method and class names, or type constructors. Even though abstract compilation is not tied to any particular kind of type constructor, previous work [4, 8] has proved that the coexistence of union [10, 19] and object types allow precise type inference of Java-like languages. Hence, throughout the paper we will use the following types:

- The two constant primitive types `bool` and `int`.[2]
- Object types `obj(`$c$`,[`$\mathtt{f}_1$`:`$t_1$`,...,`$\mathtt{f}_n$`:`$t_n$`])`, corresponding to all instances created from class $c$, with fields $f_1, \ldots, f_n$ associated with values of types $t_1, \ldots, t_n$, respectively; *data polymorphism* is supported, since the field of two instances of the same class can be associated with unrelated types. For instance, `obj(neList,[el:bool])` and `obj(neList,[el:int])` correspond to `NEList` instances whose first element is an integer and a boolean value, respectively. Fields are finite and distinct, and their order is immaterial.
- Union types $t_1 \vee t_2$, corresponding to all values of type either $t_1$ or $t_2$.
- Product types represented by list terms $[t_1, \ldots, t_n]$, and used for specifying the types of method parameters.

[1] To keep the example simple, we are considering a quite naive implementation of linked lists. In the examples we assume that numeric primitive data types are supported, even though the language formalized in Section 5 supports only Boolean values.

[2] For simplicity only `bool` will be considered in the formalization in Section 5.

Throughout the rest of the paper we will use the standard syntactic notations of logic programming for Horn clauses; for instance, logical variables begin with an upper case letter, while function and predicate symbols begin with a lower case one.

Predicates correspond to the language constructs; for instance, `invoke(`$t_0$`,`$m$`,[`$t_1$, ..., $t_n$`],`$t$`)` corresponds to invocation of method $m$ on target (a.k.a. receiver) object of type $t_0$ with arguments of types $t_1, \ldots, t_n$, and returned value of type $t$. As another example, `new(`$c$`,[`$t_1, \ldots, t_n$`],`$t$`)` corresponds to invocation of constructor of class $c$, with arguments of types $t_1, \ldots, t_n$, and returned value of type $t$.

Auxiliary predicates are introduced for defining the abstract semantics of the language; for instance, predicate `invoke` is defined in terms of the predicate `has_meth` corresponding to method look-up:

```
invoke(obj(C,R),M,A,RT) ← has_meth(C,M,[obj(C,R)|A],RT).
```

Invocation of method `M` on target of type `obj(C,R)` with arguments of type `A` returns a value of type `RT` if method look-up of `M` starting from class `C` with argument type `[obj(C,R)|A]` succeeds and returns a value of type `RT`. Note that the type of the target object is added at the beginning of the list[3] of argument types of the method.

The translation of a method declaration generates a new Horn clause for predicate `has_meth`. For instance, method `iter` of class `Fact` can be compiled into the following Horn clause:

```
has_meth(fact,iter,[This,int],L1∨L2) ←
    type_comp(This,fact),new(eList,[],L1),
    invoke(This,iter,[int],L),new(neList,[int,L],L2).
```

The body of the clause has been obtained by compiling the body of the corresponding method; for simplifying the example we have performed a simple optimization by removing the atoms corresponding to the expressions `i<=0` and `i-1`, which have types `bool` and `int`, respectively, and require `i` having type `int`. The method has two parameters `[This,int]`, where the first corresponds to the target object, and the second must necessarily be of type `int`. The target object must be an instance of class `Fact` or of one of its subclasses (`type_comp(This,fact)`), since the method might be inherited.

```
type_comp(obj(C1,R),C2) ← subclass(C1,C2).
```

The returned type is `L1∨L2`, where `L1` and `L2` are the types of the "then" and "else" branches, respectively, of the conditional statement. The "then" branch has been compiled into `new(eList,[],L1)`, the "else" into `invoke(This,iter,[int],L), new(neList,[int,L],L2)`.

### 2.2. Infinite terms, and the coinductive Herbrand model

Given the above example code, the invocation `x.iter(n)` is expected to be type safe whenever `x` and `n` contain an instance of class `Fact` and an integer value, respectively. Consequently, the goal `has_meth(fact,iter,[obj(fact,[]),int],T)`

[3]The term $[t_1 \mid t_2]$ denotes the list where $t_1$ is the first element and $t_2$ is the rest of the list.

should succeed for a substitution mapping `T` to a term $t$ corresponding to the type of the returned value.

Unfortunately, this is not true for the standard inductive Herbrand model [30,31]. There are two intimately related reasons for that. The first one is that the clause corresponding to method `iter` as defined above does not correspond to a well-founded inductive definition. Indeed, the recursive invocation **this**`.iter(i-1)` in the body of the method is compiled in the atom `invoke(This,iter,[int],L)`, and predicate `invoke` is defined in terms of predicate `has_meth`, hence, resolving `invoke(This,iter,[int],L)` amounts to resolve `has_meth(fact,iter,[obj(fact,[]),int],L)`. Therefore standard SLD resolution based on the inductive Herbrand model would diverge in this case, since there is no finite proof tree for the goal.

The other reason why the inductive Herbrand model does not work is that the logical variable `T` of the goal should be substituted with a type $t$ specifying the set of all lists of integers, and this set cannot be expressed with a finite term with union and object types. However, such a type can be easily expressed in the coinductive Herbrand model where terms can be infinite: $t$ is the unique solution of the following unification problem (see below, and Section 2.3):

$t =$`obj(eList,[])` $\vee$ `obj(neList,[el:int,next:`$t$`])`

The coinductive Herbrand model is defined in terms of the greatest fixed point operator, or equivalently, of possibly infinite proof trees [4,8,30,31]. Consequently, the atom $A =$`has_meth(fact,iter,[obj(fact,[]),int],`$t$`)` succeeds; indeed, given the above clauses defining predicate `invoke` and encoding method `iter`, one can verify that $A$ succeeds if the following four atoms succeed:

```
type_comp(obj(fact,[]),fact), new(eList,[],obj(eList,[])),
new(neList,[int,t],obj(neList,[el:int,next:t])),
invoke(obj(fact,[]),iter,[int],t).
```

The reader can easily verify that, by encoding the whole program on page 2 with the rules defined in Section 5, the first three atoms above succeed, while the last atom succeeds if $A$ succeeds, hence we can conclude by coinduction that the original goal succeeds.

We can now give the relevant formal definitions, based on the standard notion of infinite tree [2, 12].

**Definition 2.1.** A path $p$ is a finite and possibly empty sequence of natural numbers, that is, an element of the set $\mathbb{N}^*$. The empty path is denoted by $\epsilon$, $p_1 \cdot p_2$ denotes the concatenation of $p_1$ and $p_2$, and $|p|$ the length of $p$. For simplicity we consider $\mathbb{N}$ a subset of $\mathbb{N}^*$, hence depending on the context, $n$ may denote either a natural number, or the path of length 1 containing $n$.

**Definition 2.2.** A tree over a set $S$ is a partial function $t{:}\mathbb{N}^* \rightharpoonup S$ from paths to $S$ s.t.

(1) its domain, denoted by $dom(t)$, is not empty;
(2) $dom(t)$ is prefix-closed;
(3) for all paths $p \in dom(t)$, and $n \in \mathbb{N}$, $p \cdot (n+1) \in dom(t)$ implies $p \cdot n \in dom(t)$ and there exists $k$ s.t. $p \cdot k \notin dom(t)$.

Note that by 1 and 2 we have that $\epsilon \in dom(t)$ always holds, and by 3 $t$ is always finitely branching; that is, $t$ is infinite iff the set $\{|p| \mid p \in dom(t)\}$ has no upper bound.

A ranked alphabet $\Sigma_o$ associates an arity $n$ with each operation *op* (a.k.a. functor or function) symbol; more precisely, symbol *op* has arity $n$ in $\Sigma_o$ iff $(op, n) \in \Sigma_o$.

**Definition 2.3.** A ground term over $\Sigma_o$ is a tree $t$ over $\Sigma_o$ s.t. for all $p \in dom(t)$, if $t(p) = (op, n)$, then $p \cdot (n-1) \in dom(t)$ and $p \cdot n \notin dom(t)$. The set of ground terms over $\Sigma_o$ is called the *coinductive Herbrand universe over* $\Sigma_o$.

Let $\mathcal{X}$ be an enumerable set of variables disjoint from $\Sigma_o$. Terms over $\Sigma_o$ and $\mathcal{X}$ are easily defined by considering variables as symbols of arity 0, and by taking trees over $\Sigma_o \cup \mathcal{X}$.

A substitution $\theta$ is a total map from $\mathcal{X}$ to trees over $\Sigma_o \cup \mathcal{X}$.

**Definition 2.4.** If $p \in dom(t)$, then the subtree of $t$ rooted at $p$ is the tree $t'$ defined by $dom(t') = \{p' \mid p \cdot p' \in dom(t)\}$, $t'(p') = t(p \cdot p')$.

**Definition 2.5.** We denote with $t\theta$ the term obtained by substituting all occurrences of any variable $X$ (more precisely, all subtrees $t'$ of $t$ s.t. $t'(\epsilon) = X$) with $\theta(X)$. More formally,

- $dom(t\theta) = dom(t) \cup \{p \cdot p' \mid t(p) \in \mathcal{X}, p' \in dom(\theta(t(p)))\}$
- $(t\theta)(p) = \begin{cases} t(p) & \text{if } p \in dom(t), t(p) \notin \mathcal{X} \\ \theta(t(p'))(p'') & \text{if } p = p' \cdot p'',\ p' \in dom(t),\ t(p') \in \mathcal{X} \end{cases}$

A guarded equation over $\Sigma_o \cup \mathcal{X}$ is a syntactic equation $X = t$ [4, 8, 30, 31], where $t(\epsilon) \notin \mathcal{X}$, and $dom(t)$ is finite (that is, $t$ is not a variable and is finite).

A solution of $X = t$ is a substitution $\theta$ s.t. $\theta(X) = t\theta$. Such a definition can be naturally extended to a system of guarded equations, that is, an enumerable set $\{X_i = t_i \mid i \in I \subseteq \mathbb{N}\}$ of guarded equations s.t. $X_i = X_j$ implies $i = j$ for all $i, j \in I$.

**Proposition 2.6.** *Every tree can be represented by a system of guarded equations* $\{X_i = t_i \mid i \in I \subseteq \mathbb{N}\}$ *s.t. for some* $i \in I$, $\theta(X_i) = t$ *for any solution* $\theta$.

Note that the proposition stated above [12] is trivial for finite trees, but less obvious for infinite ones. As we will see below, there are infinite trees which can be represented by a finite number of guarded equations.

