

Structural Resolution for Abstract Compilation of Object-Oriented Languages

Luca Franceschini

University of Genoa
Italy

luca.franceschini@dibris.unige.it

Davide Ancona

davide.ancona@unige.it

Ekaterina Komendantskaya

Heriot-Watt University
Edinburgh, Scotland, UK

e.komendantskaya@hw.ac.uk

We propose *abstract compilation* for precise static type analysis of object-oriented languages based on *coinductive logic programming*. Source code is translated to a logic program, then type-checking and inference problems amount to queries to be solved with respect to the resulting logic program. We exploit a coinductive semantics to deal with *infinite terms and proofs* produced by recursive types and methods. Thanks to the recent notion of *structural resolution* for coinductive logic programming, we are able to infer very precise type information, including a class of irrational recursive types causing non-termination for previously considered coinductive semantics. We also show how to transform logic programs to make them satisfy the preconditions for the operational semantics of structural resolution, and we prove this step does not affect the semantics of the logic program.

1 Introduction

Object-oriented programming is the most widespread computational paradigm in programming languages¹. Statically typed languages like Java, C#, and C++ are heavily employed for large scale software development in several domains; however, for implementing applications based on Web or scientific programming, where often less skilled developers are involved, dynamically typed object-oriented languages like JavaScript and Python have gained more popularity for their gentler learning curve, flexibility and ease of use.

Although such languages favour rapid development, and dynamic software adaptation, they also lack the benefits of a static type system: since all errors are detected only at runtime, programs are less reliable, and debugging and testing are more challenging; furthermore, the absence of type information is a severe obstacle to more efficient implementations, and more effective IDE tools.

Static typing needs more effort and knowledge from the programmer, who is burdened by the task of annotating code with types; this load is bearable for simple nominal type systems, but when more accurate type analysis is required, one has to resort to structural type systems, which are more verbose, complex, and, thus, less intuitive to grasp; for instance, Java wildcards are a form of structural type, though integrated with the nominal type system, which cannot be tamed so simply by ordinary programmers.

From the considerations above we can draw the conclusion that there exists a fundamental trade-off between the benefits of static typing, and those of dynamic typing. In order to reduce this “gap”, type inference, and, more in general, any static type analysis which does not require type annotations, is a viable solution; programmers are relieved from declaring types, and still have all benefits of a dynamic language, but also an effective analysis tool to develop more reliable, and maintainable software. Unfortunately, depending on the language in use and on the expressive power of the type system, type analysis of a dynamic language can become quite hard (or even impossible, i.e., undecidable) to solve.

¹See for instance Tiobe index at <http://www.tiobe.com/tiobe-index/>.

Consider the following program implementing linked lists, written in a hypothetical dynamic object-oriented language; for simplicity we adopt a Java-like syntax, but the program contains no type annotations.

```

class EList extends Object {
  EList() {
    super();
  }
  addLast(elem) {
    new NEList(elem, this)
  }
}

class NEList extends Object {
  head;
  tail;
  NEList(head, tail) {
    super();
    this.head = head;
    this.tail = tail;
  }
  addLast(elem) {
    new NEList(this.head, this.tail.addLast(elem))
  }
}

```

Listing 1: Untyped linked lists. `EList.addLast` simply creates a new list with (only) the given element, while `NEList.addLast` recursively reach the end of the list to add the element, and returns the new list.

Depending on the expressive power of the underlying type system, a static analysis tool could be able or not to successfully analyze the expression

```
new EList().addLast(42).addLast(false).head
```

and compute its expected type `int`; however, in a dynamically typed language a tool rejecting such an expression would not be considered very useful, since it should be quite natural for a dynamic language to allow manipulation of lists of heterogeneous elements. Therefore, dynamic languages call for more precise type analysis able to support both *parametric* and *data polymorphism*; the former is the ability to pass arguments of unrelated types to the same parameter, the latter allows assignment of values of unrelated types to the same class field. Correct type analysis of the expression above requires parametric polymorphism because the same method `addLast` of class `NEList` is invoked twice with the first argument of type `int` and `boolean`, respectively, but also data polymorphism is needed, because the two invocations of `addLast` assign to the field `head` values of type `int` and `boolean`, respectively.

Supporting parametric and data polymorphism requires the use of advanced structural types, and ensuring termination of the analysis in presence of recursive types and methods can be challenging. In this paper we investigate an improvement of *abstract compilation* [5] to get more precise type analysis of object-oriented code involving recursive method invocations.

Abstract compilation is a modular approach to type analysis that exploits logic programming; programs under analysis are abstractly compiled to logic programs, and then analysis is performed through goal resolution.

