Abstract compilation of object-oriented languages into coinductive CLP(X): can type inference meet verification? (extended version)*

Davide Ancona[1], Andrea Corradi[1], Giovanni Lagorio[1], and Ferruccio Damiani[2]

[1] DISI, University of Genova, Italy
{davide,lagorio}@disi.unige.it,andreac@unstable.it
[2] Dipartimento di Informatica, University of Torino, Italy
damiani@di.unito.it

Abstract. This paper further investigates the potential and practical applicability of *abstract compilation* in two different directions.
First, we formally define an abstract compilation scheme for precise prediction of uncaught exceptions for a simple Java-like language; besides the usual user declared checked exceptions, the analysis covers the runtime `ClassCastException`.
Second, we present a general implementation schema for abstract compilation based on coinductive CLP with variance annotation of user-defined predicates, and propose an implementation based on a Prolog prototype meta-interpreter, parametric in the solver for the subtyping constraints.

1 Introduction

Mapping type checking and type inference algorithms to inductive constraint logic programming (CLP) is not a novel idea. Sulzmann and Stuckey [20] have shown that the generalized Hindley/Milner type inference problem HM(X) [17] can be mapped to inductive CLP(X): type inference of a program can be obtained by first translating it in a set of CLP(X) clauses, and then by resolving a certain goal w.r.t. such clauses. This result is not purely theoretical, indeed it has also some important practical advantages: 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 a more modular approach, since different type inference algorithms can be obtained by just modifying the translation phase, while reusing the same engine defined in the logical inference phase.

Recent work has shown how coinductive logic programming [19] and coinductive CLP can be fruitfully applied to a handful applications ranging over type inference of object-oriented languages [5, 1, 2], verification of real time systems [16], model checking, and SAT solvers [15].

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

Type inference can be defined in terms of *abstract compilation* [5, 1, 2] into a Horn formula of the program to be analyzed, and of resolution of an appropriate goal in coinductive CLP with subtyping constraints. In contrast to conventional inductive CLP, coinductive CLP allows the specification of much more expressive type systems and, therefore, of more precise forms of type analysis able to better detect malfunctioning of a program.

Abstract compilation is particularly interesting for type inference of object-oriented languages when coinduction, union and object types are combined together. A first formal definition of abstract compilation [5, 1] has been given for a purely functional object-oriented language similar to Featherweight Java (FJ) [12] with optional nominal type annotations, generalized explicit constructor declarations and primitive types, but no type casts. The proposed abstract compilation scheme supports precise type inference based on coinductive union and object types, and smoothly integrates it with nominal type annotations, which are managed as additional constraints imposed by the user.

To further investigate the scalability of the approach, we have studied an abstract compilation scheme [4] for a simple Java-like language with imperative features as variable and field assignment and iterative constructs, by considering as source to abstract compilation an SSA [8] intermediate form. Natural encoding of SSA φ functions with union types proves how SSA intermediate forms can be fruitfully exploited by abstract compilation.

Though these results show that abstract compilation is attractive and promising, its full potential has not been completely explored yet, and more efforts are required before the approach can be applied to realistic object-oriented languages. In this paper we add a further step towards the long way to real applicability of abstract compilation, in two different directions. First, we consider an important feature in modern mainstream object-oriented language, namely exception handling, and show an abstract compilation scheme allowing precise prediction of uncaught exceptions for a simple Java-like language. Second, we presents a general implementation schema for abstract compilation based on coinductive CLP, and propose an implementation based on a Prolog prototype meta-interpreter, parametric in the solver for the subtyping constraints. The implementation exploits variance annotations of user-defined predicates to use subsumption instead of simple term unification when the coinductive hypothesis rule is applied.

The paper is organized as follows. Section 2 provides some minimal background on coinductive LP and on inductive CLP. Section 3 introduces abstract compilation with some examples, whereas Section 4 formally defines abstract compilation for a simple Java-like language with exceptions. Section 5 presents a general implementation schema for abstract compilation and is devoted to the semantics and implementation of coinductive CLP, whereas Section 6 draws some conclusions and outlines some directions for further investigation.

2 Background

Coinductive LP and SLD Simon et al [19] have introduced *coinductive-LP*, or simply *co-LP*. Its declarative semantics is given in terms of *co-Herbrand* universe, infinitary Herbrand base and maximal models, computed using greatest fixed-points. While in traditional LP this semantics corresponds to build finite proof trees, co-LP allows infinite terms and proofs as well, which in general are not finitely representable and, for this reason, are called idealized. The operational semantics, defined in a manner similar to SLD, is called *co-SLD*. For an obvious reason, co-SLD is restricted to *regular* terms and proofs, that is, to trees which may be infinite, but can only contain a finite number of different subtrees (and, hence, can be finitely represented). To correctly deal with infinite regular derivations an implicit *coinductive hypothesis rule* is introduced. This rule allows a predicate call to succeed if it unifies with one of its ancestor calls.

CLP(X) CLP introduces *constraints* in the body of the clauses of a logic program, specifying conditions under which the clauses hold, and let external constraint solvers interpret/simplify these constraints. For instance, the clause $p(X) \leftarrow \{X > 3\},\ q(X)$ expresses that $p(X)$ holds when $q(X)$ holds *and* the value of X is greater than three. Furthermore, constraints serve also as answers returned by derivations. For instance, if we add $q(X) \leftarrow \{X > 5\}$ to the clause above, then the goal $p(X)$ succeeds with answer $\{X > 5\}$. Of course, the standard resolution has to be extended in order to embed calls to the external solvers. At each resolution step new constraints are generated and collected, and the solver checks that the whole set of collected constraints is still satisfiable before execution can proceed further.

3 Abstract compilation by examples

This section shows how abstract compilation allows accurate analysis of uncaught exceptions, and informally introduces the main concepts which will be used in the formalization given in Section 4.

The terms of our type domain are class, method and field names (represented by constants), and types coinductively defined over integer, boolean, object, union, and exception types.

$$\begin{array}{lll} bt & ::= int \mid bool & \text{(basic types)} \\ vt & ::= bt \mid obj(c, [f_1{:}vt_1, \ldots, f_n{:}vt_n]) \mid vt_1 \vee vt_2 & \text{(value types)} \\ t & ::= vt \mid t_1 \vee t_2 \mid ex(c) & \text{(types)} \end{array}$$

An object type $obj(c, [f_1{:}vt_1, \ldots, f_n{:}vt_n])$ specifies the class c to which the object belongs, together with the set of available fields with their corresponding value types. A value type does not contain exception types, and represents a set of values. Exception types are inferred for expressions whose evaluation throws an exception, hence cannot be associated with a field or with the parameter of a method. The class name of the object type is needed for typing method invocations. We assume that fields in an object type are finite, distinct and that their order is immaterial. Union types $t_1 \vee t_2$ have the standard meaning [6, 11].

Finally, if an an expression has type $ex(c)$, then it means that its evaluation throws an exception of class c. In general we expect the type of an expression to be the union of value and exception types; for instance, if an expression has type $ex(c) \vee int$, then it means that its evaluation may either throw an exception of class c, or return an integer value. However, if the type of an expression is the union of sole exception types, then it means that the evaluation of that expression will always throw an exception, thus revealing a problem in the program. This accurate analysis is not possible in the approach of Jo et al. [14].