**Definition 2.7.** Given a ranked alphabet $\Sigma_p$ for predicate symbols, a ground atom $A$ over $\Sigma_o$ and $\Sigma_p$ is a term over $\Sigma_o \cup \Sigma_p$ s.t. $A(\epsilon) \in \Sigma_p$ and for all $n \in dom(A)$ $A(n)$ is a term over $\Sigma_o$. The set of ground atoms over $\Sigma_o$ and $\Sigma_p$ is called the *coinductive Herbrand base over* $\Sigma_o$ *and* $\Sigma_p$ and denoted by $\mathcal{HB}(\Sigma_o, \Sigma_p)$.

Atoms over a set of variables $\mathcal{X}$ are defined analogously as for terms. In the rest of the paper we assume that fixed $\Sigma_o$ and $\Sigma_p$ are given.

**Definition 2.8.** The *immediate consequence operator* $T_{Hf}$ associated with formula $Hf$ is the endofunction over $\mathcal{P}(\mathcal{HB}(\Sigma_o, \Sigma_p))$ defined as follows:

$$T_{Hf}(S) = \{A \mid A \leftarrow B \text{ is a ground instance of a clause of } Hf, B \in S\}.$$

A Herbrand model of *Hf* is a subset of $\mathcal{HB}(\Sigma_o, \Sigma_p)$ which is a fixed-point of $T_{Hf}$.

Since $T_{Hf}(S)$ is monotone by definition, by the Knaster-Tarski theorem there always exists the greatest fixed-point of $T_{Hf}$, which is called the coinductive Herbrand model of *Hf*, and is denoted by $M_{Hf}$. A goal $A_1, \ldots, A_n$ (a finite sequence of atoms) is coinductively derivable (we will simply write derivable in the rest of the paper) from *Hf* iff there exists a substitution $\theta$ s.t. for all $i = 1, \ldots, n$, $A_i\theta$ belongs to the coinductive Herbrand model of *Hf*.

The coinductive Herbrand model can be equivalently defined in terms of infinite proof trees.

**Definition 2.9.** A proof tree for *Hf* is a tree $t$ over $\mathcal{HB}(\Sigma_o, \Sigma_p)$ s.t. for all $p \in dom(t)$, if $p{\cdot}(n{-}1) \in dom(t)$ and $p{\cdot}n \notin dom(t)$, then $t(p) \leftarrow t(p{\cdot}0), \ldots, t(p{\cdot}n{-}1)$ is a ground instance of a clause of *Hf*.

We can now state the following proposition [21].

**Proposition 2.10.** *The set* $\{A \mid A = t(\epsilon), t$ *proof tree for Hf*$\}$ *is equal to the coinductive Herbrand model of Hf.*

### 2.3. Regular terms, subtyping and subsumption

The term $t =$`obj(eList,[])`$\vee$`obj(neList,[el:int,next:`$t$`])` introduced in the previous section can be finitely represented by a system with a finite number of guarded equations (in fact, in this example just an equation suffices) and is called *regular* (a.k.a. *rational*). Not all infinite terms in the coinductive Herbrand model are regular, hence, not all terms can be finitely represented. The same consideration applies for proof trees. Therefore, coinductive type systems are inherently *idealized* and allow only sound but not complete implementations. The notions of *regular term and proof* [12, 21, 31], *subtyping* and *subsumption* are introduced mainly for allowing sound approximations, so that infinite types and proof trees can be approximated with arbitrary precision by regular ones.

**Definition 2.11.** A regular tree is a possibly infinite tree containing a finite set of subtrees.

**Proposition 2.12.** *Every regular tree can be represented by a system with a finite number of guarded equations* $\{X_i = t_i \mid i \in I \subseteq \mathbb{N}, I$ *finite*$\}$ *s.t. for some* $i \in I$, $\theta(X_i) = t$ *for any solution* $\theta$.

The goal `has_meth(fact,iter,[obj(fact,[]),int],T)` presented in Section 2.2 is derivable for `T=`$t$ (with $t$ regular term defined as above); indeed, there exists a regular proof tree for `has_meth(fact,iter,[obj(fact,[]),int],`$t$`)` having the following shape (with the root at the bottom):

$\vdots$

`has_meth(fact,iter,[obj(fact,[]),int],`$t$`)`

$\vdots$

`has_meth(fact,iter,[obj(fact,[]),int],`$t$`)`

However not always a goal is derivable with a regular proof tree, even when all involved terms are regular. To see that, consider a slightly more elaborated version of meth `iter`:

```
iter2(i,l) {
  if(i<=0) return l;
  else return this.iter(i-1,new NEList(i,l));
}
```

Although `has_meth(fact,iter2,[obj(fact,[]),int,obj(eList,[])],`$t$`)` is derivable for the same type $t$ as defined above, such a goal has the following non regular proof tree:

$$\vdots$$
`has_meth(fact,iter2,[obj(fact,[]),int,`$t_n$`],`$t$`)`
$$\vdots$$
`has_meth(fact,iter2,[obj(fact,[]),int,`$t_1$`],`$t$`)`
$$\vdots$$
`has_meth(fact,iter2,[obj(fact,[]),int,`$t_0$`],`$t$`)`

The proof contains infinite distinct regular terms $t_0, \ldots, t_n, \ldots$ defined by

$t_0$`=obj(eList,[])`
$t_{n+1}$`=obj(neList,[el:int,next:`$t_n$`])`

corresponding to the type of the second argument of all the (recursive) invocations of `iter2`. Type $t_i$ represents all lists of integer numbers of length $i$, whereas $t$ represents all finite and circular lists of integer numbers, hence $t_i \leq t$ for all $i$. By contravariance of method arguments, atom `has_meth(fact,iter2,[obj(fact,[]),int,`$t$`],`$t$`)` *subsumes* `has_meth(fact,iter2,[obj(fact,[]),int,`$t_0$`],`$t$`)`, that is, if the former is derivable, then latter is derivable as well. Hence, by applying subsumption it is possible to build the following regular proof tree for the goal:

$$\vdots$$
`has_meth(fact,iter2,[obj(fact,[]),int,`$t$`],`$t$`)`
$$\vdots$$
`has_meth(fact,iter2,[obj(fact,[]),int,`$t$`],`$t$`)`
`has_meth(fact,iter2,[obj(fact,[]),int,`$t_0$`],`$t$`)`

Subtyping corresponds to inclusion between type interpretations, which are sets of values defined coinductively [5, 6]. In this paper we provide a sound, but not

complete, definition of subtyping coinductively defined by the rules below. Completeness issues are out of the scope of this paper [6].

$$\text{(int)}\frac{}{int \le int} \qquad \text{(bool)}\frac{}{bool \le bool} \qquad \text{(tuple)}\frac{\forall\, i = 1..n \quad t_i \le t'_i}{[t_1, \ldots, t_n] \le [t'_1, \ldots, t'_n]}$$

$$\text{(obj)}\frac{t_1 \le t_2}{obj(c, t_1) \le obj(c, t_2)} \qquad \text{(rec)}\frac{t_1 \le t'_1, \ldots, t_n \le t'_n}{[f_1{:}t_1, \ldots, f_n{:}t_n, g_1{:}u_1, \ldots, g_k{:}u_k] \le [f_1{:}t'_1, \ldots, f_n{:}t'_n]}$$

$$\text{(}\vee\text{R1)}\frac{t \le t_1}{t \le t_1 \vee t_2} \qquad \text{(}\vee\text{R2)}\frac{t \le t_2}{t \le t_1 \vee t_2} \qquad \text{(}\vee\text{L)}\frac{t_1 \le t \quad t_2 \le t}{t_1 \vee t_2 \le t}$$

$$\text{(distr)}\frac{\begin{array}{c} obj(c, [f_1{:}t_1, \ldots, f_n{:}t_n, f{:}u_1, g_1{:}t'_1, \ldots, g_k{:}t'_k]) \le t \\ obj(c, [f_1{:}t_1, \ldots, f_n{:}t_n, f{:}u_2, g_1{:}t'_1, \ldots, g_k{:}t'_k]) \le t \end{array}}{obj(c, [f_1{:}t_1, \ldots, f_n{:}t_n, f{:}u_1 \vee u_2, g_1{:}t'_1, \ldots, g_k{:}t'_k]) \le t}$$

Subtyping between object types (obj) holds only between instances of the same class (see below for further explanations), whereas (rec) defines the standard width and depth subtyping between immutable records. As we will see in Section 4, depth subtyping is unsound in the presence of mutable fields, therefore another rule is required for updatable records.

Rules ($\vee$R1), ($\vee$R2) and ($\vee$L) are standard. Rule (distr) ensures that object types "distributes over" union.

To avoid unsound subtyping, all derivations for $\le$ are required to be *contractive* [4].

**Definition 2.13.** A derivation for $t_1 \le t_2$ is contractive iff it contains no subderivations built only with subtyping rules ($\vee$R), and ($\vee$L).

The judgment $t_1 \le t_2$ is derivable iff there is a contractive derivation for it.

The problem with rules ($\vee$R), and ($\vee$L) is that they "consume" only a part of the term on the righthand side, hence it is possible to build unsound non contractive proof trees. For instance, by only applying rule ($\vee$L), it would be possible to build an infinite proof tree for $bool \le t_{int}$, where $t_{int} = t_{int} \vee int$, which is unsound since $t_{int}$ is equivalent to $int$ (intuitively, $t_{int}$ is an infinite union of $int$). However, according to Definition 2.13, such a proof tree is not contractive.

Once $\le$ is defined, one has to define subsumption, that is, how each predicate behaves w.r.t. subtyping. For instance, the predicate *invoke* is invariant w.r.t. its first and second argument, contravariant w.r.t. its third, and covariant w.r.t. its fourth; this is specified by the variance annotation $==\geq\leq$, meaning that the ground atom $invoke(t_1, m, t_2, t_3)$ subsumes the ground atom $invoke(t'_1, m', t'_2, t'_3)$ iff $t_1 = t'_1$, $m = m'$, $t_2 \ge t'_2$, and $t_3 \le t'_3$.

We extend the notion of ranked alphabet for predicate symbols to include variance annotations: $(p, (\alpha_1, \ldots, \alpha_n)) \in \Sigma_{p,\le}$ means that predicate $p$ has variance annotation $\alpha_1, \ldots, \alpha_n$, and, hence, arity $n$. We also write more succinctly $p_{\alpha_1,\ldots,\alpha_n}$.

**Definition 2.14.** If $p_{\alpha_1,\ldots,\alpha_n}$, then the ground atom $p(t_1, \ldots, t_n)$ *subsumes* the ground atom $p(t'_1, \ldots, t'_n)$ iff for all $i = 1..n$ the relation $t_i\, \alpha_i\, t'_i$ holds, where $t \ge t'$ holds iff $t' \le t$ holds, and $t = t'$ holds iff $t$ and $t'$ are syntactically equal.