For instance, in order to infer the type of the last expression about list classes we considered, a goal clause similar to the following one could be used:

$$new(elist, [], E), invoke(E, addLast, [int], R), invoke(R, addLast, [int], R')$$

The three atoms encode the calls to the constructor and to the `addLast` method (twice), respectively. Variables E, R, R' are the types resulting from the operation. After formulating the goal query, it has to be resolved with respect to the logic program generated by abstract compilation of the original one, as it will be shown in Section 3. Finally, the computed substitutions will give terms encoding types, thus effectively solving the inference problem.

To support parametric and data polymorphism in the presence of recursion, the resolution method has to support the coinductive interpretation of the generated logic program [22] based on the greatest complete Herbrand model. However, implementation of the operational semantics of coinductive logic programming (Co-LP for short) fails to successfully analyze some kinds of recursion.

In this paper we show how this drawback can be overcome by adopting *structural resolution* [17] for the inference engine used for abstract compilation; thanks to the notion of *productivity*, structural resolution allows an operational semantics which is more expressive than Co-LP (under certain assumptions that will be discussed).

The rest of the paper is organized as follows. Section 2 introduces the necessary background on coinductive logic programming and presents structural resolution, while Section 3 is a detailed introduction to abstract compilation. Section 4 motivates the usefulness of structural resolution for abstract compilation and shows how it can improve previous results, which is the main contribution of this work. Section 5 is devoted to the notion of productivity, and shows a transformation technique that guarantees productivity for abstract compilation. Section 6 contains some concluding remarks and future work directions.

2 (Coinductive) Logic Programming

2.1 Logic Programming Preliminaries

Given a first-order signature consisting of variables, function and predicate symbols, we define terms inductively as is standard: they can be either *variables* or *function symbols* of arity n applied to n terms $(f(t_1, \dots, t_n))$. *Constants* are function symbols of arity 0. *Atoms* or atomic formulas have the shape $p(t_1, \dots, t_n)$, where p is a *predicate symbol* of arity n and t_1, \dots, t_n are terms. *Logic programs* are finite sets of *definite Horn clauses* (clauses for short) $A \leftarrow B_1 \wedge \dots \wedge B_n$, where A, B_1, \dots, B_n are atoms. A is called the *head* of the clause and $B_1 \wedge \dots \wedge B_n$ is called the *body* of the clause. When the body is empty the head is considered to be *true*. A *goal clause* has the shape $A_1 \wedge \dots \wedge A_n$.

A *substitution* is a finite (partial) mapping from variables to terms, where all variables are simultaneously substituted. Two terms t_1 and t_2 are *unifiable* by substitution σ if $\sigma(t_1) = \sigma(t_2)$; t_1 *matches* t_2 by σ if $\sigma(t_1) = t_2$. If in these two cases, σ is additionally the most general substitution, we say it is the *most general unifier (mgu)* or *most general matcher (mgm)* respectively. Terms are said to be *ground* if they contain no variables. Given a term t , a substitution σ is *grounding for t* if $\sigma(t)$ is ground. All these definitions can be extended to atoms and clauses in the standard way.

Traditionally, the inductive semantics of logic programs is given by Herbrand models. The *Herbrand base* B_P of a logic program P is the set of all ground atoms built from function and predicate symbols in P . The *least Herbrand model* M_P is the smallest subset of B_P that is also a model for each clause in P . An atom A is *logically entailed* from P if $\sigma(A) \in M_P$, for some grounding substitution σ for A .

Given a logic program P and a goal clause $A_1 \wedge \dots \wedge A_n$, *SLD resolution* is a semi-decision procedure to check whether A_1, \dots, A_n are logically entailed from P and, if so, to compute a substitution σ such that $\sigma'(\sigma(A_i)) \in M_P$ for every A_i in the goal, and for all substitutions σ' grounding for $\sigma(A_1 \wedge \dots \wedge A_n)$. Thus, goal clauses can be seen as queries to be solved with respect to a logic program, and the computed substitution encodes the answer (if any).

Definition 1 (SLD-resolution reduction). If P is a logic program and $A_1 \wedge \dots \wedge A_n$ is a goal clause, the *SLD-resolution reduction* (with respect to P) is given by $(A_1 \wedge \dots \wedge A_n) \rightsquigarrow_P (\sigma(A_1) \wedge \dots \wedge \sigma(A_{i-1}) \wedge \sigma(B_0) \wedge \dots \wedge \sigma(B_m) \wedge \sigma(A_{i+1}) \wedge \dots \wedge \sigma(A_n))$ if $A \leftarrow B_0 \wedge \dots \wedge B_m \in P$ and σ is the mgu of A_i and A .