As pointed out in Section 2, in coinductive logic programming terms and derivations can correspond to arbitrary infinite trees [7], hence not all the terms and derivations can be represented in a finite way, therefore the corresponding type systems are called *idealized*. However, an implementable sound approximation of an idealized type system can be obtained by restricting terms and derivations to regular ones. A regular tree can be infinite, but can only contain a finite number of subtrees or, equivalently, can be represented as the solution of a unification problem, that is, a finite set of syntactic equations of the form $X_i = e_i$, where all variables X_i are distinct and expressions e_i may only contain variables X_i [7, 19, 18].

A *type domain* $\mathcal{D}$ is a constraint domain which defines two predicates: strong equivalence and subtyping. In this example strong equivalence corresponds to syntactic equality and is interpreted in the coinductive Herbrand universe, whereas subtyping is interpreted as set inclusion between sets of values: $t_1 \leq t_2$ iff $[\![t_1]\!] \subseteq [\![t_2]\!]$, where $[\![t]\!]$ depends on the considered type language. For space limitation, we have omitted the definition of interpretation for our types; however, the definition given by Ancona and Lagorio [2] can be extended in a straightforward way to deal with exception types too.

An accurate analysis of uncaught exceptions We show by a simple example how abstract compilation allows accurate uncaught exceptions in Java-like languages. We share the same motivations as in the work by Jo et al. [14]: an analysis of uncaught exceptions independent of declared thrown exceptions is a valuable tool for avoiding unnecessary or too broad declarations, and, hence, unnecessary **try** statements or too general error handling. Last but not least, reporting some kinds of unchecked exceptions, as `ClassCastException`, would allow static detection of typical run-time errors. This last feature is supported in the language defined in Section 4.

Consider the following example of Java[3] code.

```
class Exc extends Exception {}

interface Node {Node next() throws Exc;} // linked nodes

class TNode implements Node {  // terminal nodes
    public Node next() throws Exc {throw new Exc();}
```

[3] For clarity we use full Java, even though this example could be easily recast in the language defined in Section 4.

```
}

class NTNode implements Node { // non terminal nodes
    private Node next;
    public NTNode(Node n){this.next=n;}
    public Node next() {return this.next;}
}

class Test {
    void m() throws Exc { // Exc must be declared
                          // but will never been thrown
        new NTNode(new NTNode(new TNode())).next().next();}
}
```

In order to be correctly compiled, method `m` must declare `Exc` in its **throws** clause, or its body must be wrapped by a dummy **try** statement, even though such a method will never throw an exception of type `Exc`. The **throws** clause can be safely removed, if a type more precise than `Node` is inferred for the expression **new** `NTNode(`**new** `NTNode(`**new** `TNode())).next()`; indeed, by abstract compilation it is possible to infer the type $obj(ntnode, [next{:}obj(tnode, [\,])])$ and, hence, deduce that the second call to `next()` cannot throw an exception.

To compile the program shown above into a Horn formula, we introduce a predicate for each language construct; for instance, *invoke* for method invocation, *new* for constructor invocation, *field_acc* for field access, and *cond* for conditional expressions. Furthermore, auxiliary predicates are introduced for expressing the semantics of the language; for instance, predicate *has_meth* corresponds to method look-up. Each method declaration is abstractly compiled into a Horn clause: the compilation of method `next()` of classes `TNode` and `NTNode` generates the following two clauses, respectively.

```
has_meth(tnode,next,[This],ex(exc)).
has_meth(ntnode,next,[This],N) ← field_acc(This,next,N).
```

Predicate *has_meth* has four arguments: the class where the method is declared, the name of the method, the types of the arguments, and the type of the returned value. If a method has n arguments, then its argument type is a list of $n+1$ types, where the first type always corresponds to the target object **this**. The first clause is a fact specifying that method `next()` declared in class `TNode` always throws an exception. The second clause has a non empty body corresponding to the abstract compilation of the body of the method: `field_acc(This,next,N)` means that accessing field `next` of the object `This` returns a value of type `N`.

Each method declaration is compiled into a clause defining predicate *has_meth*, and, analogously, each constructor declaration is compiled into a clause defining predicate *has_constr*. Furthermore, other program independent clauses are generated to specify the behavior of the various constructs w.r.t. the available types (see Section 4).

Coinductive derivations and subtyping To see an example of coinductive derivation and to explain the importance of subtyping constraints, let us add the following factory method[4] to class `Test`.

```
Node addNodes(int i, Node n) { // adds i nodes before n
     if(i<=0) return n;
     else return addNodes(i-1,new NTNode(n));}
```

Let us assume now that we would like to infer the type of the expression `new Test().addNodes(5,new TNode())`; the inferred type can be obtained by resolving the goal $\mathit{invoke}(\mathit{obj}(\mathit{test},[\,]),\mathit{addNodes},[\mathit{int},\mathit{obj}(\mathit{tnode},[\,])],R_0)$ w.r.t. the Horn formula obtained from the abstract compilation of our example classes.

If we consider unification with no subtyping constraints, then we can only get an infinite derivation containing the following sequence of atoms:

$$
\begin{array}{c}
\mathit{at}_0 = \mathit{invoke}(\mathit{obj}(\mathit{test},[\,]),\mathit{addNodes},[\mathit{int},t_0],R_0)\\
\mathit{at}_1 = \mathit{invoke}(\mathit{obj}(\mathit{test},[\,]),\mathit{addNodes},[\mathit{int},t_1],R_1)\\
\vdots\\
\mathit{at}_k = \mathit{invoke}(\mathit{obj}(\mathit{test},[\,]),\mathit{addNodes},[\mathit{int},t_k],R_k)\\
\vdots
\end{array}
$$

with answer $R_0 = t_0 \vee R_1,\ldots,R_k = t_k \vee R_{k+1},\ldots$, where $t_0 = \mathit{obj}(\mathit{tnode},[\,])$, and $t_{k+1} = \mathit{obj}(\mathit{ntnode},[n{:}t_k])$ for all $k \geq 0$; hence, the solution is a non regular term obtained from a non regular derivation. The main problem is that for all k, atom at_k does not unify with atoms $\mathit{at}_0,\ldots,\mathit{at}_{k-1}$, hence no coinductive hypothesis can be used to build a regular proof.

If we consider subtyping and observe that method invocation is contravariant in the argument type and covariant in the returned type, then we have that[5] our initial goal succeeds if the atom $\mathit{at} = \mathit{invoke}(\mathit{obj}(\mathit{test},[\,]),\mathit{addNodes},[\mathit{int},T],R_0)$ succeeds, and $t_0 \leq T$ holds. To resolve at, the following atom at' needs to be resolved, under the constraint $R_0 \geq T \vee R_1$:

$$\mathit{invoke}(\mathit{obj}(\mathit{test},[\,]),\mathit{addNodes},[\mathit{int},\mathit{obj}(\mathit{ntnode},[n{:}T])],R_1)$$

To derive at' we can use the coinductive hypothesis at if the additional constraints $R_1 \geq R_0$ and $\mathit{obj}(\mathit{ntnode},[n{:}T]) \leq T$ hold. Hence, the initial goal can be resolved if the following set of constraints is satisfiable:

$$t_0 \leq T, R_0 \geq T \vee R_1, R_1 \geq R_0, \mathit{obj}(\mathit{ntnode},[n{:}T]) \leq T.$$

A possible solution is given by $R_0 = R_1 = T, T = t$, where t is the regular term t s.t. $t = t_0 \vee \mathit{obj}(\mathit{ntnode},[n{:}t])$. Therefore, by exploiting the subtyping constraint $\leq$, we can resolve our goal with a regular derivation and a regular solution

[4] This is just a simple example in our functional Java-like language; in Java the method would be static and tail recursion would be replaced with a loop.