With subtyping we switch from coinductive logic programming to coinductive constraint logic programming; however, the subtyping constraint is not introduced explicitly in the clause bodies, but rather implicitly with variance annotations, which must be provided for all predicates. Variance annotations allow more compact clauses, and a more convenient definition of the operational semantics of Horn formulas [3].

We can now extend the definition of coinductive Herbrand model with subtyping constraints.

**Definition 2.15.** The *immediate consequence operator* $T_{Hf,\leq}$ associated with formula $Hf$ is the endofunction over $\mathcal{P}(\mathcal{HB}(\Sigma_o, \Sigma_{p,\leq}))$ defined as follows:

$$T_{Hf,\leq}(S) = \{A' \mid \begin{array}{l} A \leftarrow B \text{ is a ground instance of a clause of } Hf \\ A \text{ subsumes } A' \text{ and } B \in S\}. \end{array}$$

A Herbrand model of $Hf$ with subtyping constraints is a subset of $\mathcal{HB}(\Sigma_o, \Sigma_{p,\leq})$ which is a fixed-point of $T_{Hf,\leq}$. The coinductive Herbrand model of $Hf$ with subtyping constraints is the greatest fixed-point of $T_{Hf,\leq}$, and is denoted by $M^{\leq}_{Hf}$.

Finally, we explain why $obj(c_1, [\ldots])$ is not a subtype of $obj(c_2, [\ldots])$ when $c_1$ is a proper subclass of $c_2$. Indeed, to allow more precise type inference we decouple inheritance from subtyping (see the seminal paper by Cook and Canning [11]) and do not impose any overriding rule. Let us consider the following classical example, where class `ColPoint` inherits method `move` from `Point` and overrides method `equals`.

```
class Point {
    x,y;
    move(dx,dy) {x+=dx; y+=dy;}
    equals(p) {return p.x==x && p.y==y;}
}
class ColPoint extends Point {
    c;
    equals(cp) {return super.equals(cp) && cp.c==c;}
}
```

The type $obj(point, [x{:}int, y{:}int]) \vee obj(colPoint, [x{:}int, y{:}int])$ can be inferred for `z` in the expression `z.move(1,2)` (note that for method `move` no information on field `c` is required for `colPoint` instances), hence, the inherited method can be effectively used by class `ColPoint` as well. Such an inference is allowed by the following two clauses,[4] shared by all translated programs (see Figure 5), which specify the behavior of `invoke` w.r.t. union types, and of `has_meth` for inherited methods.

```
invoke(T1∨T2,M,A,RT1∨RT2) ← invoke(T1,M,A,RT1),invoke(T2,M,A,RT2).
has_meth(C,M,A,R) ← extends(C,P),has_meth(P,M,A,R),¬dec_meth(C,M).
```

[4]We use negation for brevity, see further comments in Section 5.

If we consider method `equals`, then the type which can be inferred for `z` in the expression `z.equals(new Point())` is $obj(point, [x{:}int, y{:}int])$, since instances of class `Point` do not have a `c` field; therefore $obj(colPoint, [x{:}int, y{:}int])$ is not a subtype of $obj(point, [x{:}int, y{:}int])$, as correctly captured by subtyping rule (obj).

## 3. SSA intermediate forms for typing imperative features

We start this section with a simple example to show how a source program can be transformed into an SSA intermediate form, and why this transformation enhances static type analysis.

Consider the following class declarations written in our simple untyped language[5] and defining simple geometrical shapes.

```
class Circle {
    radius;
    Circle(r) {
        super();
        this.radius=r;
    }
    getRadius() {
        return this.radius;
    }
    area() {
        return 3.141592*this.radius*this.radius;
    }
}
class Square {
    side;
    Square(s) {
        super();
        side=s;
    }
    getSide() {
        return this.side;
    }
    area() {
        return this.side*this.side;
    }
}
```

[5]In the following we assume that string and floating-point data types are supported, as well as methods for printing values.

For simplicity, both classes extend the predefined root class `Object`, instead of introducing a superclass `Shape` to factor out all common features, as it would be customary in practice.

The following code fragment, contained in another class called `ShapeReader`, creates a shape which is read from an input stream `reader` and invokes on it some methods. The `reader` object must be an instance of a class which provides a method `next()` returning the next string read from the stream. The methods `readCircle()` and `readSquare()` of class `ShapeReader` (whose definitions have been omitted since they are not important for the example) read a double precision floating-point number from the input, create a new instance of `Circle` and `Square`, respectively, and return it.

```
    read(reader) {
2       st=reader.next();
        if(st.equals("circle")) {
4           sh=this.readCircle();
            this.print("A␣circle␣with␣radius␣");
6           this.print(sh.getRadius());
        }
8       else if(st.equals("square")) {
            sh=this.readSquare();
10          this.print("A␣square␣with␣side␣");
            this.print(sh.getSide());
12      }
        else return; // an exception should be thrown
14      this.print("Area␣=␣");
        this.print(sh.area());
16  }
```

It is clear from the code that **sh** on line 6 and line 11 will always contain `Circle` and `Square` instances, respectively, hence, both **sh**`.getRadius()` and **sh**`.getSide()` are type safe. On the other hand, **sh** on line 15 can either contain a circle or a square, depending on the input, therefore **sh**`.getRadius()` and **sh**`.getSide()` would not be type safe in this context, whereas method `area` can be safely invoked, since the method is defined in both classes.

This means that the most accurate type that can be inferred for **sh** is the following: $obj(circle, [radius{:}double]) \vee obj(square, [side{:}double])$. As a consequence, method **read** cannot be typed, since the type of **sh** is not compatible with the invocation of methods **sh**`.getRadius()` and **sh**`.getSide()`. However, method **read** can be typed if the types associated with the occurrences of **sh** are allowed to be different. This can be achieved by performing type inference on the SSA intermediate form of **read**, rather than on its source code. The flow graph corresponding to the SSA form of **read** is shown in Figure 1.

A program is in SSA form if the value of each variable is determined by exactly one assignment statement in the program [15]. To obtain this property, the transformation from source to SSA form performs a suitable renaming of variables to

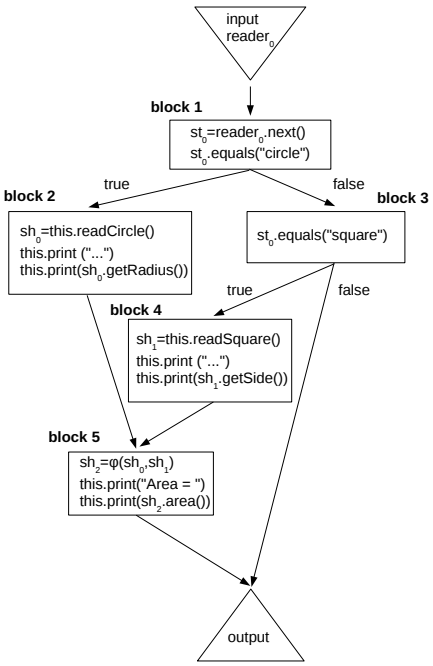


FIGURE 1. Control flow graph corresponding to the body of method **read**

keep track of the possibly different versions of the same variable. Conventionally, such renaming preserves the names of the original variable and introduces a different subscript for each version. For instance, in Figure 1 there are three different versions for variable **sh**: $\mathbf{sh}_0$, $\mathbf{sh}_1$, and $\mathbf{sh}_2$, defined respectively in block 2, 4, and 5.

To transform a program into SSA form, pseudo-functions, which are conventionally called $\varphi$ functions, have to be inserted when multiple definitions converge in merge points. Consider block 5 in Figure 1, which can be reached either from block 2 or 4: The value of **sh** in **print(sh.area())** is that of either $\mathbf{sh}_0$ or $\mathbf{sh}_1$, therefore a new version $\mathbf{sh}_2$ must be introduced. The definition $\varphi(\mathbf{sh}_0,\mathbf{sh}_1)$ of $\mathbf{sh}_2$ keeps track of the fact that the value of $\mathbf{sh}_2$ is determined either by $\mathbf{sh}_0$ or $\mathbf{sh}_1$.

The transformation of a source program into its SSA form is standard [15], and there exists a quite efficient algorithm to perform it [16], therefore for simplicity our type system is directly defined on programs in SSA form. Expressing SSA forms with flow graphs enhances readability of programs, but for formalizing the type system it is better to adopt a textual language. For instance, in our language the SSA form of method **read** is the following:

```
read(reader0) {
  b1:{st0 = reader0.next();
      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 out;
     }
  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;
     }
  out:{return;}
}
```

The body of a method in SSA form is a sequence of uniquely labeled blocks; each block ends with either a conditional or unconditional `jump`, or a `return`. For simplifying the encoding, we require that only the last block[6] contains the `return` statement.

Let us show how the intermediate form of method **read** can be encoded[7] in a Horn clause:

```
meth(shapeReader,read,[This,Reader0],void) ←
   type_comp(This,shapeReader),
   invoke(Reader0,next,[],St0),
   invoke(St0,equals,[string],bool),
```

[6]This can be always obtained with simple transformations; in case of multiple returned values, several versions of a fresh variable containg the returned values and a $\varphi$ function is introduced.

[7]To save space, we have optimized the encoding by removing atoms which are always clearly satisfied, as `invoke(This,print,[string],_)`, and used the anonymous variable `_` for unused returned values.

```
invoke(This,readCircle,[],Sh0),
invoke(Sh0,getRadius,[],X0),
invoke(This,print,[X0],_),
invoke(This,readSquare,[],Sh1),
invoke(Sh1,getSide,[],X1),
invoke(This,print,[X1],_),
var_upd(Sh2,Sh0∨Sh1),
invoke(Sh2,area,[],X2),
invoke(This,print,[X2],_).
```

The predicate `var_upd/`$\leq\geq$ defines type safe assignments to local variables and is trivially specified by the following fact: `var_upd(X,X).` The first argument is the type of the destination, the second argument is the type of the source of the assignment. An assignment is type safe when the type of the source is a subtype of the type of the destination; the predicate is covariant in the first argument, and contravariant in the second one, therefore `var_upd(`$t_1$`,`$t_2$`)` succeeds iff there exists $t$ s.t. $t \leq t_1$ and $t_2 \leq t$ (since `var_upd(`$t$`,`$t$`)` holds by definition), that is, iff $t_2 \leq t_1$.