A more constructive definition of the least Herbrand models can be given in terms of *fixed points* of a suitable function. Given a logic program P , the *immediate consequence operator* $T_P : \mathcal{P}(B_P) \rightarrow \mathcal{P}(B_P)$ is

a function defined on the powerset of the Herbrand base as follows:

$$T_P(S) = \{A \mid A \leftarrow B_1 \wedge \dots \wedge B_n \text{ ground instance of a clause in } P, \{B_1, \dots, B_n\} \subseteq S\}$$

Since T_P is monotonic for any logic program P , by the Knaster-Tarski theorem it has a *least fixed point*, which is precisely the least Herbrand model M_P .

For the rest of the paper, we use the following syntactic conventions: function and predicate symbols start with a lowercase letter, and constants are sometimes numbers; variables start with an uppercase letter; atoms, clauses and logic programs are written as single uppercase letters.

2.2 Coinduction in Logic Programming

In inductive logic programming, only terminating SLD derivations are meaningful. However, there are logic programs for which there are no terminating derivations and there is no natural inductive semantics, yet they can still be understood *coinductively*.

Example 2. The following logic program P_{zeros} defines the infinite list of zeros:

$$zeros(cons(0, X)) \leftarrow zeros(X)$$

Starting from the goal $zeros(X)$, SLD resolution does not terminate. Indeed, the program above has no inductive meaning, and its least Herbrand model is empty. Still, its clause has a clear meaning.

Recall that in the inductive interpretation, models contain only finite terms. Coinductive interpretation admits both finite *and infinite* terms. Given a logic program P , B_P^{co} is the *complete Herbrand base* containing all finite and infinite atoms built on the top of function and predicate symbols in P . For the coinductive interpretation, the *greatest complete Herbrand model* M_P^{co} is considered, that is, the greatest subset of B_P^{co} that is also a model for P . In example 2, $M_P = \emptyset$ but $M_P^{co} = \{zeros(cons(0, cons(0, \dots)))\}$.

The duality between the inductive and the coinductive interpretation extends to the fixed point semantics: inductive models are the least fixed point of the immediate consequence operator while coinductive models are the *greatest* fixed point of the operator (extended to possibly infinite terms). The existence of the greatest fixed point is again ensured by the Knaster-Tarski theorem.

In the 80s, the notion of formulas *computable at infinity* was introduced [21]. An infinite formula A is computable at infinity, if there exists a finite formula A' such that A' has an infinite (and fair) SLD-derivation, and substitutions $\sigma_0, \sigma_1, \dots$ computed in the course of this derivation yield $\sigma_0(\sigma_1(\dots(A')\dots)) = A$. For example, $zeros(cons(0, cons(0, \dots)))$ is computable at infinity for the program P_{zeros} and the query $zeros(X)$. In such cases we also say that the infinite SLD-derivation for $zeros(X)$ is *globally productive*, in a sense of producing an infinite term as a substitution.

Operationally, dealing with infinite terms and non-terminating derivations is a challenge. *Co-LP* [22] extends SLD resolution with a cycle detection mechanism that allows the derivation to be concluded when a goal unifies with a previously encountered one. Considering again Example 2, the reduction $zeros(X) \rightsquigarrow_{P_{zeros}} zeros(X')$ holds with the computed substitution $\{X \mapsto cons(0, X')\}$. At this point, Co-LP checks if the two goals unify, and indeed they do: the computed answer is $\{X \mapsto cons(0, X)\}$ ² which corresponds to the infinite term specified by the recursive equation $X = cons(0, X)$, that is $cons(0, cons(0, \dots))$.

Note that the recursive term $cons(0, cons(0, \dots))$ is *regular* [9] (a.k.a rational or cyclic) since it has a finite number of subterms, namely 0 and itself. Because Co-LP's algorithm relies on unification of the looping coinductive goals, it can only handle regular terms and derivations. As a result, it does not terminate on irregular derivations, thus it is only sound but not complete w.r.t the greatest complete Herbrand model.

²In the coinductive setting the occurs check needs to be removed.

2.3 Structural Resolution

Structural resolution [19, 16, 11] (or S-resolution for short) proposes a solution for cases when formulas computable at infinity are not regular. Consider the following example.

Example 3. The coinductive program P_{from} below has the following single clause:

$$from(X, scon(s(X), Y)) \leftarrow from(s(X), Y)$$

Given the query $from(0, X)$, and writing $[-|_]$ as an abbreviation for the stream constructor $scons$, here we have that the infinite atom $a = from(0, [0|[s(0)|[s(s(0))|...]])$ is computable at infinity by P_{from} and it is also contained in the greatest complete Herbrand model of P_{from} . Coinductive reasoning on this query cannot be handled by the loop detection mechanism of Co-LP because the atom a is irrational and the looping subgoals will fail to unify.

In such cases, it may still be possible to automatically prove that the SLD-derivation for the query $from(0, X)$ will be infinite, non-failing, and moreover will compute an infinite term at infinity, even if we cannot generate its closed form, as for P_{zeros} . In the core of this new argument is the detection of a regular pattern – a constructor – that works as a building block of the infinite term computed at infinity; in P_{from} this constructor is $scons$. We now explain the method that detects such patterns in S-resolution.