[5] The resolution steps have been slightly simplified for the sake of clarity.

(other interesting examples of regular derivations, which can be computed by considering the subtyping constraint, can be find in related papers [1–3]).

The derivation sketched above follows the rules of coinductive CLP as defined in Section 5, where user-defined predicates are associated with *variance annotations*.

The key point is that each predicate is expected to behave in a specific way w.r.t. subtyping. If p is a predicate with only one[6] argument, we have the following four possibilities:

- p is *covariant* in its argument: if $p(t_1)$ and $t_1 \leq t_2$ hold, then $p(t_2)$ holds has well (we say that $p(t_1)$ subsumes $p(t_2)$).
- p is *contravariant* in its argument: if $t_2 \leq t_1$ holds, then $p(t_1)$ subsumes $p(t_2)$.
- p is *weakly invariant* in its argument: if $t_1 \leq t_2, t_1 \geq t_2$ holds, then $p(t_1)$ subsumes $p(t_2)$. In this case we abbreviate $t_1 \leq t_2, t_1 \geq t_2$ with $t_1 \cong t_2$, and we call $\cong$ *weak equivalence*.
- p is *strongly invariant* in its argument: if $t_1 \equiv t_2$ holds, then $p(t_1)$ subsumes $p(t_2)$. We call $\equiv$ *strong equivalence* since it is expected to be stronger than $\cong$, that is $t_1 \equiv t_2 \Rightarrow t_1 \cong t_2$, but not conversely. In most cases $\equiv$ coincides with syntactic equality.

For instance, *invoke* is strongly invariant w.r.t. its first[7] and second arguments, contravariant in its third argument, and covariant in its fourth argument.

4 Formalization

In this section we formally define abstract compilation for a simple functional Java-like language supporting exceptions (Figure 1). Syntactic assumptions listed in the figure have to be verified before abstract compilation is performed. Bars denote sequences of n items, where n is the superscript of the bar.

A program consists of a sequence of class declarations and a main expression. Type annotations in all declarations can be either primitive types *bool* or *int*, or class names. We assume that the language supports boxing conversions, hence *bool* and *int* are both subtypes of *Object*. Hence *Object* is the top type annotation which, in fact, does not impose any restriction on the type of fields, parameters and returned values.

A class declaration contains field and method declarations, and a single constructor declaration. We assume predefined classes *Object*, *Throwable* and *ClassCastExc*: the first is the root of the inheritance tree, the second extends *Object* and is the most general type for exceptions, the third extends *Throwable* and is the class of the unchecked exceptions thrown when a runtime type check fails; for simplicity, we assume that all three classes contain no fields and methods

[6] The definition can be straightforwardly generalized for an arbitrary number of arguments.

[7] In contrast with what intuition may suggest, weak invariance in the first argument of *invoke* is unsound.

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \overline{cd}^{n}\ e \\
cd & ::= & \texttt{class}\ c_1\ \texttt{extends}\ c_2\ \{\ \overline{fd}^{n}\ cn\ \overline{md}^{k}\ \} \quad (c_1 \neq \mathit{Object}, \mathit{Throwable}, \mathit{ClassCastExc}) \\
fd & ::= & \tau\ f; \\
cn & ::= & c(\overline{\tau\ x}^{n})\ \{\texttt{super}(\overline{e}^{k}); \overline{f = e';}^{h}\} \\
md & ::= & \tau_0\ m(\overline{\tau\ x}^{n})\ \{e\} \\
e & ::= & \texttt{new}\ c(\overline{e}^{n}) \mid x \mid e.f \mid e_0.m(\overline{e}^{n}) \mid \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2 \mid \texttt{false} \mid \texttt{true} \mid i \mid e_1\ op\ e_2 \\
 & & \texttt{throw}\ c \mid \texttt{try}\ e_1\ \texttt{catch}(c)\ e_2 \mid (c)\ e \\
op & ::= & \mathit{relOp} \mid \mathit{boolOp} \mid \mathit{intOp} \\
\tau & ::= & c \mid \texttt{bool} \mid \texttt{int}
\end{array}
$$

Assumptions: $n, k, h \geq 0$, inheritance is acyclic, names of declared classes in a program, methods and fields in a class, and parameters in a method are distinct.

Fig. 1. Syntax of the language

and have a constructor without parameters. The body of a constructor consists of an invocation of the superclass constructor and a sequence of field initializations, one for each field declared in the class. Method declarations are standard; note however that they do not include **throws** clause, since our analysis is independent of the declared thrown exceptions.

Expressions deserve few comments: i denotes integer literals, *relOp*, *boolOp* and *intOp* denotes the usual relational, boolean and integer binary operators; for simplicity, we consider `==` and `!=` monomorphic operators over integers; an extension allowing `==` and `!=` to be polymorphic is straightforward. Expressions for exception handling have been deliberately simplified to make the presentation lighter: when an exception is thrown, no instance is created, but only the type of the exception (which is required to be a subtypes of *Throwable*) is specified; consequently, catch clauses do not have any formal parameter. Furthermore **try** expressions can have only one catch clause.

For space reasons we have omitted the quite standard operational semantics of the language. Abstract compilation for programs and declarations (defined in Figure 2) is very similar to those given in previous work ([1, 4]).

Abstract compilation of a program generates a pair $(Hf|B)$, where Hf is a Horn formula and B is a goal (a sequence of atoms). Abstract compilation of a class, field, constructor, and method declaration yields two clauses (for classes and methods) or one (for fields and constructors).

For simplicity class, field, method and variable names are not affected by abstract compilation, even though in practice appropriate bijections[8] (different from the identity) have to be considered.

For any expression e, the abstract compilation of e (defined in Figure 3) generates a pair $(t|B)$, where t is the term corresponding to the type of e, and B is the sequence of atoms whose satisfaction ensures that e is well-typed.

[8] This is due to the fact that in logic programming names beginning with an upper case letter denote logical variables, while those beginning with a lower case letter denote constant, function and predicate symbols.

$$(\text{prog})\frac{\forall\, i = 1..n\ cd_i \leadsto Hf_i \qquad e\ \mathbf{in}\ \emptyset \leadsto (t \mid B)}{\overline{cd}^n\ e \leadsto (Hf_0 \cup (\cup_{i=1..n} Hf_i) | B)}\ Hf_0 \text{ defined in Fig.4 and 5}$$

$$(\text{class})\frac{\forall\, i = 1..n\ fd_i\ \mathbf{in}\ c_1 \leadsto Cl_i \quad cn\ \mathbf{in}\ \overline{fd}^n \leadsto Cl \quad \forall\, j = 1..k\ md_j\ \mathbf{in}\ c_1 \leadsto Hf_j}{\begin{array}{l}\texttt{class}\ c_1\ \texttt{extends}\ c_2\ \{\ \overline{fd}^n\ cn\ \overline{md}^k\ \} \leadsto \left\{\begin{array}{l} \mathit{class}(c_1) \leftarrow \mathit{true}. \\ \mathit{extends}(c_1, c_2) \leftarrow \mathit{true}. \end{array}\right\} \bigcup \\ \qquad\qquad\qquad\qquad\qquad\qquad \bigcup_{i=1..n} \{Cl_i\} \bigcup \{Cl\} \bigcup_{j=1..k} Hf_j \end{array}}$$