The atom `meth(shapeReader,read,[`*this*`,`*reader*`],void)` succeeds, by instantiating the clause above with the following substitution

```
St0=string
Sh0=obj(circle,[radius:double])
Sh1=obj(square,[side:double])
X0=X1=X2=double
Sh2=obj(circle,[radius:double])∨obj(square,[side:double])
```

if we assume that the object *this* is an instance of `ShapeReader` or of any of its subclasses (`type_comp(`*this*`,shapeReader)`), that object *reader* has a method `next` which has no arguments and returns a string, that the class `ShapeReader` has the two methods `readCircle` and `readSquare` taking no arguments and returning an instance of class `Circle` and `Square`, respectively, that objects of type string has method `equals` taking a string as argument, and returning a boolean value, and that any object has the predefined method `print`, which can be invoked on any argument and returns the `void` value.

As a final remark, note that the encoding of method `read` is control flow insensitive; however, the control flow information contained in the SSA intermediate form could be exploited to elaborate more sophisticated forms of encoding, and, thus, to define more precise type systems.

## 4. Field assignments

In the previous section we have shown how it is possible to perform type analysis of imperative constructs such as assignment to local variables, conditional execution and iteration.

In this section we deal with the problem of typing field assignments. Ensuring the type integrity of objects whose address (identity) is stored on the heap, rather than on the stack, is more difficult. Indeed, while the use of the SSA form

allows a more precise analysis on the content of local variables (and, therefore, of parameters), the same approach cannot be applied to object fields (that is, variables allocated on the heap). In this case a more conservative, even though precise enough in most cases, type analysis is performed, by exploiting coinduction, union types, and subtyping.

### 4.1. Subtyping

It is well known that depth record subtyping is unsound when fields are mutable. To allow more expressiveness, record types are extended by annotating fields with an access modifier ranging over the following three values: **r** (read-only field), **w** (write-only field), **rw** (read-write field). For instance, $[f_1^{\mathbf{r}}{:}t_1, f_2^{\mathbf{w}}{:}t_2, f_3^{\mathbf{rw}}{:}t_3]$ is the type of records with field $f_1$ of type $t_1$ which can be selected but not updated, $f_2$ of type $t_2$ which can be updated but not selected, $f_3$ of type $t_3$ which can be both updated and selected. Consequently, rule (rec) is generalized as follows.

$$(\text{rec})\frac{(a_1,t_1) \leq (a_1',t_1') \ \ldots \ (a_n,t_n) \leq (a_n',t_n')}{[f_1^{a_1}{:}t_1,\ldots,f_n^{a_n}{:}t_n, g_1^{b_1}{:}u_1,\ldots,g_k^{b_k}{:}u_k] \leq [f_1^{a_1'}{:}t_1',\ldots,f_n^{a_n'}{:}t_n']}$$

Subtyping between pairs of the form $(a,t)$ is defined by the following rules:

$$(\text{r})\frac{a \leq \mathbf{r} \qquad t_1 \leq t_2}{(a,t_1) \leq (\mathbf{r},t_2)} \qquad (\text{w})\frac{a \leq \mathbf{w} \qquad t_2 \leq t_1}{(a,t_1) \leq (\mathbf{w},t_2)} \qquad (\text{rw})\frac{t_1 \cong t_2}{(\mathbf{rw},t_1) \leq (\mathbf{rw},t_2)}$$

The subtyping relation $\leq$ on access modifiers is defined as follows:

$$a_1 \leq a_2 \text{ iff } a_1 = a_2 \text{ or } a_1 = \mathbf{rw}.$$

The definition corresponds to the intuition that the relation is reflexive and that a field which is both readable and writeable, is also readable or writeable.

Rules (r), (w) and (rw) simply state that depth record subtyping is covariant w.r.t. read-only fields, contravariant w.r.t write-only fields, and weakly invariant w.r.t. read-write fields ($t \cong t'$ iff $t \leq t'$ and $t' \leq t$). Note that (rec) allows width subtyping as well, with no restrictions on field access modifiers.

### 4.2. Encoding of field assignment

Consider the following simple code fragment example:

```
x=new NEList(1,new EList());
x.next=x;
```

After the assignment to field `next`, the variable `x` contains a recursive object corresponding to an infinite list. The above code can be encoded into the following sequence of atoms:

```
new(eList,[],T),new(neList,[int,T],X),field_upd(X,next,X).
```

The predicate `field_upd/`$_{\cong=\geq}$ defines type safe assignments to fields: the first argument is the type of the object whose field is modified, the second is the name of the field, and the third is the type of the value which is assigned to the field. As happens for `invoke`, `field_upd` is invariant w.r.t. its first and second argument, whereas is contravariant w.r.t. the third argument (as `assign`).

The predicate `field_upd` can be easily defined on top of the predicate `rec_upd/`$_{\geq=\geq}$ specifying type safe record updates:

```
field_upd(obj(C,R),F,T) ← has_field(C,F),rec_upd(R,F,T).
field_upd(T1∨T2,F,T) ← field_upd(T1,F,T),field_upd(T2,F,T).
```

Assigning values of type `T` to a field `F` of an object of type `obj(C,R)` is type safe if class `C` has field `F`, and in the record type `R` the field `F` can be safely updated with a value of type `T` (first clause for `field_upd`). Similarly to method invocations and field accesses, a field assignment on an object of type `T1`$\vee$`T2` is correct if the same assignment is correct for both types `T1` and `T2` (second clause for `field_upd`).

Predicate `rec_upd` is defined by the following fact: `rec_upd([F^w:T],F,T).` If a record has a writable field `F` of type `T`, then `F` can be safely updated with any value of type `T`. Note that, since `rec_upd` is contravariant w.r.t. its first and third arguments, the update is type safe for any record whose type is a subtype of `[F^w:T]`, and for any assigned value whose type is a subtype of `T`; for instance, by rules (rec) and (rw), we can deduce that updates are correct also for records with more fields where `F` has type `T2`, with `T` $\leq$ `T2`, and is both readable and writeable (`F^rw`).

Given the clauses defining `field_upd` and `rec_upd`, we can now verify that the goal `new(eList,[],T),new(neList,[int,T],X),field_upd(X,next,X)`, which encodes the example at the begining of Section 4.2, succeeds for the substitution `T=`$t_T$ `X=`$t_X$, where $t_T$ and $t_X$ are defined as follows:

$t_T$ = `obj(eList,[])`$\vee t_X$
$t_X$ = `obj(neList,[el^rw:int,next^rw:`$t_T$`])`

Indeed, `new(eList,[],`$t_T$`)` holds by subsumption, since `new(eList,[],obj(eList,[]))` succeeds and `obj(eList,[])`$\leq t_T$.

The atom `new(neList,[int,`$t_T$`],obj(neList,[el^rw:int,next^rw:`$t_T$`]))` clearly holds.

Finally, `field_upd(`$t_X$`,next,`$t_X$`)` succeeds because `rec_upd([next^rw:`$t_T$`],next,`$t_X$`)` holds by subsumption, since `rec_upd([next^w:`$t_T$`],next,`$t_T$`)` succeeds, and $t_X \leq t_T$ and `[next^rw:`$t_T$`]`$\leq$`[next^w:`$t_T$`]` hold.

Note that, by slightly generalizing the definition of $t_T$ and $t_X$ as follows:

$t_T$ = `obj(eList,[])`$\vee t_X \vee t$
$t_X$ = `obj(neList,[el^rw:int,next^rw:`$t_T$`])`

we obtain an infinite set of substitutions (each obtained from a different type $t$) which all satisfy our goal.

The types $t_X$ obtained from different definitions of $t$ (and, hence, of $t_T$) are not comparable in general, since subtyping is invariant w.r.t. the field `next` with **rw** access.

For instance, let us consider $t_X^1$ and $t_X^2$ obtained from $t_T^1$ and $t_T^2$ defined respectively as follows:

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \overline{cd}^{\,n}\ e \quad (e \text{ ground}) \\
\mathit{cd} & ::= & \texttt{class}\ c_1\ \texttt{extends}\ c_2\ \{\ \overline{f}^{\,n}\ \mathit{cn}\ \overline{\mathit{md}}^{\,m}\ \} \quad (c_1 \neq \mathit{Object}) \\
\mathit{cn} & ::= & c(\overline{x_0}^{\,n})\ \{l{:}\texttt{this}.c_{\texttt{super}}(\overline{e}^{\,m}); \overline{\texttt{this}.f = e';}^{\,k}\ \texttt{return this}\} \\
\mathit{md} & ::= & m(\overline{x_0}^{\,n})\ \{\overline{b}^{\,n}\} \\
b & ::= & l{:}e \\
\mathit{name} & ::= & \texttt{this} \mid x_v \\
\nu & ::= & \texttt{false} \mid \texttt{true} \mid \texttt{null} \\
e & ::= & \varphi(x_{v_1}, \ldots, x_{v_n}) \mid \texttt{new}\ c(\overline{e}^{\,n}) \mid x_v \mid e.f \mid e_0.m(\overline{e}^{\,n}) \mid \texttt{this} \mid \nu \mid e_1; e_2 \\
 & & x_v = e \mid e_1.f = e_2 \mid \texttt{jump}\ l \mid \texttt{if}\ (e)\ \texttt{jump}\ l_1\ \texttt{else jump}\ l_2 \mid \texttt{return}\ \mathit{name}
\end{array}
$$

*Syntactic assumptions*: inheritance is not cyclic, constructor initialization expressions do not contain `this`, $c_{\texttt{super}}$ indicates the name of the superclass, method bodies are in correct SSA form (where the last block consists of a `return`, and no other block uses `return`), method and class names are disjoint, no name conflicts in class, field, method and parameter declarations.

FIGURE 2. SSA intermediate language

$t_T^1 = \texttt{obj(eList,[])} \vee t_X$
$t_T^2 = \texttt{obj(eList,[])} \vee t_X \vee \texttt{obj(neList,[el\^{}rw:int,next\^{}rw:obj(eList,[])])}$

The type $t_X^1$ assigns a more precise type to `x.next`, whereas $t_X^2$ accepts as type safe more updates of `x.next`.

## 5. Formalization

In this section we formalize abstract compilation of the language used in the examples of the previous sections. The syntax of the language is defined in Figure 2.

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, a single[8] constructor, and its methods.[9]

For simplicity, we treat constructors like methods named as their corresponding classes; for this reason, no (real) method can be named like any class used in the program. Constructor bodies are simple: they contain a single block, whose label is immaterial. This block first invokes the constructor of the direct superclass $c_{\texttt{super}}$, to initialize the inherited fields. Then, it initializes all fields declared in the class (for simplicity in the same order as they have been declared). Finally, the initialized object is returned. Constructor bodies are so simple that in fact there is no difference between their source and SSA form. However, for uniformity, all parameters are annotated with version 0.