S-resolution [19, 16, 11] stratifies the SLD-derivation steps into those done by term-matching and those requiring full unification. Term-matching in this case plays a role that pattern-matching on constructors of data structures plays in functional programming.

Definition 2. [13] If P is a logic program and $A_1 \wedge \dots \wedge A_n$ is a goal clause, then:

- *rewriting reduction*: $(A_1 \wedge \dots \wedge A_i \wedge \dots \wedge A_n) \rightarrow_P (A_1 \wedge \dots \wedge A_{i-1} \wedge \sigma(B_0) \wedge \dots \wedge \sigma(B_m) \wedge A_{i+1} \wedge \dots \wedge A_n)$ if $A \leftarrow B_0 \wedge \dots \wedge B_m \in P$ and σ is the mgm for A against A_i ($\sigma(A) = A_i$);
- *substitution reduction*: $(A_1 \wedge \dots \wedge A_i \wedge \dots \wedge A_n) \hookrightarrow_P (\sigma(A_1) \wedge \dots \wedge \sigma(A_i) \wedge \dots \wedge \sigma(A_n))$ if $A \leftarrow B_0 \wedge \dots \wedge B_m \in P$ and A_i and A are unifiable via mgu σ .

The *S-resolution reduction* with respect to P is $\hookrightarrow_P \circ \rightarrow_P^\mu$. We write $A_1 \wedge \dots \wedge A_n \rightarrow_P^\mu$ to indicate the reduction of $A_1 \wedge \dots \wedge A_n$ to its \rightarrow -normal form with respect to P if this normal form exists, and to indicate an infinite reduction of $A_1 \wedge \dots \wedge A_n$ with respect to P otherwise.

One can show that under certain conditions, SLD-resolution reductions and structural resolution reductions are equivalent, see [13, 17]; but S-resolution has one advantage: it helps to detect the constructors from which the infinite data structure is built.

Firstly, we represent the structural resolution reductions as tree rewriting: the figure below shows how rewriting reduction steps can be represented as *rewriting trees* and substitution reduction steps shown horizontally as rewriting tree transitions. This separation makes it easy to see that in this derivation, the same pattern $[-|_]$ gets consumed by rewriting steps and gets added, or produced, in the substitution steps:

$$\begin{array}{ccc}
 \{X \mapsto [0|X']\} & \{X' \mapsto [s(0)|X'']\} & \{X'' \mapsto [s(s(0))|X''']\} \\
 \hookrightarrow & \hookrightarrow & \hookrightarrow \\
 from(0, X) & from(0, [0|X']) & from(0, [0|[s(0)|X'']]) \\
 & from(s(0), X') & from(s(0), [s(0)|X'']) \\
 & & from(s(s(0)), X''')
 \end{array}$$

We can only detect this pattern if rewriting trees are finite, i.e. all rewriting reductions are normalising. By definition, *observational productivity* of an S-resolution reduction for a program P and a query A is in fact a conjunction of two properties [11]:

- *universal observability*: normalisation of all rewriting reductions in this S-resolution reduction, and
- *existential liveness*: non-termination of this S-resolution reduction.

S-resolution terminates when these two properties are satisfied. Otherwise, S-resolution generates infinite derivations lazily, showing only partial answers. For example, S-resolution will terminate lazily after the three substitution steps shown above, and will output the partial answer: $X = [0|[s(0)|[s(s(0))|X''']]$.

One can show that observational productivity implies global productivity [17, 11]. Note that universal observability is thus a formal pre-condition for reasoning about observational productivity of S-resolution. Not every non-terminating program is globally and observationally productive.

Example 4. Consider the following logic program P :

$$\begin{aligned} p(f(X)) &\leftarrow p(X) \\ q(X) &\leftarrow q(X) \end{aligned}$$

The two goal clauses $p(X)$ and $q(X)$ lead to the following non-terminating SLD derivations, respectively:

$$\begin{array}{ccccccc} p(X) & \rightsquigarrow_P^{X=f(X')} & p(X') & \rightsquigarrow_P^{X'=f(X'')} & p(X'') & \rightsquigarrow_P^{X''=f(X''')} & \dots \\ q(X) & \rightsquigarrow_P & q(X) & \rightsquigarrow_P & q(X) & \rightsquigarrow_P & \dots \end{array}$$

In the first derivation, each derivation step gives a better approximation of the rational term $f(f(\dots))$ by incrementally instantiating free variables X, X', X'', \dots . In the second one, the goal never changes, and there is no “real” progress. Nevertheless, both the rational atoms $p(f(f(\dots)))$ and $q(f(f(\dots)))$ belong to the greatest complete Herbrand model of the program above.