$$(\text{field})\frac{}{\tau\ f;\ \mathbf{in}\ c \leadsto \mathit{dec_field}(c, f, \tau) \leftarrow \mathit{true}.}$$

$$(\text{constr})\frac{\forall\, i = 1..k\ e_i\ \mathbf{in}\ \{\overline{x}^n\} \leadsto (t_i \mid B_i) \quad \forall\, j = 1..h\ e'_j\ \mathbf{in}\ \{\overline{x}^n\} \leadsto (t'_j \mid B'_j)}{\begin{array}{l} c(\overline{\tau\ x}^n)\ \{\texttt{super}(\overline{e}^k); \overline{f = e;}^h\}\ \mathbf{in}\ \overline{\tau'\ f;}^h \leadsto \\ \quad \mathit{has_constr}(c, [\overline{x}^n], \mathit{obj}(c, [\overline{f{:}t'}^h | R])) \leftarrow \begin{array}[t]{l} \mathit{type_comp}([\overline{x}^n], [\overline{\tau}^n]), \overline{B}^k, \\ \mathit{extends}(c, P), \\ \mathit{has_constr}(P, [\overline{t}^k], \mathit{obj}(P, R)), \overline{B'}^h, \\ \mathit{type_comp}([\overline{t'}^h], [\overline{\tau'}^h]). \end{array} \end{array}}$$

$$(\text{meth})\frac{e\ \mathbf{in}\ \{\mathit{This}, \overline{x}^n\} \leadsto (t \mid B)}{\begin{array}{l} \tau_0\ m(\overline{\tau\ x}^n)\{e\}\ \mathbf{in}\ c \leadsto \\ \quad \mathit{dec_meth}(c, m) \leftarrow \mathit{true}. \\ \quad \mathit{has_meth}(c, m, [\mathit{This}, \overline{x}^n], t) \leftarrow \begin{array}[t]{l} \mathit{type_comp}(\mathit{This}, c), \\ \mathit{type_comp}([\overline{x}^n], [\overline{\tau}^n]), \\ B, \mathit{type_comp}(t, \tau_0). \end{array} \end{array}}$$

Fig. 2. Abstract compilation of programs and declarations

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

$$(\text{var})\ \frac{}{x\ \textbf{in}\ V \leadsto (x \mid \mathit{true})}\ x \in V \qquad (\text{field-acc})\ \frac{e\ \textbf{in}\ V \leadsto (t \mid B)}{e.f\ \textbf{in}\ V \leadsto (R \mid B, \mathit{field_acc}(t, f, R))}\ R\ \text{fresh}$$

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

$$(\text{if})\ \frac{e\ \textbf{in}\ V \leadsto (t \mid B) \quad e_1\ \textbf{in}\ V \leadsto (t_1 \mid B_1) \quad e_2\ \textbf{in}\ V \leadsto (t_2 \mid B_2)}{\texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2\ \textbf{in}\ V \leadsto (R \mid B, B_1, B_2, \mathit{cond}(t, t_1, t_2, R))}\ R\ \text{fresh}$$

$$(\text{true})\ \frac{}{\texttt{true}\ \textbf{in}\ V \leadsto (\mathit{bool} \mid \mathit{true})} \qquad (\text{false})\ \frac{}{\texttt{false}\ \textbf{in}\ V \leadsto (\mathit{bool} \mid \mathit{true})}$$

$$(\text{int})\ \frac{}{i\ \textbf{in}\ V \leadsto (\mathit{int} \mid \mathit{true})} \qquad (\text{bin-op})\ \frac{e_1\ \textbf{in}\ V \leadsto (t_1 \mid B_1) \quad e_2\ \textbf{in}\ V \leadsto (t_2 \mid B_2)}{e_1\ \mathit{op}\ e_2\ \textbf{in}\ V \leadsto (R \mid B_1, B_2, \mathit{bin_op}(\mathit{op}, t_1, t_2, R))}\ R\ \text{fresh}$$

$$(\text{throw})\ \frac{}{\texttt{throw}\ c\ \textbf{in}\ V \leadsto (R \mid \mathit{throw}(c, R))}\ R\ \text{fresh}$$

$$(\text{try})\ \frac{e_1\ \textbf{in}\ V \leadsto (t_1 \mid B_1) \quad e_2\ \textbf{in}\ V \leadsto (t_2 \mid B_2)}{\texttt{try}\ e_1\ \texttt{catch}(c)\ e_2\ \textbf{in}\ V \leadsto (R \mid B_1, B_2, \mathit{try}(t_1, c, t_2, R))}\ R\ \text{fresh}$$

$$(\text{cast})\ \frac{e\ \textbf{in}\ V \leadsto (t \mid B)}{(c)\ e\ \textbf{in}\ V \leadsto (R \mid B, \mathit{cast}(c, t, R))}\ R\ \text{fresh}$$

Fig. 3. Abstract compilation of expressions

The compilation is straightforward and is based on a set of predicates which specify the behavior of each construct. For instance, predicates *invoke* is defined as follows:

```
invoke(obj(C,R),M,A1,RT∨ET) ← val_types(A1,A2),
              exc_types(A1,ET),has_meth(C,M,[obj(C,R)|A2],RT).
invoke(obj(C,R),M,A,ET) ← no_val_types(A), exc_types(A,ET).
invoke(T1∨T2,M,A,RT1∨RT2) ← invoke(T1,M,A,RT1),
                            invoke(T2,M,A,RT2).
invoke(ex(C),M,A,ex(C)).
```

The first two clauses specify the behavior of method calls when the target is an object type. The predicates `val_types`, `exc_types`, and `no_val_types` (defined in Figure 5) control exception propagation during argument evaluation. The atom `val_types(`l`,`l'`)` succeeds only if type list l corresponds to an expression sequence whose evaluation may be completed normally (see Section 14.1, [10]) with type list l' (which necessarily contains no exception types). For instance, `val_types([`$ex(c_1) \vee vt_1$`,`$ex(c_2) \vee vt_2$`],[`vt_1`,`vt_2`])` succeeds, whereas `val_types([`$ex(c_1)$`,`$ex(c_2) \vee vt_2$`],X)` fails. The atom `exc_types(`l`,`t`)` succeeds if l corresponds to an expression sequence whose evaluation may be completed abruptly with type t (which necessarily does not contain value types). For instance, `exc_types([`$ex(c_1) \vee vt_1$`,`$ex(c_2) \vee vt_2$`],`$ex(c_1) \vee ex(c_2)$`)` succeeds; note that `exc_types` succeeds also when no exceptions are thrown: `exc_types([`vt_1`,`vt_2`],X)` succeeds with `X=`$\perp$ (that is, the empty[9] type). Finally, `no_val_types(`l`)` succeeds iff `val_types(`l`,X)` fails.

The first clause of *invoke* deals with cases where argument expressions may evaluate normally. Method look-up is started (predicate `has_meth`) from the class of the target object, and its type is added as first argument to correctly deal with **this**. Note that this case does not prevent argument expressions to evaluate abruptly: `ET` represents all thrown exceptions. The second clause is used when argument expressions never evaluate normally: in this case no method look-up is performed[10]; this clause allows exact propagation of union types containing sole exception types, thus inferring that the evaluation of the method invocation will always throw an exception, and, hence, that something is wrong in the source code.