Method bodies are in SSA form, that is, a sequence of uniquely labeled blocks where each variable is determined by exactly one assignment. Parameters can

[8]For simplicity.

[9]Recall that the language does not allow type annotations.

be modified like the other local variables. Each block contains a sequence of expressions (for simplicity we do not distinguish expressions and statements) which is always terminated by a jump, which can be either a return from the method, or a conditional or unconditional jump to another block of the method.

Expressions include assignments to variables and to object fields, object creations, method invocations, variables, field selections, the keyword `this` denoting the target object, conditional and unconditional jumps, sequences, and the boolean literals `false`, `true` and `null`.

Note that the syntax definition is more liberal than that corresponding to programs in SSA form generated by a front-end compiler; for instance, in a correct program in SSA form jumps can only be the last expression in a block. Hence in Figure 2 we assume that method bodies are in correct SSA form. The syntax is more liberal because we decided not to distinguish expressions and statements; however, such a choice allows a lighter technical treatment, at the negligible cost of adding some reasonable syntactic assumptions.

Besides those standard expressions, there is also applications of $\varphi$ functions to different versions of the same variable. We have omitted numerical literals and the usual logic-arithmetic operators, since their translation is straightforward.

The translation of programs, class, field, constructor, and method declarations is defined in Figure 3.

To avoid a too cumbersome definition, we assume that in the translation, the keyword `this`, variables, class, field and method names are mapped to themselves, even though, to be more precise, appropriate injections should be used [4].

The translation of a program is a pair consisting of a Horn formula $Hf^d, \overline{Hf}^n$ and a conjunction of atoms $B$, where $Hf^d$ is the set of shared clauses generated by any compilation (see Figure 5 below), $\overline{Hf}^n$ are the clauses generated from all class declarations, and $B$ is the translation of the main expression of the program. The generated program is type safe if there exists a substitution satisfying all atoms in $B$ w.r.t. the coinductive Herbrand model of $Hf^d, \overline{Hf}^n$ (see the claim 6.7 at the end of this section).

The translation of a class $c_1$ generates all clauses obtained from all field, constructor, and method declarations in $c_1$; furthermore, two facts are generated, to keep track of the name of the class (predicate *class*) and of its direct superclass (predicate *extends*). The keyword **in** introduces all parameters needed by the translations of the syntactic categories. To correctly translate field and method declarations, the class where the declarations are contained is needed; for translating constructor declarations we need to know the sequence of all fields declared in the class of the constructor.

The translation of a constructor declaration generates just one clause for the predicate *new*. The body of the clause contains all the atoms generated from the compilation of the expressions in the body of the constructor, plus two atoms which check that the invocation of the constructor of the direct superclass is type safe. The returned type is an object instance of the class $c$ of the constructor, where the record type of fields is obtained by appending the type $\overline{f{:}t'}^k$ of the fields

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

$$\text{(class)}\ \frac{\forall\, i = 1..n\ f_i\ \mathbf{in}\ c_1 \rightsquigarrow Cl_i \quad cn\ \mathbf{in}\ \overline{f}^n \rightsquigarrow Cl \quad \forall\, j = 1..m\ md_j\ \mathbf{in}\ c_1 \rightsquigarrow Hf_j}{\begin{array}{l}\texttt{class}\ c_1\ \texttt{extends}\ c_2\ \{\ \overline{f}^n\ \ cn\ \overline{md}^m\ \} \rightsquigarrow \\ \quad \overline{Cl}^n \cup Cl \cup \overline{Hf}^m \cup \left\{ \begin{array}{l} class(c_1) \leftarrow true. \\ extends(c_1, c_2) \leftarrow true. \end{array} \right\}\end{array}}$$

$$\text{(field)}\ \frac{}{f\ \mathbf{in}\ c \rightsquigarrow dec\_field(c, f) \leftarrow true.}$$

$$\text{(constr)}\ \frac{\forall\, i = 1..m\ e_i \rightsquigarrow (t_i \mid B_i) \quad \forall\, j = 1..k\ e'_j \rightsquigarrow (t'_j \mid B'_j)}{\begin{array}{l} c(\overline{x_0}^n)\ \{l{:}\texttt{this}.c_{\texttt{super}}(\overline{e}^m); \overline{f = e'}^k\ \texttt{return this}\}\ \mathbf{in}\ \overline{f}^k \rightsquigarrow \\ \quad new(c, [\overline{x_0}^n], obj(c, [\overline{f{:}t'}^k \mid R])) \leftarrow \ \ \overline{B}^m, new(c_{\texttt{super}}, [\overline{t}^m], obj(P, R)), \overline{B'}^k. \end{array}}$$

$$\text{(meth)}\ \frac{\overline{b}^n \rightsquigarrow (t \mid B)}{\begin{array}{l} m(\overline{x_0}^n)\{\overline{b}^n\}\ \mathbf{in}\ c \rightsquigarrow \\ \quad dec\_meth(c, m) \leftarrow true. \\ \quad has\_meth(c, m, [This, \overline{x_0}^n], t) \leftarrow \ \ type\_comp(This, c), B. \end{array}}$$

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

FIGURE 3. Translation of programs, class, field, constructor, and method declarations.

declared in $c$ to the record type $R$ of the inherited fields returned by the call to the superclass constructor.

Method declarations generate two clauses, one for predicate *dec_meth* and the other for *has_meth*. The first clause is a fact which specifies that method $m$ is declared in class $c$, whereas the second clause defines the type of the method: its body contains the atoms generated from the body of the method, plus the atom which requires `this` to have type $c'$, where $c'$ is a subtype of the class $c$ where the method is declared.

A method body is a sequence of blocks in SSA form which always ends with a `return` block (which is the only block in the body of a method containing a `return` statement). The compilation of a method body consists of the returned type, that is, the type of the returned variable $x_v$, and the conjunction of all the atoms generated by the blocks of the method body.

Figure 4 defines the translation of blocks, statements and expressions.

A block corresponds to the translation of its expression (which actually can be more than one, since the syntax admits sequence expressions) where the returned type is discarded. As already noted, the presented compilation scheme is control flow insensitive; however, more precise forms of abstract compilation could be devised to exploit the control flow information of the SSA form.

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

$$\text{(jmp)}\frac{}{\texttt{jump}\ l \rightsquigarrow (\bot \mid \mathit{true})} \qquad \text{(c-jmp)}\frac{e \rightsquigarrow (t \mid B)}{\texttt{if}\ (e)\ \texttt{jump}\ l_1\ \texttt{else jump}\ l_2 \rightsquigarrow (\bot \mid B, \mathit{type\_comp}(t, \mathit{bool}))}$$

$$\text{(var-upd)}\frac{e \rightsquigarrow (t \mid B)}{x_v = e \rightsquigarrow (t \mid B, \mathit{var\_upd}(x_v, t))}$$

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

$$\text{(phi)}\frac{}{\varphi(x_{v_1}, \ldots, x_{v_n}) \rightsquigarrow (x_{v_1} \vee \ldots \vee x_{v_n} \mid \mathit{true})}$$

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

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

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

$$\text{(name)}\frac{}{\mathit{name} \rightsquigarrow (\mathit{name} \mid \mathit{true})} \qquad \text{(bool)}\frac{\nu \in \{\texttt{true}, \texttt{false}\}}{\nu \rightsquigarrow (\mathit{bool} \mid \mathit{true})} \qquad \text{(null)}\frac{}{\texttt{null} \rightsquigarrow (\bot \mid \mathit{true})}$$

FIGURE 4. Translation of blocks, statements and expressions.

The translation of a sequence expression $e_1; e_2$ collects the atoms generated from the translation of both $e_1$ and $e_2$, keeps the type of $e_2$ and discard that of $e_1$.

The translation of an unconditional jump generates no atoms, whereas a conditional jump is translated in the conjunction of atoms generated from the condition $e$, plus the atom requiring the type $t$ of $e$ to be compatible[10] with the type *bool*. The type of a jump is void, that is, the bottom type $\bot$ (which is the type $t$ s.t. $t = t \vee t$ [5]).

The translation of assignments to local variables and fields yields the conjunction of the atoms generated from the corresponding sub-expressions, plus the atom specific of the statement (built on predicates *var_upd* and *field_upd*, respectively). The returned type is the type of the righthand side expression.

For object creation, field selection, and method invocation, the generated type is a fresh logical variable which is instantiated with the type returned by the specific predicates (*new*, *field_acc*, and *invoke*, respectively). The returned constraints are the conjunction of the atoms generated from the corresponding sub-expressions, plus the atom specific of the statement.

Application of $\varphi$ functions are translated in the corresponding union type, and in the empty conjunction.

[10]For instance, $t$ could be $\mathit{bool} \vee \mathit{bool}$.

$class(object) \leftarrow true.$
$subclass(X, X) \leftarrow class(X).$
$subclass(X, object) \leftarrow class(X).$
$subclass(X, Y) \leftarrow extends(X, Z), subclass(Z, Y).$
$type\_comp(bool, bool) \leftarrow true.$
$type\_comp(obj(C1, X), C2) \leftarrow subclass(C1, C2).$
$type\_comp(T1 \vee T2, C) \leftarrow type\_comp(T1, C), type\_comp(T2, C).$
$field\_acc(obj(C, R), F, T) \leftarrow has\_field(C, F), rec\_acc(R, F, T).$
$field\_acc(T1 \vee T2, F, FT1 \vee FT2) \leftarrow field\_acc(T1, F, FT1), field\_acc(T2, F, FT2).$
$rec\_acc([F\hat{}r{:}T], F, T) \leftarrow true.$
$invoke(obj(C, R), M, A, RT) \leftarrow has\_meth(C, M, [obj(C, R)|A], RT).$
$invoke(T1 \vee T2, M, A, RT1 \vee RT2) \leftarrow invoke(T1, M, A, RT1), invoke(T2, M, A, RT2).$
$new(object, [\,], obj(object, [\,])) \leftarrow true.$
$has\_field(C, F) \leftarrow dec\_field(C, F).$
$has\_field(C, F) \leftarrow extends(C, P), has\_field(P, F).$
$has\_meth(C, M, A, R) \leftarrow extends(C, P), has\_meth(P, M, A, R), \neg dec\_meth(C, M).$
$var\_upd(T, T) \leftarrow true.$
$field\_upd(obj(C, R), F, T) \leftarrow has\_field(C, F), rec\_upd(R, F, T).$
$field\_upd(T1 \vee T2, F, T) \leftarrow field\_upd(T1, F, T), field\_upd(T2, F, T).$
$rec\_upd([F\hat{}w : T], F, T) \leftarrow true.$

FIGURE 5. Shared clauses $Hf^d$.

| | | | | | |
|---|---|---|---|---|---|
| $class_{=}$ | $extends_{==}$ | $subclass_{==}$ | $type\_comp_{\geq=}$ | $field\_acc_{==\leq}$ | $rec\_acc_{\geq=\leq}$ |
| $invoke_{==\geq\leq}$ | $new_{=\geq\leq}$ | $dec\_field_{==}$ | $has\_field_{==}$ | $dec\_meth_{==}$ | $has\_meth_{==\geq\leq}$ |
| $var\_upd_{\leq\geq}$ | $field\_upd_{==\geq}$ | $rec\_upd_{\geq=\geq}$ | | | |

FIGURE 6. Subsumption annotations for all predicates.