Following the idea of *computations at infinity* [12, 21], only the first derivation actually computes an infinite term after an infinite number of steps and only the first derivation is globally productive. To see what happens with observational productivity, note that the clause $q(X) \leftarrow q(X)$ makes the program break this requirement: $q(X) \rightarrow q(X) \rightarrow \dots$. On the other hand, S-resolution reductions for $p(X)$ will be productive (again note that the constructor f gets added in substitution reductions and consumed in rewriting reductions):

$$p(X) \hookrightarrow^{X=f(X')} p(f(X')) \rightarrow p(X') \hookrightarrow^{X'=f(X'')} p(f(X'')) \rightarrow p(X'') \dots$$

Thus, S-resolution will work for queries on $p(X)$ but not $q(X)$. For the above derivation for $p(X)$, it will detect that f plays a role of an infinite data structure constructor.

3 Abstract Compilation

Abstract compilation [5, 8] is a technique developed in the context of object-oriented programming to exploit the potentialities of logic programming for supporting advanced static type analysis, as investigated also by other authors [23, 1].

In a nutshell, the approach consists in translating the program under analysis into a logic program which abstracts the semantics of the source program; then, performing static type analysis on the program amounts to solving a goal w.r.t. the obtained logic program.

Abstraction is mainly obtained by structural types which represent set of values, following the semantic subtyping approach [2, 4]; boolean type constructors, as union types, and record types allow quite precise analysis if employed in conjunction with abstract compilation; in particular, both parametric and data

polymorphism are supported. Solving a goal corresponds to symbolically executing the original source program with types representing set of values.

Abstract compilation strives to reconcile compositional and global analysis, because once the program under analysis has been abstractly compiled, its source code is no longer needed, as long as it remains unmodified, and classes can be abstractly compiled separately. Since analysis corresponds to goal solving, it can only be performed when the whole relevant program has been compiled; this limitation is also a feature, because it promotes precise analysis through context sensitive data and parametric polymorphism.

Finally, abstract compilation offers interesting opportunities to fruitfully exploit compiler technology [6, 7, 3] for more precise and efficient analysis.

3.1 Abstract Compilation at Work

Let us consider the two classes implementing linked lists defined in listing 1 and show how they could be translated into a logic program to perform type analysis on them.

The translation depends on the way values are abstracted, that is, the underlying type system. In this particular case we may use the primitive types *null*, *int*, and *bool* to represent the singleton value `null`, and the sets of integer and boolean values, respectively; then, $obj(c, [f_1:\tau_1, \dots, f_n:\tau_n])$ represents the set of all instances created from class *c* having at least fields f_1, \dots, f_n associated with values of types τ_1, \dots, τ_n , respectively. To make the type system more expressive, we also introduce union types, corresponding to logical disjunction: $\tau_1 \vee \tau_2$ represents the set of all values which have type τ_1 or τ_2 .

Types represent sets of values and are the terms manipulated by the generated logic programs; for instance, referring to the classes of listing 1, the type $obj(elist, []) \vee obj(nelist, [head:int, tail:obj(elist, [])])$ represents all objects implementing integer linked lists of length ≤ 1 .

Predicates are introduced for representing the different kinds of declarations and constructs of the source language. For instance, predicates *new* and *invoke* abstract object creation, and method invocation, respectively, while predicate *hasmeth* represents method declarations. Consequently, the atom $new(nelist, [int, null], obj(nelist, [head:int, tail:null]))$ formally states that the invocation of the constructor of class `NEList` with arguments of type *int* and *null*, respectively, returns a value of type $obj(nelist, [head:int, tail:null])$. As another example,

$$invoke(obj(elist, []), addlast, [int], obj(nelist, [head:int, tail:obj(elist, [])]))$$

formally states that the invocation of method `addLast` on an object of type $obj(elist, [])$, and argument of type *int*, returns an object of type $obj(nelist, [head:int, tail:obj(elist, [])])$.

There exist two separate kinds of Horn clauses³ that are generated by the translation: those encoding the abstract semantics of the source programming languages, which are independent of any analyzed program, and those which are directly derived from the code under analysis. For instance, the clause

$$invoke(obj(C, F), M, A, R) \leftarrow hasmeth(C, M, [obj(C, F)|A], R)$$

partly specifies⁴ the abstract semantics of method invocation; it states that the invocation of method *M* on an object of type $obj(C, F)$ with arguments of type *A* returns a value of type *R*, if the class *C* of the receiver object has a method *M* returning a value of type *R* when invoked on object `this` of type $obj(C, F)$ with arguments of type *A*. Thus, the semantics of *hasmeth* depends on the code of the declared

³We use Prolog syntactic conventions: variables start with an upper case letter, constants starts with a lower case letter and $[a, b, c, \dots]$ denotes a list. Moreover, we use the infix notation for the binary function symbol \vee .