The third clause of *invoke* deals with union types: if invoking method `M` on a target of type `T1` (resp. `T2`) yields a result of type `RT1` (resp. `RT2`), then invoking `M` on a target of type `T1 ∨ T2` yields a result of type `RT1 ∨ RT2`.

Finally, the last clause deals with the case when the expression corresponding to the target evaluates abruptly by throwing an exception `exc(C)` which, therefore, is propagated.

Let us focus now on the predicates corresponding to the **throw** and **try** constructs. The clause for *throw* is pretty straightforward:

[9] The empty type can be simply represented by the type t s.t. $t = t \vee t$ [2, 3].

[10] This allows typechecking an invocation of *any* method `M` with *any* arguments `A`; though the method will never be actually invoked, since the evaluation of its arguments will *always* throw an exception.

```
throw(C,ex(C)) ← subtype(C,throwable).
```

The type of the expression **throw**(c) is $ex(c)$, providing that c is a subtype of *Throwable*, otherwise the expression is not well-typed.

Since the behavior of the **try** expression is more involved, let us consider first some examples. If e_1 and e_2 have type $t_1 = ex(c_1) \vee ex(c_2) \vee vt$ and t_2, respectively, and if c_1 is a subclass of c, while c_2 is not, then the type inferred for `try` e_1 `catch`(c) e_2 is $ex(c_2) \vee vt \vee t_2$. On the other hand, if both c_1 and c_2 are not subclasses of c, then the inferred type is just t_1. Indeed, expression e_2 is evaluated only if e_1 throws an exception which is a subclass of c, hence the type of the **try** expression includes t_2 only when t_1 contains an exception type handled by the **catch** clause. Furthermore, all exception types in t_1 handled by the **catch** clause have to be removed from t_1 to infer the most precise type.

```
try(T1,C,T2,T3∨T2) ← remove_handled(T1,C,T3).
try(T1,C,T2,T1) ← unhandled(T1,C).
```

The auxiliary predicate `remove_handled(T1,C,T3)` succeeds if `T1` contains at least an exception type covered by `C`, and the type `T3` is obtained from `T1` by removing all exception types covered by `C`; the auxiliary predicate `unhandled(T1,C)` succeeds if `T1` does not contain an exception type covered by `C`. The complete definition of all main and auxiliary predicates can be found in Figures 4 and 5.

5 A prototype implementation of coinductive CLP(X)

In this section we show a prototype implementation of the inference engine for coinductive CLP, which is an essential component for supporting abstract compilation, as depicted in Figure 6. The input is represented by the source program to be analyzed and by a query defined by the user in a high level language. Then the abstract compiler and the goal generator, which is a subcomponent of the abstract compiler, generate a Horn formula and a goal. The generated clauses can be optionally augmented by user-defined clauses defining auxiliary predicates. Finally, type inference is performed by the coinductive CLP engine. The red (or dark) components are those depending on the type system under consideration: the abstract compiler[11] and the solver for the specific type domain.

The engine supports variance annotations, which are more than a convenient syntactic notation for avoiding explicit insertion of constraints in the body of clauses; indeed, they allow definition of constraints which are associated with predicates, rather than clauses. To our knowledge, this is a novel feature not previously considered in CLP. In this way, we gain in expressive power, since instead of unification, subsumption can be exploited when the coinductive hypothesis rule is applied.

We first provide the fixed-point and operational semantics of coinductive CLP.

[11] If the source language is unchanged only the back-end will be modified.

$class(object) \leftarrow true.$
$class(throwable) \leftarrow true.$
$class(class_cast_exc) \leftarrow true.$
$extends(throwable, object) \leftarrow true.$
$extends(class_cast_exc, throwable) \leftarrow true.$
$subtype(X, X) \leftarrow type(X).$
$subtype(X, object) \leftarrow type(X).$
$subtype(X, Y) \leftarrow extends(X, Z), subtype(Z, Y).$
$type(bool) \leftarrow true.$
$type(int) \leftarrow true.$
$type(C) \leftarrow class(C).$
$type_comp(bool, bool) \leftarrow true.$
$type_comp(int, int) \leftarrow true.$
$type_comp([\,], [\,]) \leftarrow true.$
$type_comp([T1|L1], [T2|L2]) \leftarrow type_comp(T1, T2), type_comp(L1, L2).$
$type_comp(obj(C1, X), C2) \leftarrow subtype(C1, C2).$
$type_comp(T1 \vee T2, C) \leftarrow type_comp(T1, C), type_comp(T2, C).$
$bin_op(O, T1, T2, R \vee E) \leftarrow val_types([T1, T2], [T3, T4]), exc_types([T1, T2], E), ev_bin_op(O, T3, T4, R).$
$bin_op(O, T1, T2, E) \leftarrow no_val_types([T1, T2]), exc_types([T1, T2], E).$
$field_acc(obj(C, R), F, T) \leftarrow has_field(C, F, TA), field(R, F, T), type_comp(T, TA).$
$field_acc(T1 \vee T2, F, FT1 \vee FT2) \leftarrow field_acc(T1, F, FT1), field_acc(T1, F, FT1).$
$field([F{:}T|R], F, T) \leftarrow true.$
$field([F1{:}T1|R], F2, T) \leftarrow field(R, F2, T), F1 \neq F2.$
$invoke(obj(C, S), M, A1, R \vee E) \leftarrow val_types(A1, A2), exc_types(A1, ET), has_meth(C, M, [obj(C, S)|A2], R).$
$invoke(obj(C, S), M, A, E) \leftarrow no_val_types(A), exc_types(A, E).$
$invoke(T1 \vee T2, M, A, R1 \vee R2) \leftarrow invoke(T1, M, A, R1), invoke(T2, M, A, R2).$
$invoke(ex(C), M, A, ex(C)) \leftarrow true.$
$has_constr(object, [\,], obj(object, [\,])) \leftarrow true.$
$has_constr(throwable, [\,], obj(throwable, [\,])) \leftarrow true.$
$has_constr(class_cast_exc, [\,], obj(class_cast_exc, [\,])) \leftarrow true.$
$new(C, A1, R \vee E) \leftarrow val_types(A1, A2), exc_types(A1, E), has_constr(C, A2, R).$
$new(C, A, E) \leftarrow no_val_types(A), exc_types(A, E).$
$has_field(C, F, T) \leftarrow dec_field(C, F, T).$
$has_field(C, F, T1) \leftarrow extends(C, P), has_field(P, F, T1), \neg dec_field(C, F, T2).$
$has_meth(C, M, A, R) \leftarrow extends(C, P), has_meth(P, M, A, R), \neg dec_meth(C, M).$
$cond(T1, T2, T3, T2 \vee T3 \vee E) \leftarrow val_type(T1, T4), exc_type(T1, E), type_comp(T4, bool).$
$cond(T1, T2, T3, T1) \leftarrow no_val_type(T1).$
$throw(C, ex(C)) \leftarrow subtype(C, throwable).$
$try(T1, C, T2, T3 \vee T2) \leftarrow remove_handled(T1, C, T3).$
$try(T1, C, T2, T1) \leftarrow unhandled(T1).$
$cast(C, bool, bool) \leftarrow subtype(bool, C).$
$cast(C, int, int) \leftarrow subtype(int, C).$
$cast(C1, obj(C2, R), obj(C2, R)) \leftarrow subtype(C2, C1).$
$cast(C1, obj(C2, R), ex(class_cast_exc)) \leftarrow not_subtype(C2, C1).$
$cast(C, T1 \vee T2, R1 \vee R2) \leftarrow cast(C, T1, R1), cast(C, T2, R2).$
$cast(C1, ex(C2), ex(C2)) \leftarrow true.$