The translation of the remaining expressions is straightforward.

Figure 5 contains shared clauses generated by any compilation.

The use of negation ($\neg dec\_meth$) allows compilation to be fully compositional, that is, to be independent from any particular context. The predicate *dec_meth* is simply defined by a set of ground facts, hence its coinductive and inductive semantics always trivially coincide; as a consequence, our prototype interpreter follows the standard "negation as failure" approach of inductive logic programming, instead of implementing the more complex semantics of negation for coinductive logic programming [23]. However, to simplify the formal treatment, in the soundness proof we consider only definite Horn clauses, hence the generated program is first tacitly transformed into an equivalent definite Horn formula, by replacing $\neg dec\_meth$ with the predicate *not_dec_meth*, easily defined by a collection of ground facts.

Subsumption annotations of all predicates can be found in Figure 6.

# 6. Soundness

In this section we first model the small-step semantics, then we sketch the soundness proof.

In order to describe execution, we introduce the notion of *runtime expression* by enriching the definition of values $\nu$ and expressions $e$; see Figure 7. We add object identifiers $o$ to values $\nu$, and *frame expressions* $\sigma_{\langle m,l\rangle}\{e\}$ to expressions $e$.

Frame expressions are used to model the execution of methods (and constructors, which we treat as special methods); $\sigma_{\langle m,l\rangle}\{e\}$ represents the execution of an expression $e$ w.r.t. a *stack frame* $\sigma$; the annotation $\langle m, l\rangle$ indicates that the stack frame $\sigma$ is for method $m$ and that the currently executing block (of $m$) is labelled $l$. Keeping track of which block is currently executing is needed in the subject-reduction proof.

Stack frames $\sigma$ map names to their corresponding values; they are represented by a list of associations $name \mapsto \nu$ where associations on the righthand side overrides associations on the left with the same name; therefore an assignment to a local variable (see rule (var-asn) of Figure 8) transforms the current stack frame by appending a new association to it. In this way it is possible to model the semantics of a $\varphi$ function application $\varphi(x_{v_1}, \ldots, x_{v_n})$ (see rule (phi)), by selecting in the stack frame the value of the variable between $x_{v_1}, \ldots, x_{v_n}$ which has been updated most recently. This is achieved by searching the rightmost association of one of them. The append operator is $\cdot$ (a single dot).

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 $\nu$.

Execution is modelled by the judgment $\mathcal{H}, e \rightsquigarrow \mathcal{H}', e'$, to be read: "runtime expression $e$, in a heap $\mathcal{H}$, is reduced in one step into runtime expression $e'$, producing the heap $\mathcal{H}'$". The reduction arrow, and all auxiliary functions, should be parameterized by the whole executing program, $\overline{cd}^n$, which is kept implicit.

The execution inside a stack frame, that is, the execution of the body of a method/constructor, is modelled by the judgement $\mathcal{H}, \sigma_{\langle m,l\rangle}, e \rightsquigarrow \mathcal{H}', \sigma'_{\langle m,l'\rangle}, e'$ to be read: "runtime expression $e$, in a stack frame $\sigma$ (executing inside the block labeled $l$, of method $m$) and heap $\mathcal{H}$, is reduced in one step into runtime expression $e'$, producing stack frame $\sigma'$ (executing inside the block labeled $l'$, of the same method $m$) and heap $\mathcal{H}'$".

Context $\mathcal{C}[\cdot]$ and $\mathcal{D}[\cdot]$ are similar; the only difference is that the former does not enter inside frame expressions. Together they allow us to elegantly express the fact that execution steps must occur inside the most nested frame, see rule (cxt-closure) in Figure 8, which contains all execution rules.

Variable and field assignments evaluate to their right values; rule (var-asn) models variable assignments, and has the side effect of appending a new association to the current stack frame $\sigma$, while rule (fld-asn) models field assignments; in that case, the object referenced by $o$ is retrieved from the heap, and its value updated.

Rule (meth-call), describing method invocations, is a bit more involved; first of all, the object referenced by $o$ is retrieved in order to find its class, $c$. Then,

using the two auxiliary functions[11] *firstBlock* and *params*, we obtain the first block of the method and its parameter names. The result of the evaluation is a frame expression, where the stack frame is built by assigning the actual parameters to the formal ones, and the reference $o$ to `this`. Finally, the resulting runtime expression is the body of the first block. Note that the newly created stack frame is annotated by the name of the called method, $m'$, and the label of its first block, $l'$.

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

Rule (phi), (x-acc) and (this) model the access to a (set of different versions of the same) local variable, a local variable, and the current object, respectively; all of them simply extract the resulting value from the stack frame $\sigma$.

Rule (new) models object creations; 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 `null` and the resulting expression corresponds to the invocation of the constructor of class $c$.

Rule (fld-acc) models field accesses; their 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 stack frame annotation. 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 labelled $l'$ and updating the stack frame annotation accordingly.

Rule (return) models the return to the caller, by destroying (popping), the current stack frame $\sigma_{\langle m,l\rangle}$ and substituting the whole frame expression with the value of the returned variable $x_v$.

Rule (main) is the one that allows to start the execution of a program, that is, a sequence of class declarations $\overline{cd}^n$ and a main expression $e$. The premise of the rule exploits the small-step reduction, starting from an empty heap $\epsilon_{\mathcal{H}}$ and an empty stack frame $\epsilon$, annotated by the pair $\langle\bot,\bot\rangle$, which corresponds to the fact that the main expression is not actually contained in any block/method.

The relations $\rightsquigarrow^*$ and $\rightsquigarrow^*_{max}$ denote the transitive and the maximal transitive closure, respectively, of $\rightsquigarrow$: $\mathcal{H}, e \rightsquigarrow^* \mathcal{H}', e'$ iff $\mathcal{H}, e \rightsquigarrow \mathcal{H}'', e''$ and $\mathcal{H}'', e'' \rightsquigarrow^* \mathcal{H}', e'$, and $\mathcal{H}, e \rightsquigarrow^*_{max} \mathcal{H}', e'$ iff $\mathcal{H}, e \rightsquigarrow^* \mathcal{H}', e'$ and there exist no $\mathcal{H}'',\ e''$ s.t. $\mathcal{H}', e' \rightsquigarrow \mathcal{H}'', e''$.

The proof of soundness of abstract compilation is rather complex and relies on the properties of progress and subject reduction. We provide only a sketch of the proof. The first part of the proof consists in defining[12] abstract compilation for runtime expressions w.r.t. a given heap and stack frame; there are two distinct forms of compilation: $\mathcal{H}, e \rightsquigarrow (t \,|\, B)$ for the evaluation judgment $\mathcal{H}, e \rightsquigarrow \mathcal{H}', e'$, and $\mathcal{H}, \sigma, e \rightsquigarrow (t \,|\, B)$ for the evaluation judgment $\mathcal{H}, \sigma, e \rightsquigarrow \mathcal{H}', \sigma', e'$. The less

[11] The trivial definition of the auxiliary functions have been omitted.

[12] We have omitted the formal definitions.

$$
\begin{array}{rcl}
e & ::= & \sigma_{\langle m,l\rangle}\{e\} \\
\nu & ::= & o \\
\sigma & ::= & \overline{name \mapsto \nu}^{n} \\
\mathcal{H} & ::= & \overline{o \mapsto \langle c, \overline{f \mapsto \nu}^{m}\rangle}^{n} \\
\mathcal{C}[\cdot] & ::= & [\cdot] \mid x_v = \mathcal{C}[\cdot] \mid \mathcal{C}[\cdot].f = e \mid \nu.f = \mathcal{C}[\cdot] \mid \mathcal{C}[\cdot].m(\overline{e}^{n}) \mid \nu_0.m(\overline{\nu}^{n}\mathcal{C}[\cdot]\overline{e}^{m}) \\
 & & \texttt{new}\ c(\overline{\nu}^{n}\mathcal{C}[\cdot]\overline{e}^{m}) \mid \mathcal{C}[\cdot].f \mid \mathcal{C}[\cdot]; e \mid \texttt{if}\ (\mathcal{C}[\cdot])\ \texttt{jump}\ l_1\ \texttt{else jump}\ l_2 \\
\mathcal{D}[\cdot] & ::= & [\cdot] \mid x_v = \mathcal{D}[\cdot] \mid \mathcal{D}[\cdot].f = e \mid \nu.f = \mathcal{D}[\cdot] \mid \mathcal{D}[\cdot].m(\overline{e}^{n}) \mid \nu_0.m(\overline{\nu}^{n}\mathcal{D}[\cdot]\overline{e}^{m}) \\
 & & \texttt{new}\ c(\overline{\nu}^{n}\mathcal{D}[\cdot]\overline{e}^{m}) \mid \mathcal{D}[\cdot].f \mid \mathcal{D}[\cdot]; e \mid \texttt{if}\ (\mathcal{D}[\cdot])\ \texttt{jump}\ l_1\ \texttt{else jump}\ l_2 \mid \sigma_{\langle m,l\rangle}\{\mathcal{D}[\cdot]\}
\end{array}
$$

$$
\sigma(\overline{name}^{n}) = \nu_k \Leftrightarrow \begin{cases} \sigma = \overline{name'_j \mapsto \nu_j}^{m} \\ \exists i \in \{1,\ldots,n\} : name'_k = name_i \\ \forall z : z > k \implies \forall i : name'_z \neq name_i \end{cases}
$$

FIGURE 7. Syntax of runtime expressions and $\sigma_{\langle m,l\rangle}$-lookup.