⁴Two more clauses are needed to deal with union types, and with inherited methods.

methods. Indeed, for each method declaration, a corresponding clause for predicate *hasmeth* is generated; for instance, the following clause is derived from the declaration of method `addLast` in class `EList`:

$$\text{hasmeth}(\text{elist}, \text{addlast}, [\text{This}, \text{Elem}], R) \leftarrow \text{new}(\text{nelist}, [\text{Elem}, \text{This}], R)$$

It states that class `EList` has method `addLast` that, when invoked on the object `this` of type *This* with argument of type *Elem*, returns a value of type *R*, providing that constructor of class `NEList` returns a value of type *R* when invoked on arguments of type *Elem*, and *This*, respectively.

Analogously, the following clause is generated from the declaration of method `addLast` in class `NEList`, where predicate *fieldacc* abstracts the semantics of field access:

$$\begin{aligned} \text{hasmeth}(\text{nelist}, \text{addlast}, [\text{This}, \text{Elem}], R) &\leftarrow \text{fieldacc}(\text{This}, \text{head}, H) \wedge \text{fieldacc}(\text{This}, \text{tail}, T) \\ &\wedge \text{invoke}(T, \text{addlast}, [\text{Elem}], N) \wedge \text{new}(\text{nelist}, [H, N], R) \end{aligned}$$

3.2 Examples of Queries and Recursive Types

We start by showing a simple goal to typecheck expression `new EList().addLast(i)`, under the assumption that *i* has type `int`. This can be achieved by solving the goal

$$\text{new}(\text{elist}, [], R) \wedge \text{invoke}(R, \text{addlast}, [\text{int}], T)$$

which succeeds, as expected, with answer:

$$\begin{aligned} R &= \text{obj}(\text{elist}, []) \\ T &= \text{obj}(\text{nelist}, [\text{head}:\text{int}, \text{tail}:\text{obj}(\text{elist}, [])]) \end{aligned}$$

As a more elaborated example, let us consider the expression `new NEList(b, l).addLast(i)`, under the assumption that *b*, *l*, and *i* have type `bool`, `obj(elist, [])`, and `int`, respectively; typechecking this expression corresponds to solving the goal

$$\text{new}(\text{nelist}, [\text{bool}, \text{obj}(\text{elist}, [])], R) \wedge \text{invoke}(R, \text{addlast}, [\text{int}], T)$$

which succeeds for

$$\begin{aligned} R &= \text{obj}(\text{nelist}, [\text{head}:\text{bool}, \text{tail}:\text{obj}(\text{elist}, [])]) \\ T &= \text{obj}(\text{nelist}, [\text{head}:\text{bool}, \text{tail}:\text{obj}(\text{nelist}, [\text{head}:\text{int}, \text{tail}:\text{obj}(\text{elist}, [])])]). \end{aligned}$$

This example shows that typechecking can succeed also for expressions which build heterogeneous lists.

For a simple example of type inference, let us consider the problem of finding a valid type assignment for variables *x* and *y* to make the expression `x.addLast(y)` well-typed; this corresponds to the goal

$$\text{invoke}(X, \text{addlast}, [Y], T)$$

which, for instance, succeeds for

$$\begin{aligned} X &= \text{obj}(\text{elist}, F) \\ T &= \text{obj}(\text{nelist}, [\text{head}:Y, \text{tail}:\text{obj}(\text{elist}, [F])]) \end{aligned}$$

The fact that the logical variable *Y* is not in the domain of the computed substitution means that any type can be safely assigned to *y*.

In the previous examples we have only considered types specifying linked lists of fixed length, but for building more interesting types, recursion is needed; this is achieved by considering rational terms (a.k.a. regular or cyclic). For instance, the unique term defined by the solution of the unification problem

$$T = \text{obj}(\text{elist}, []) \vee \text{obj}(\text{nelist}, [\text{head:int}, \text{tail:T}])$$

corresponds to the recursive type specifying the set of all integer linked lists of arbitrary length.

All example queries considered so far can be solved w.r.t. the standard inductive interpretation of Horn clauses, that is, the least Herbrand model, even though method `addLast` is recursive in class `NEList`; however, if recursive types are involved in queries, then the least Herbrand model is no longer sufficient to capture their intended meaning, as shown in Section 4.

4 Coinduction and Structural Resolution in Abstract Compilation

4.1 The Need for Coinduction

As already mentioned in the previous section, the intended meaning of goals and logic programs does not always coincide with the least Herbrand model. When recursive types and methods are involved, the coinductive interpretation is needed, i.e., the greatest complete Herbrand model has to be considered [21].