Fig. 4. Definition of the shared clauses Hf_0 (part one)

$val_types([\,],[\,]) \leftarrow true.$
$val_types([T1|L1],[T2|L2]) \leftarrow val_type(T1,T2), val_types(L1,L2).$
$no_val_types([T|L]) \leftarrow no_val_type(T).$
$no_val_types([T|L]) \leftarrow no_val_types(L).$
$val_type(bool,bool) \leftarrow true.$
$val_type(int,int) \leftarrow true.$
$val_type(obj(C,R),obj(C,R)) \leftarrow true.$
$val_type(T1 \vee T2, T3 \vee T4) \leftarrow val_type(T1,T3), val_type(T2,T4).$
$val_type(T1 \vee T2, T3) \leftarrow val_type(T1,T3), no_val_type(T2).$
$val_type(T1 \vee T2, T3) \leftarrow no_val_type(T1), val_type(T2,T3).$
$no_val_type(ex(C)) \leftarrow true.$
$no_val_type(T1 \vee T2) \leftarrow no_val_type(T1), no_val_type(T2).$
$exc_types([\,],E) \leftarrow is_empty(E).$
$exc_types([T|L], E1 \vee E2) \leftarrow exc_type(T,E1), val_type(T,T2), exc_types(L,E2).$
$exc_types([T|L],E) \leftarrow exc_type(T,E), no_val_type(T).$
$exc_type(bool,E) \leftarrow is_empty(E).$
$exc_type(int,E) \leftarrow is_empty(E).$
$exc_type(obj(C,R),E) \leftarrow is_empty(E).$
$exc_type(ex(C),ex(C)) \leftarrow true.$
$exc_type(T1 \vee T2, E1 \vee E2) \leftarrow exc_type(T1,E1), exc_type(T2,E2).$
$remove_handled(ex(C1),C2,E) \leftarrow subtype(C1,C2), is_empty(E).$
$remove_handled(T1 \vee T2, C, T3 \vee T4) \leftarrow remove_handled(T1,C,T3), remove_handled(T2,C,T4).$
$remove_handled(T1 \vee T2, C, T3 \vee T2) \leftarrow remove_handled(T1,C,T3), unhandled(T2,C).$
$remove_handled(T1 \vee T2, C, T1 \vee T3) \leftarrow unhandled(T1,C), remove_handled(T2,C,T3).$
$unhandled(bool,C) \leftarrow true.$
$unhandled(int,C) \leftarrow true.$
$unhandled(obj(C1,R),C2) \leftarrow true.$
$unhandled(ex(C1),C2) \leftarrow not_subtype(C1,C2).$
$unhandled(T1 \vee T2, C) \leftarrow unhandled(T1,C), unhandled(T2,C).$
$not_subtype(bool,T) \leftarrow T \neq object, T \neq bool.$
$not_subtype(int,T) \leftarrow T \neq object, T \neq int.$
$not_subtype(object,C) \leftarrow C \neq object.$
$not_subtype(C1,C2) \leftarrow C1 \neq C2, extends(C1,C3), not_subtype(C3,C2).$
$is_empty(E) \leftarrow E = E \vee E.$
$ev_bin_op(O,T1,T2,R) \leftarrow is_rel_op(O), type_comp(T1,int), type_comp(T2,int), type_comp(R,bool).$
$ev_bin_op(O,T1,T2,R) \leftarrow is_bool_op(O), type_comp(T1,bool), type_comp(T2,bool), type_comp(R,bool).$
$ev_bin_op(O,T1,T2,R) \leftarrow is_int_op(O), type_comp(T1,int), type_comp(T2,int), type_comp(R,int).$
$is_rel_op('<') \leftarrow true. \ldots$
$is_bool_op('\&\&') \leftarrow true. \ldots$
$is_int_op('+') \leftarrow true. \ldots$

Fig. 5. Definition of the shared clauses Hf_0 (part two)

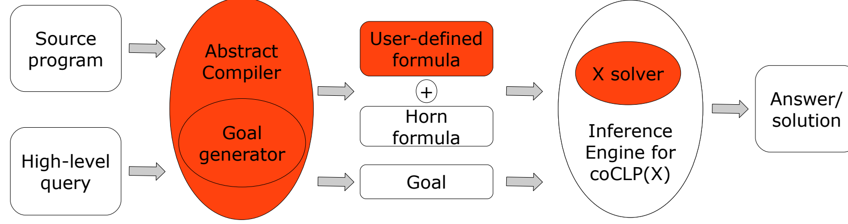

Fig. 6. General schema for abstract compilation based on coinductive CLP(X)

Fixed-point semantics For simplicity, all following definitions use a fixed coinductive Herbrand universe and base and type domain $\mathcal{D}$.

We write $p_{\overline{\alpha}^n}$ to mean that predicate symbol p has arity n and variance annotation $\overline{\alpha}^n$, where each α_i may be one of the constraint predicates $\{\leq, \geq, \cong, \equiv\}$, as defined in Sect. 3.

Definition 1. *If $p_{\overline{\alpha}^n}$ is a predicate symbol, then the ground atom $p_{\overline{\alpha}^n}(t_1, \ldots, t_n)$ subsumes the ground atom $p_{\overline{\alpha}^n}(t'_1, \ldots, t'_n)$ iff $\{t_1 \alpha_1 t'_1, \ldots, t_n \alpha_n t'_n\}$ is satisfiable, that is, $\mathcal{D} \models \{t_1 \alpha_1 t'_1, \ldots, t_n \alpha_n t'_n\}$.*

The one-step consequence function $T_{Hf,\mathcal{D}}$, induced by a Horn formula Hf where all predicates are associated with a variance annotation, is the function over sets of ground atoms contained in the coinductive Herbrand base, defined as follows:

$$T_{Hf,\mathcal{D}}(S) = \{A' \mid A \leftarrow A_1, \ldots, A_n \text{ ground instance of a clause in } Hf, \\ A_i \in S \text{ for all } i = 1, \ldots, n,\ A \text{ subsumes } A'\}$$

The coinductive Herbrand model of Hf w.r.t. the type domain $\mathcal{D}$ is the greatest fixed-point of $T_{Hf,\mathcal{D}}$. Equivalently, the semantics of Hf can be expressed by translating Hf into a formula Hf' where constraints are explicitly introduced in the clauses of Hf, and then by considering the greatest fixed-point of $T^{CLP}_{Hf',\mathcal{D}}$, where $T^{CLP}_{Hf',\mathcal{D}}$ is the standard one-step consequence function defined for CLP [13]:

$$T^{CLP}_{Hf,\mathcal{D}}(S) = \{A \mid A \leftarrow C, A_1, \ldots, A_n \text{ ground instance of a clause in } Hf, \\ A_i \in S \text{ for all } i = 1, \ldots, n,\ \mathcal{D} \models C\}$$