$$
\text{(ctx-closure)}\ \frac{\mathcal{H}, \sigma_{\langle m,l\rangle}, e \rightsquigarrow \mathcal{H}', \sigma'_{\langle m,l'\rangle}, e'}{\mathcal{H}, \mathcal{D}[\sigma_{\langle m,l\rangle}\{\mathcal{C}[e]\}] \rightsquigarrow \mathcal{H}', \mathcal{D}[\sigma'_{\langle m,l'\rangle}\{\mathcal{C}[e']\}]}
\qquad
\text{(var-asn)}\ \frac{}{\mathcal{H}, \sigma_{\langle m,l\rangle}, x_v = \nu \rightsquigarrow \mathcal{H}, \sigma_{\langle m,l\rangle} \cdot x_v \mapsto \nu, \nu}
$$

$$
\text{(fld-asn)}\ \frac{\begin{array}{c}\mathcal{H}(o) = \langle c, \overline{f \mapsto \nu}^{n}\rangle \\ f = f_j \\ \nu'_i = \begin{cases} \nu_i & i \neq j \\ \nu & i = j \end{cases}\end{array}}{\mathcal{H}, \sigma_{\langle m,l\rangle}, o.f = \nu \rightsquigarrow \mathcal{H}[\langle c, \overline{f \mapsto \nu'}^{n}\rangle/o], \sigma_{\langle m,l\rangle}, \nu}
\qquad
\text{(meth-call)}\ \frac{\begin{array}{c}\mathcal{H}(o) = \langle c, \_\rangle \\ \mathit{firstBlock}(c, m') = l' : e \\ \mathit{params}(c, m') = \overline{x_0}^{n}\end{array}}{\begin{array}{l}\mathcal{H}, \sigma_{\langle m,l\rangle}, o.m'(\overline{\nu_0}^{n}) \rightsquigarrow \\ \mathcal{H}, \sigma_{\langle m,l\rangle}, (\overline{x_0 \mapsto \nu_0}^{n} \cdot \texttt{this} \mapsto o)_{\langle m',l'\rangle}\{e\}\end{array}}
$$

$$
\text{(sequence)}\ \frac{}{\mathcal{H}, \sigma_{\langle m,l\rangle}, \nu; e \rightsquigarrow \mathcal{H}, \sigma_{\langle m,l\rangle}, e}
\qquad
\text{(phi)}\ \frac{}{\begin{array}{l}\mathcal{H}, \sigma_{\langle m,l\rangle}, \varphi(x_{v_1}, \ldots, x_{v_n}) \rightsquigarrow \\ \mathcal{H}, \sigma_{\langle m,l\rangle}, \sigma(x_{v_1}, \ldots, x_{v_n})\end{array}}
$$

$$
\text{(new)}\ \frac{\begin{array}{c}o \text{ fresh in } \mathcal{H} \\ \mathit{fieldNames}(c) = \overline{f}^{n}\end{array}}{\begin{array}{l}\mathcal{H}, \sigma_{\langle m,l\rangle}, \texttt{new}\ c(\overline{\nu}^{n}) \rightsquigarrow \\ \mathcal{H}[\langle c, \overline{f \mapsto \texttt{null}}^{n}\rangle/o], \sigma_{\langle m,l\rangle}, o.c(\overline{\nu}^{n})\end{array}}
\qquad
\text{(fld-acc)}\ \frac{\begin{array}{c}\mathcal{H}(o) = \langle c, \overline{f \mapsto \nu}^{n}\rangle \\ f = f_j\end{array}}{\mathcal{H}, \sigma_{\langle m,l\rangle}, o.f \rightsquigarrow \mathcal{H}, \sigma_{\langle m,l\rangle}, \nu_j}
$$

$$
\text{(x-acc)}\ \frac{}{\mathcal{H}, \sigma_{\langle m,l\rangle}, x_v \rightsquigarrow \mathcal{H}, \sigma_{\langle m,l\rangle}, \sigma(x_v)}
\qquad
\text{(this)}\ \frac{}{\mathcal{H}, \sigma_{\langle m,l\rangle}, \texttt{this} \rightsquigarrow \mathcal{H}, \sigma_{\langle m,l\rangle}, \sigma(\texttt{this})}
$$

$$
\text{(jump)}\ \frac{\begin{array}{c}\mathcal{H}(\sigma(\texttt{this})) = \langle c, \_\rangle \\ \mathit{block}(c, m, l') = l' : e\end{array}}{\mathcal{H}, \sigma_{\langle m,l\rangle}, \texttt{jump}\ l' \rightsquigarrow \mathcal{H}, \sigma_{\langle m,l'\rangle}, e}
\qquad
\text{(if)}\ \frac{\begin{array}{c}\mathcal{H}(\sigma(\texttt{this})) = \langle c, \_\rangle \\ l' = \begin{cases} l_1 & \nu = \texttt{true} \\ l_2 & \nu = \texttt{false} \end{cases} \\ \mathit{block}(c, m, l') = l' : e\end{array}}{\begin{array}{l}\mathcal{H}, \sigma_{\langle m,l\rangle}, \texttt{if}\ (\nu)\ \texttt{jump}\ l_1\ \texttt{else jump}\ l_2 \rightsquigarrow \\ \mathcal{H}, \sigma_{\langle m,l'\rangle}, e\end{array}}
$$

$$
\text{(return)}\ \frac{}{\mathcal{H}, \mathcal{D}[\sigma_{\langle m,l\rangle}\{\texttt{return}\ x_v\}] \rightsquigarrow \mathcal{H}, \mathcal{D}[\sigma(x_v)]}
\qquad
\text{(main)}\ \frac{\epsilon_{\mathcal{H}}, \epsilon_{\langle \bot,\bot\rangle}\{e\} \rightsquigarrow^{*}_{max} \mathcal{H}, e}{\overline{cd}^{n}\ e \Rightarrow e}
$$

FIGURE 8. Small-step semantics.

obvious case is the translation of stack frame expressions $\sigma_{\langle m,l\rangle}\{e\}$; all variables in $e$ defined in $\sigma$ must be substituted with the corresponding values, before translating $e$. Particular care must be taken for $\varphi$ functions: variables in $\varphi(x_{v_1},\ldots,x_{v_n})$ must not be substituted even when $\sigma(x_{v_1},\ldots,x_{v_n})$ is defined, otherwise subject reduction does not hold. Consider for instance the following code fragment in SSA form:

```
b1:{x0=0;i0=0;jump b2;}
b2:{x1=φ(x0,x4); i2=φ(i0,i1);
    if(i2>2) jump b7 else jump b3;}
b3:{if(i2%2==0) jump b4 else jump b5;}
b4:{x2=true;jump b6;}
b4:{x3=1;jump b6;}
b6:{x4=φ(x2,x3);i1=i2+1;jump b2;}
b7:{return x1;}
```

The type statically inferred for variable $\mathtt{x}_1$ is $\mathtt{int}\vee\mathtt{bool}$, however during execution $\mathtt{x}_1$ is assigned both an integer and a boolean value, and the type of the returned value can only be decided when block `b7` is executed, otherwise subject reduction would not hold, since `int` and `bool` are not comparable.

To prove progress we need the following lemmas.

**Lemma 6.1.** *If $\mathcal{H},\mathcal{D}[\sigma_{\langle m,l\rangle}\{\mathcal{C}[e]\}] \rightsquigarrow (t \mid B)$, then $\mathcal{H}, e \rightsquigarrow (t' \mid B')$, with $B'' \subseteq B$, where $B''$ is obtained from $B'$ by an appropriate bijective renaming of logical variables.*

*Proof.* By case analysis on the contexts and by induction on their structure. □

**Lemma 6.2.** *If for all $i = 1..n$ $cd_i \rightsquigarrow Hf_i$, $Hf = Hf^d, \overline{Hf}^n$, and there exists a substitution $\theta$ s.t. $invoke(c, m, [t_1,\ldots,t_n], t)\theta$ is in $M^{\leq}_{Hf}$, then $firstBlock(c, m) = l : e$ and $params(c, m) = \overline{x_0}^n$ for some variables $\overline{x_0}^n$ and block $l : e$.*

*Proof.* By induction on the height of the inheritance tree. Recall that by assumption (see Figure 2) inheritance cannot be cyclic. □

**Theorem 6.3** (Progress)**.** *If for all $i = 1..n$ $cd_i \rightsquigarrow Hf_i$, $Hf = Hf^d, \overline{Hf}^n$, $\mathcal{H}, e \rightsquigarrow (t \mid B)$ and there exists a substitution $\theta$ s.t. all atoms in $B\theta$ are contained in $M^{\leq}_{Hf}$, then either $e$ is a value or $\mathcal{H}, e \rightsquigarrow \mathcal{H}', e'$ for some $\mathcal{H}'$ and $e'$.*

The following lemma is instrumental to the proof of the subject reduction property.

**Lemma 6.4.** *If for all $i = 1..n$ $cd_i \rightsquigarrow Hf_i$, $Hf = Hf^d, \overline{Hf}^n$, $\mathcal{H},\sigma, e \rightsquigarrow (t \mid B)$, there exists a substitution $\theta$ s.t. all atoms in $B\theta$ are contained in $M^{\leq}_{Hf}$, and $\mathcal{H},\sigma, e \rightsquigarrow \mathcal{H}',\sigma', e'$, then there exist $t'$, $B'$ and $\theta'$ s.t. $\mathcal{H}',\sigma', e' \rightsquigarrow (t' \mid B')$, all atoms in $B'\theta'$ are contained in $M^{\leq}_{Hf}$, and $t'\theta' \leq t\theta$.*

**Theorem 6.5** (Subject reduction)**.** *If for all $i = 1..n$ $cd_i \rightsquigarrow Hf_i$, $Hf = Hf^d, \overline{Hf}^n$, $\mathcal{H}, e \rightsquigarrow (t \mid B)$, there exists a substitution $\theta$ s.t. all atoms in $B\theta$ are contained in*

$M^{\leq}_{\overline{Hf}}$, *and* $\mathcal{H}, e \rightsquigarrow \mathcal{H}', e'$, *then there exist* $t'$, $B'$ *and* $\theta'$ *s.t.* $\mathcal{H}', e' \rightsquigarrow (t' \mid B')$, *all atoms in* $B'\theta'$ *are contained in* $M^{\leq}_{\overline{Hf}}$, *and* $t'\theta' \leq t\theta$.

Finally, to prove soundness we need the following main lemma.