Let us consider the recursive method `replicate` from listing 5: if `n` is not positive, then the method returns an empty list, otherwise it recursively builds a list of `n-1` occurrences of `x`, and then returns the (newly created) list where element `x` has been added at the beginning. We refer to listing 1 for the definitions of the classes `EList` and `NEList`.

```
class ListFact extends Object {
  ListFact() { super(); }
  replicate(n, x) {
    if (n ≤ 0) new EList()
    else new NEList(x, this.replicate(n-1, x))
  }
}
```

Listing 5: Given an integer `n` and an element `x`, `replicate` returns a list containing `n` occurrences of `x`.

By means of abstract compilation, method `replicate` would be translated⁵ to the following clause:

$$\text{hasmeth}(\text{listfact}, \text{replicate}, [\text{This}, \text{int}, \text{X}], E \vee NE) \leftarrow \text{new}(\text{elist}, [], E) \wedge \text{invoke}(\text{This}, \text{replicate}, [\text{int}, \text{X}], R) \wedge \text{new}(\text{nelist}, [\text{X}, \text{R}], NE)$$

The first atom of the body corresponds to the invocation of the constructor of `EList`, while the other two atoms are generated from the recursive invocation, and from the invocation of the constructor of `NEList`, respectively. Finally, the use of the conditional expression is reflected in the term $E \vee NE$. Note that the Horn clause above is the *only* clause generated by abstract compilation for method `replicate`.

Let us now consider the expression `new ListFact().replicate(10, 42)`. To infer the type T of the expression above (w.r.t. the classes in listings 1 and 5), the following goal is generated:

$$\text{new}(\text{listfact}, [], L) \wedge \text{invoke}(L, \text{replicate}, [\text{int}, \text{int}], T)$$

⁵For the sake of readability, we have applied some simplifications to the resulting clause; however, such changes do not affect its semantics.

However, such a goal fails to succeed if the inductive interpretation is considered; indeed, the SLD derivation⁶ is *non-terminating*:

$$\begin{aligned}
& new(listfact, [], L) \wedge invoke(L, replicate, [int, int], T) \rightsquigarrow^* \\
& \quad invoke(obj(listfact, []), replicate, [int, int], T) \rightsquigarrow^* \\
& \quad hasmeth(listfact, replicate, [obj(listfact, []), int, int], T) \rightsquigarrow^* \\
& hasmeth(listfact, replicate, [obj(listfact, []), int, int], obj(elist, []) \vee obj(nelist, [head: int, tail: T'])) \rightsquigarrow \dots
\end{aligned}$$

In the derivation above, every new atom for predicate *hasmeth* yields a better approximation for T , but unfortunately the derivation never terminates. If we interpret *coinductively* the logic program obtained by abstract compilation, thus considering its greatest complete Herbrand model rather than its least one, the goal above is actually entailed by the program with a substitution instantiating T with the following type:

$$obj(elist, []) \vee obj(nelist, [head: int, tail: (obj(elist, []) \vee obj(nelist, [head: int, \dots]))])$$

The type above represents all integer lists of arbitrary length; however, such a type corresponds to an infinite term. Fortunately, there exists an equivalent type corresponding to the *rational* term [9] specified by the following recursive equation:

$$T = obj(elist, []) \vee obj(nelist, [head: int, tail: T])$$

This example shows that goals involving *recursive* types and methods require a coinductive interpretation of the logic program obtained by abstract compilation, in order to make static type analysis more precise.

As already illustrated in Section 2, answer substitutions with rational terms can be computed by extending SLD resolution with cycle-detection techniques [22], as proposed with Co-LP; hence, an inference engine based on Co-LP improves the result of static analysis performed with abstract compilation [5]. The Co-LP inference engine is however limited, since it succeeds with rational terms, and derivations, but it cannot handle more complex scenarios.

4.2 Structural Resolution for Abstract Compilation

Listing 6 shows a slightly more involved example. Suppose we add to `ListFact` a method `buildList` that, given an integer n , builds the list of integers $1, 2, \dots, n$ with an auxiliary method `buildList` which exploits tail recursion with an accumulator parameter `acc`; a more realistic Java implementation would of course avoid recursion, and use instead a simpler and more efficient loop, for which static type analysis for abstract compilation is less problematic if one exploits SSA intermediate form [6, 7, 3] during the compilation phase; however, in the past years object-oriented languages have begun to exploit more and more patterns based on functional style programming, possibly with recursion and accumulators.

Abstract compilation of `buildList` would yield⁷ the following Horn clause:

$$hasmeth(listfact, buildlist, [This, int, A], A \vee R) \leftarrow new(nelist, [int, A], A') \wedge invoke(This, buildlist, [int, A'], R)$$

Suppose we want to infer the type T_0 of the expression `new List().buildList(42, new EList())`. Such an expression is abstractly compiled to the following goal:

$$new(listfact, [], L) \wedge new(elist, [], A_0) \wedge invoke(L, buildlist, [int, A_0], T_0)$$

⁶Colours enlighten the substitution computed by unification along the way.