A clause having general shape $p_{\overline{\alpha}^n}(\overline{t}^n) \leftarrow \overline{A}^k$ is translated in the CLP clause $p_{\overline{\alpha}^n}(\overline{X}^n) \leftarrow gen(\overline{t}^n, \overline{\alpha}^n, \overline{X}^n), \overline{A}^k$, where $\overline{X}^n$ are distinct and fresh variables and constraints are generated by the function gen defined as follows:

$$gen(\epsilon, \epsilon, \epsilon) = \emptyset \\ gen((t, \overline{t}^{n-1}), (\alpha, \overline{\alpha}^{n-1}), (u, \overline{u}^{n-1})) = \{t\, \alpha\, u\} \cup gen(\overline{t}^{n-1}, \overline{\alpha}^{n-1}, \overline{u}^{n-1})$$

$$\text{(empty)}\quad Hf \mid H \vdash \mathit{true} \rightsquigarrow \emptyset$$

$$\text{(co-hyp)}\frac{Hf \mid H_1, p_{\overline{\alpha}^n}(\overline{t}^n), H_2 \vdash G_1, G_2 \rightsquigarrow C_1 \qquad \vdash C_1 \cup C_2 \rightarrow C'}{Hf \mid H_1, p_{\overline{\alpha}^n}(\overline{t}^n), H_2 \vdash G_1, p_{\overline{\alpha}^n}(\overline{u}^n), G_2 \rightsquigarrow C'}\ C_2 = gen(\overline{t}^n, \overline{\alpha}^n, \overline{u}^n)$$

$$\text{(cls)}\frac{Hf \mid H, p_{\overline{\alpha}^n}(\overline{t}^n) \vdash G_1, G, G_2 \rightsquigarrow C_1 \qquad \vdash C_1 \cup C_2 \rightarrow C'}{Hf \mid H \vdash G_1, p_{\overline{\alpha}^n}(\overline{u}^n), G_2 \rightsquigarrow C'}\ \begin{array}{l} p_{\overline{\alpha}^n}(\overline{t}^n) \leftarrow G \text{ fresh instance of} \\ \text{a clause of } Hf \\ C_2 = gen(\overline{t}^n, \overline{\alpha}^n, \overline{u}^n) \end{array}$$

Fig. 7. Operational semantics

The function *gen* simply takes three tuples of the same length n, $t_1, \ldots, t_n$, $\alpha_1, \ldots, \alpha_n$, and $u_1, \ldots, u_n$, and generates the set of constraints $\{t_1\alpha_1 u_1, \ldots, t_n\alpha_n u_n\}$. This function is used in the next section for expressing the operational semantics of a Horn formula, where the meta-variables u_i may be instantiated with general terms and not only with variables.

Operational semantics The operational semantics of a Horn formula *Hf* is inductively defined in Fig. 7. When restricting to regular terms and proofs, results on the equivalence between the fixed-point and the operational semantics, which holds for both coinductive LP [19] and inductive CLP [13], can be adapted also to the case of coinductive CLP.

In the judgment $Hf \mid H \vdash G \rightsquigarrow C$, meta-variables Hf, H, and G represent the input of the judgment, whereas C is the only output; if the judgment is derivable, then the goal G succeeds w.r.t. the Horn formula *Hf* and the coinductive hypotheses H, with the satisfiable set of constraints C as solution.

The coinductive hypotheses H (a stack of atoms) are needed for building regular derivations; for doing that, we have to keep track of all atoms that have been already resolved with a standard SLD step (see rule (cls) below).

The rules are parametric in the judgment $\vdash C \rightarrow C'$, which corresponds to the abstract specification of the constraint solver for the specific type domain under consideration, hence if the judgment is derivable then $\mathcal{D} \models C$ holds (hence, C represents the input of the solver) and returns an equivalent but simplified version C' (which, therefore, represents the output of the solver).

Rule (empty) deals with the empty goal (represented by *true*) which always succeeds; in this case the returned solution is the empty set of constraints.

Coinduction is managed by rule (co-hyp), where the atom $p_{\overline{\alpha}^n}(\overline{u}^n)$ (non deterministically selected from the goal) is resolved by using a coinductive hypothesis (non deterministically selected from H). This happens when H contains an atom $p_{\overline{\alpha}^n}(\overline{t}^n)$ (that is, with the same predicate symbol p and arity n of the atom selected from the goal) subsuming the atom $p_{\overline{\alpha}^n}(\overline{u}^n)$ of the goal for a certain assignment of values to variables. Such an assignment is determined by the set of constraints C_2 generated by $gen(\overline{t}^n, \overline{\alpha}^n, \overline{u}^n)$ and the set of constraints C_1 corresponding to the solution of the remaining atoms G_1, G_2 of the goal. Hence, if $C_1 \cup C_2$ is satisfiable, then the rule is applicable. The returned solution is the

simplification C' of $C_1 \cup C_2$ computed by the solver. Note that the rule uses subsumption instead of simple term unification, thanks to variance annotations. This would not be possible in standard CLP where constraints are associated with clauses and not with predicates.

Rule (cls) non deterministically selects an atom $p_{\overline{\alpha}^n}(\overline{u}^n)$ from the goal, and a clause from Hf s.t. its head has the same predicate symbol p and arity n of the atom selected from the goal. Then, an instance $p_{\overline{\alpha}^n}(\overline{t}^n) \leftarrow G$ of the clause where all variables are bijectively renamed with fresh variables is considered, and the new goal G_1, G, G_2, obtained by replacing the atom $p_{\overline{\alpha}^n}(\overline{u}^n)$ with the body G of the clause, is resolved w.r.t. the coinductive hypotheses augmented with the head $p_{\overline{\alpha}^n}(\overline{t}^n)$ of the clause. If resolution of G_1, G, G_2 succeeds with constraints C_1, and C_2 is the set of constraints generated from the head of the clause and the atom selected from the goal, then the solver checks whether $C_1 \cup C_2$ is satisfiable. If it so, then the clause is applicable, and resolution of the initial goal succeeds with the constraint set C' obtained by simplifying $C_1 \cup C_2$.

Prototype implementation We have implemented the operational semantics defined in Fig. 7 with a meta-interpreter[12] written in SWI Prolog.

The implementation performs a depth first search of the tree of all possible derivations, by selecting the atoms of the goal and the applicable clauses in the usual order (left to right and top to bottom, respectively). Furthermore, rule (co-hyp) takes the precedence over (cls), and coinductive hypotheses are selected starting from the top of the stack (that is, the most recent coinductive hypothesis is selected first). The basic structure of the meta-interpreter can be specified by the following pseudo-code.

```
coCLP(Goal, Solver, Solution) ←
  coCLP(Goal, Solver, [], [], Solution).
% (empty)
coCLP(true, _Solver, _CoHyp, Solution, Solution).
% (co-hyp)
coCLP((p_ᾱⁿ(ūⁿ), Goal), Solver, CoHyp, C1, Solution) ←
  fresh_atom(p, n, p_ᾱⁿ(X̄ⁿ)),member(p_ᾱⁿ(X̄ⁿ), CoHyp),
  gen(X̄ⁿ, ᾱⁿ, ūⁿ, C2), union(C1, C2, C3), call(Solver, C3, C4),
  coCLP(Goal, Solver, CoHyp, C4, Solution).
% (cls)
coCLP((p_ᾱⁿ(ūⁿ), Goal), Solver, CoHyp, C1, Solution) ←
  fresh_atom(p, n, p_ᾱⁿ(X̄ⁿ)), clause(p_ᾱⁿ(X̄ⁿ), Body),
  gen(X̄ⁿ, ᾱⁿ, ūⁿ, C2), union(C1, C2, C3), call(Solver, C3, C4),
  append_goal(Body, Goal, NewGoal),
  coCLP(NewGoal, Solver, [p_ᾱⁿ(X̄ⁿ)|CoHyp], C4, Solution).
```

The main predicate[13] `coCLP/3` (not specified in Fig. 7) is defined in terms of the auxiliary predicate `coCLP/5` which implements the judgment $Hf \mid H \vdash G \rightsquigarrow C$. The definition is parametric in the predicate corresponding to the constraint

[12] Available at `ftp://ftp.disi.unige.it/person/AnconaD/coCLP.zip`.
[13] We assume that the goal is always terminated by *true*.

solver, which is represented by the variable `Solver`. The two additional arguments of `coCLP/5` (when compared with `coCLP/3`) are the coinductive hypotheses and the accumulated constraints, which are both initially empty. The use of an accumulator for the generated constraints allows a more efficient implementation: `coCLP/5` is tail-recursive, hence its execution can be optimized; furthermore, the constraints generated from the application of a coinductive hypothesis or of a clause are checked before proceeding with the resolution of the remaining atoms of the goal.