**Lemma 6.6.** *Let us assume that the following conditions hold: If for all* $i = 1..n$ $cd_i \rightsquigarrow Hf_i$, $Hf = Hf^d, \overline{Hf}^n$, $\mathcal{H}, e \rightsquigarrow (t \mid B)$, *there exists a substitution* $\theta$ *s.t. all atoms in* $B\theta$ *are contained in* $M^{\leq}_{\overline{Hf}}$, $\mathcal{H}, e \rightsquigarrow^* \mathcal{H}', e'$, *and there exist no* $\mathcal{H}''$, $e''$ *s.t.* $\mathcal{H}', e' \rightsquigarrow \mathcal{H}'', e''$. *Then* $e'$ *is a value.*

*Proof.* By induction on the number $n$ of reduction steps. The claim for $n = 0$ holds by progress. If $n > 0$, then there exist $\mathcal{H}''$, $e''$ s.t. $\mathcal{H}, e \rightsquigarrow \mathcal{H}'', e'$, and $\mathcal{H}', e' \rightsquigarrow^* \mathcal{H}, e$ in $n - 1$ steps. By subject reduction we have that there exist $t'$, $B'$ and $\theta'$ s.t. $\mathcal{H}'', e'' \rightsquigarrow (t' \mid B')$, and all atoms in $B'\theta'$ are contained in $M^{\leq}_{\overline{Hf}}$, therefore we can conclude by inductive hypothesis. □

**Theorem 6.7** (Soundness)**.** *Let us assume that the following conditions hold:* $\overline{cd}^n$ $e \rightsquigarrow (Hf|B)$, *there exists a substitution* $\theta$ *s.t. all atoms in* $B\theta$ *are contained in* $M^{\leq}_{\overline{Hf}}$, $\overline{cd}^n$ $e \rightsquigarrow e$. *Then* $e$ *is a value.*

*Proof.* If $\overline{cd}^n$ $e \rightsquigarrow (Hf|B)$, then by definition, for all $i = 1..n$ $cd_i \rightsquigarrow Hf_i$, $e \rightsquigarrow (t \mid B)$, and $Hf = Hf^d, \overline{Hf}^n$. If $e \rightsquigarrow (t \mid B)$, then by definition, $\epsilon_{\mathcal{H}}, e \rightsquigarrow (t \mid B)$; furthermore, if $\overline{cd}^n$ $e \rightsquigarrow e$, then by definition $\epsilon_{\mathcal{H}}, \epsilon_{\langle \bot, \bot \rangle}\{e\} \rightsquigarrow^*_{max} \mathcal{H}, e$, that is, $\epsilon_{\mathcal{H}}, \epsilon_{\langle \bot, \bot \rangle}\{e\} \rightsquigarrow^* \mathcal{H}, e$, and there exist no $\mathcal{H}'$, $e'$ s.t. $\mathcal{H}, e \rightsquigarrow \mathcal{H}', e'$. Therefore we can conclude by lemma 6.6. □

## 7. Related work and conclusion

We have defined abstract compilation [4, 8] for a simple imperative Java-like language to allow precise type inference of imperative constructs as assignment to local variables and object fields, conditional execution and iteration.

This has been achieved by considering programs in SSA form and translating them into Horn formulas where $\varphi$ functions are translated into union types. Furthermore, subtyping between record types has been extended to deal correctly with field updates and to allow more precise type inference in contexts where a field is only accessed (read only) or assigned (write only); finally, the two predicates *var_upd* and *field_upd* have been defined to allow abstract compilation of assignment to local variables and to fields.

We have shown, by means of an example, how the type system can infer types also for infinite objects, like circular lists. The type system has been formalized by providing the full encoding of a simple object-oriented language in SSA form. The small-step semantics of the language has been defined; to our knowledge this is the first formalization of the operational semantics of a language in SSA form. Soundness of the type system w.r.t. such operational semantics has been proved.

Besides type inference, recent work has shown how coinductive logic programming [31] and coinductive CLP can be fruitfully applied to a handful applications ranging over verification of real time systems [25], model checking, and SAT solvers [22].

The first works on constraint-based type inference for object-oriented languages date the early 90s [24, 27, 28]. Agesen has developed the Cartesian Product Algorithm (CPA) [1] to perform more efficient type inference with parametric polymorphism for the language Self. However all these approaches fail to support data polymorphism.

To our knowledge, the only existing approaches in literature able to support data polymorphism are the iterative flow analysis (IFA) proposed by Plevyak and Chien [29] based on iterative refinement of whole program analysis, and the DCPA algorithm [35], an extension of the CPA algorithm able to verify the correctness of Java downcasts. We are not yet able to compare these approaches with ours w.r.t. scalability, since we are still working on a prototype implementation of the inferential engine based on coinductive constraint logic programming [3], to investigate whether it is possible to get a reasonable trade-off between precision and efficiency. However our approach has two main advantages. (1) It is modular: maintaining a strict distinction between the translation phase and the logical inference one, when the goal and the constraints are solved, allows a much clearer specification of type inference and different type inference algorithms can be obtained by just modifying the translation phase, while reusing the same engine defined in the logical inference phase. Similar benefits have been experienced by Sulzmann and Stuckey [33] who have mapped the generalized Hindley/Milner type inference problem HM(X) [26] to inductive CLP(X). (2) Easy integration with compiler technology: an advantage emerged in this paper is that abstract compilation can be easily and fruitfully integrated with optimization techniques used in compiler theory. SSA is just an example, but other more advanced intermediate forms based on virtual register renaming [32], could be exploited to enhance type inference.

There exist several papers on type inference for dynamic object-oriented languages as JavaScript and Ruby [9,17,18,36]. All these works are mainly concerned with object extension and member initialization, but do not support data polymorphism.

Finally, the relation between abstract compilation and abstract interpretation [14] certainly deserves investigation, since our approach is based on the same idea that program analysis, and more specifically type inference [13], can be seen as an abstract semantics which approximates the standard concrete semantics of the program to be analyzed. The main differences are that in our approach the abstract semantics is implemented by a compiler, rather then by an interpreter, and that the semantics is coinductive. A first step towards studying the relationship with abstract interpretation would consist in directly defining an abstract coinductive operational semantics [21] for the language under consideration.

Even though this paper is mainly focused on theoretical aspects, we are intensively working on solving the implementation issues of abstract compilation. A

first prototype[13] has been developed for a small Java-like functional language. Its inferential engine is a Prolog meta-interpreter which implements the operational semantics of coinductive logic programming [31]. Currently we are extending such a meta-interpreter[14] to support subtyping, hence, coinductive constraint logic programming [3].

## References


[1] O. Agesen. The cartesian product algorithm. In W. Olthoff, editor, *ECOOP'05 - Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 1995.

[2] R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[3] D. Ancona, A. Corradi, G. Lagorio, and F. Damiani. Abstract compilation of object-oriented languages into coinductive CLP(X): when type inference meets verification. Technical report, Karlsruhe Institute of Technology, 2010. Formal Verification of Object-Oriented Software. Papers presented at the International Conference, June 28-30, 2010, Paris, France.

[4] D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In S. Drossopoulou, editor, *ECOOP'09 - Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 2009.

[5] D. Ancona and G. Lagorio. Coinductive subtyping for abstract compilation of object-oriented languages into Horn formulas. In *Proceedings of GandALF 2010*, volume 25 of *Electronic Proceedings in Theoretical Computer Science*, pages 214–223, 2010.

[6] D. Ancona and G. Lagorio. Complete coinductive subtyping for abstract compilation of object-oriented languages. In *Formal Techniques for Java-like Programs (FTfJP10)*, ACM Digital Library, 2010.

[7] D. Ancona, G. Lagorio, and E. Zucca. Type inference for polymorphic methods in Java-like languages. In Giuseppe F. Italiano, Eugenio Moggi, and Luigi Laura, editors, *ICTCS'07 - Italian Conf. on Theoretical Computer Science*, eProceedings. World Scientific, 2007.

[8] D. Ancona, G. Lagorio, and E. Zucca. Type inference by coinductive logic programming. In *Post-Proceedings of TYPES'08*, number 5497 in Lecture Notes in Computer Science. Springer, 2009.

[9] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for Javascript. In *ECOOP'05 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 428–452. Springer, 2005.

[10] F. Barbanera, M. Dezani-Cincaglini, and U. de'Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

[11] W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance is not subtyping. In *ACM Symp. on Principles of Programming Languages 1990*, pages 125–135. ACM Press, 1990.

[12] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[13] P. Cousot. Types as abstract interpretations. In *ACM Symp. on Principles of Programming Languages 1997*, pages 316–331, 1997.

[14] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symp. on Principles of Programming Languages 1977*, pages 238–252. ACM Press, 1977.

[15] 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. *ACM Transactions on Programming Languages and Systems*, 13:451–490, 1991.


[13] Available at `http://www.disi.unige.it/person/LagorioG/J2P`.

[14] Available at `ftp://ftp.disi.unige.it/person/AnconaD/coCLP.zip`.

[16] D. Das and U. Ramakrishna. A practical and fast iterative algorithm for phi-function computation using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 27(3):426–440, 2005.

[17] M. Furr, J. An, J. S. Foster, and M. Hicks. Static type inference for Ruby. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied computing*. ACM Press, 2009.

[18] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In T. D'Hondt, editor, *ECOOP'10 - Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 200–224. Springer, 2010.

[19] A. Igarashi and H. Nagira. Union types for object-oriented programming. *Journ. of Object Technology*, 6(2):47–68, 2007.

[20] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[21] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207:284–304, 2009.

[22] R. Min and G. Gupta. Coinductive logic programming and its application to boolean sat. In *FLAIRS Conference*, 2009.

[23] R. Min and G. Gupta. Coinductive logic programming with negation. In *LOPSTR*, pages 97–112, 2009.

[24] N.Oxhøj, J. Palsberg, and M. I. Schwartzbach. Making type inference practical. In *ECOOP'92 - European Conference on Object-Oriented Programming*, pages 329–349, 1992.

[25] N.Saeedloei and G. Gupta. Verifying complex continuous real-time systems with coinductive CLP(R). In *Proc. of LATA 2010*, Lecture Notes in Computer Science. Springer, 2010.

[26] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[27] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1991*, pages 146–161. ACM Press, 1991.

[28] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

[29] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *Ninth Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–340. ACM Press, 1994.

[30] L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007*, pages 472–483, 2007.

[31] L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In *Logic Programming, 22nd International Conference, ICLP 2006*, pages 330–345, 2006.

[32] J. Singer. *Static Program Analysis based on Virtual Register Renaming*. PhD thesis, Computer Laboratory, University of Cambridge, 2006.

[33] M. Sulzmann and P. J. Stuckey. HM(X) type inference is CLP(X) solving. *Journ. of Functional Programming*, 18(2):251–283, 2008.

[34] T. Wang and S. Smith. Polymorphic constraint-based type inference for objects. Technical report, The Johns Hopkins University, 2008. Submitted for publication.

[35] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In *ECOOP'01 - European Conference on Object-Oriented Programming*, volume 2072, pages 99–117. Springer, 2001.

[36] T. Zhao. Type inference for scripting languages with implicit extension. In *FOOL 2010 - Intl. Workshop on Foundations of Object-Oriented Languages*, 2010.

Communicated by (The editor will be set by the publisher).
(The dates will be set by the publisher).