The search of an applicable coinductive hypothesis is performed by first creating an atom with the same predicate symbol and arity of the atom selected from the goal, where all arguments are fresh distinct variables (predicate `fresh_atom`, directly implementable with the standard meta-predicate **functor**), then such atom is searched in the list of coinductive hypotheses with the standard `member` predicate. Predicate `gen` corresponds to the function *gen* defined at the beginning of this section, whereas `union` performs union of sets of constraints.

The implementation of rule (cls) (last clause) is analogous except for two details: the standard meta-predicate **clause** is used to find applicable clauses in the program, and the auxiliary predicate `append_goal` is used for appending the body of the selected clause to the remaining part of the initial goal.

Finally, we outline some of the details of our implementation not shown in the pseudo-code defined above.

To associate the variance annotation $\overline{\alpha}^n$ with the predicate p/n, one has to call the goal `register_coind_atom(`$p(\overline{\alpha}^n)$`)`. As a side effect of this call, every clause of p/n is associated with the set of constraints $gen(\overline{t}^n, \overline{\alpha}^n, \overline{X}^n)$, where $p(\overline{t}^n)$ is the head of the clause, and $\overline{X}^n$ are distinct and fresh variables. Hence, the meta-interpreter initially performs the same translation to an equivalent CLP program as defined at the beginning of this section. The set of pre-generated constraints is associated with the clause by means of a dynamically asserted fact containing the reference indexing the clause; such a reference can be retrieved with the library predicate **clause**`/3`.

Pre-generated constraints of clauses are exploited in two different ways: they are used for dynamically generating the set of constraints which must verified for allowing application of a clause, and for dynamically pre-generating the constraints corresponding to the variance annotation of the predicate of the coinductive hypothesis, which is pushed onto the stack when the clause turns out to be applicable. Such pre-generated constraints are used for dynamically generating the set of constraints which must verified for allowing the application of a coinductive hypothesis.

Finally, the answer returned by the interpreter is the set of computed constraints restricted to the variables contained in the initial goal; this final step has been not included in the pseudo-code described above.

The definition of `gen` change into `gen(Ref,` $\overline{X}^n$`,` $\overline{u}^n$`,` C_2`)` that for each $\alpha(t_n, Y)$ in $\overline{C}^n$ unify X_n with t_n and u_n with Y; we need to unify X_n to keep correlation between all constraints. This approach enforce the relation with CLP(X) because we associate the constraints directly to a clause.

We take advantage of the pre-generated constraints also when we manage hypotheses and in a similar way. An hypothesis is an atom with this form `[Functor/Arity|`$\overline{C}^n$`]`, where $\overline{C}^n$ is taken from `constrs`. When we have to add an hypothesis (in (cls)) we unify X_n with t_n; instead, in (co-hypo) we must unify u_n with Y before check the constraints. Note that $\overline{C}^n$ which we put in hypothesis is a completely fresh one, different from the set the we use to verify the constraint of the current atom.

6 Conclusion

This paper provides a further step towards applicability of abstract compilation to realistic object-oriented languages, in two directions.

We have defined a formal abstract compilation scheme allowing precise prediction of uncaught exceptions for a simple Java-like language. The analysis covers both user declared checked exceptions, and the unchecked predefined runtime exception `ClassCastException`. Furthermore, we have presented a general implementation schema for abstract compilation based on coinductive CLP with variance annotation of user-defined predicates, and proposed an implementation based on a Prolog prototype meta-interpreter, parametric in the solver for the subtyping constraints.

Our approach seems particularly promising in the context of object-oriented programming, when the type domain contains union and object types. More efforts are required to obtain results for realistic object-oriented languages. Devising a constraint solver for subtyping on regular union and object types is of paramount importance. We have already investigated several sound but not complete axiomatizations of subtyping [2, 3], but we still do not know whether subtyping on regular union and object types is decidable; currently, we are developing a CHR [9] based implementation of a sound but not complete constraint solver for the abstract compilation scheme presented in this paper.

Although scalability of the approach in the presence of imperative features has been already investigated [4], much work should be accomplished in this direction; for instance, it would be interesting to investigate whether abstract compilation could be integrated with other kinds of analysis to detect reference aliasing, or other runtime exceptions as `NullPointerException` or `IndexOutOfBoundsException`.

References

1. D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In S. Drossopoulou, editor, *ECOOP 2009*, volume 5653 of *LNCS*, pages 2–26. Springer, 2009. Best paper prize.
2. D. Ancona and G. Lagorio. Coinductive subtyping for abstract compilation of object-oriented languages into Horn formulas. In *GandALF 2010*, Electronic Proceedings in Theoretical Computer Science, 2010.

3. D. Ancona and G. Lagorio. Complete coinductive subtyping for abstract compilation of object-oriented languages. In *12th Intl. Workshop on Formal Techniques for Java-like Programs*, ACM Digital Library, 2010.
4. D. Ancona and G. Lagorio. Idealized coinductive type systems for imperative object-oriented programs. Technical report, DISI, January 2010. Submitted for journal publication.
5. D. Ancona, G. Lagorio, and E. Zucca. Type inference by coinductive logic programming. In *Post-Proceedings of TYPES'08*, number 5497 in LNCS. Springer, 2009.
6. F. Barbanera, M. Dezani-Cincaglini, and U. de'Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.
7. B. Courcelle. Fundamental properties of infinite trees. *TCS*, 25:95–169, 1983.
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. *ACM TOPLAS*, 13:451–490, 1991.
9. Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.
10. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java language specification*. The Java series. Addison-Wesley, third edition, 2005.
11. A. Igarashi and H. Nagira. Union types for object-oriented programming. *Journ. of Object Technology*, 6(2):47–68, 2007.
12. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
13. J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.
14. J. Jo, B. Chang, K. Yi, and K. Choe. An uncaught exception analysis for Java. *Journal of Systems and Software*, 72(1):59–69, 2004.
15. R. Min and G. Gupta. Coinductive logic programming and its application to boolean sat. In *FLAIRS Conference*, 2009.
16. N.Saeedloei and G. Gupta. Verifying complex continuous real-time systems with coinductive CLP(R). In *Proc. of LATA 2010*, LNCS. Springer, 2010.
17. M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.
18. L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP 2007*, pages 472–483, 2007.
19. L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In *ICLP 2006*, pages 330–345, 2006.
20. M. Sulzmann and P. J. Stuckey. HM(X) type inference is CLP(X) solving. *Journ. of Functional Programming*, 18(2):251–283, 2008.