⁷Again, for readability we are simplifying the clause that would be automatically generated.

```

buildList(n, acc) {
  if (n ≤ 0) acc
  else this.buildList(n-1, new NEList(n, acc))
}

```

Listing 6: Given an integer n , `buildList` returns the list of integers $1, 2, \dots, n$ followed by `acc`, which is used as an accumulating parameter.

The derivation for such a goal is again infinite, hence the coinductive interpretation is needed again:

$$\begin{aligned}
& new(listfact, [], L) \wedge new(elist, [], A_0) \wedge invoke(L, buildlist, [int, A_0], T_0) \rightsquigarrow^* \\
& new(elist, [], A_0) \wedge invoke(obj(listfact, []), buildlist, [int, A_0], T_0) \rightsquigarrow^* \\
& invoke(obj(listfact, []), buildlist, [int, A_0], T_0) \rightsquigarrow^* \\
& invoke(obj(listfact, []), buildlist, [int, A_1], T_1) \rightsquigarrow^* \\
& invoke(obj(listfact, []), buildlist, [int, A_i], T_i) \rightsquigarrow \dots
\end{aligned}$$

However, as opposed to the previous example, in this case the derivation is *not rational*. Indeed, at each step of the derivation a non-equivalent type is computed both for the accumulator and the returned value, since lists of different lengths have non-equivalent types. The following countably infinite set of equations defines the computed answer substitution associated with the whole derivation:

$$\begin{array}{ll}
A_0 = obj(elist, []) & T_0 = A_0 \vee T_1 \\
A_1 = obj(nelist, [head: int, tail: A_0]) & T_1 = A_1 \vee T_2 \\
\vdots & \vdots \\
A_i = obj(nelist, [head: int, tail: A_{i-1}]) & T_i = A_i \vee T_{i+1} \\
\vdots & \vdots
\end{array}$$

After a closer look at the set above, we can deduce that the considered non rational derivation succeeds because the set of equations above admits a solution, although such a solution involves non rational terms; in particular, the type T_0 of the expression `new List().buildList(42, new EList())` is non-rational; as a consequence, an inference engine based on Co-LP [22] would fail to compute a type, because no cycle can be detected in the derivation.

In order to solve this problem, we propose to use *structural resolution* [17] as inference engine for abstract compilation. This new resolution method relies on a *productivity* notion which is not limited to rational trees, thus it offers the possibility to exploit a more flexible inference engine to allow more expressive static type analysis through abstract compilation.

Starting from the goal above, after a finite number of derivation steps, structural resolution is able to compute the substitution⁸ $T_0 = obj(elist, []) \vee T_1$, effectively solving the task of determining the type of the expression `new List().buildList(42, new EList())`. It works by noticing that the pattern $_ \vee _$ is consumed by the terminating rewriting reductions, and is also infinitely produced in a chain of substitution reductions. Hence, \vee serves as a constructor of the infinite data structure produced at infinity.

⁸Depending on the implementation of structural resolution, the computed answer can be more or less precise, since type T_0 could be “unfolded” more than once before the (first) answer is returned.

The computed answer $T_0 = \text{obj}(\text{elist}, []) \vee T_1$ is only *partial*, since variable T_1 is still “unresolved”. However, structural resolution ensures that the non-rational term corresponding to the computed type T_0 can be incrementally unfolded for an *arbitrary number* of steps: if needed, the substitution for T_1 can be computed in a finite number of derivation steps, thus providing a better approximation of the type associated with T_0 . In this sense, the use of structural resolution in conjunction with abstract compilation gives rise to the implementation of a *lazy* type inference procedure.

5 Ensuring Universal Observability of Coinductive Logic Programming

5.1 Universal Observability and Program Transformation

In the previous section we discussed how structural resolution can be useful to handle (a class of) non-terminating derivations in finite time, when derivations compute an infinite irrational term. However, this new resolution method can be successfully employed only if logic programs are universally observable.

Logic programs resulting from abstract compilation are *not* universally observable in the general case.

Example 7. Consider for instance abstract compilation for an object-oriented language supporting nominal subtyping; the following three clauses should be generated for all source programs:

$$\begin{aligned} \text{subclass}(X, X) &\leftarrow \text{class}(X) \\ \text{subclass}(X, \text{object}) &\leftarrow \text{class}(X) \\ \text{subclass}(X, Z) &\leftarrow \text{extends}(X, Y) \wedge \text{subclass}(Y, Z) \end{aligned}$$

The following infinite derivation shows that some logic programs obtained by abstract compilation are not universally observable:

$$\begin{aligned} \text{subclass}(A, B) &\rightarrow \text{extends}(A, X) \wedge \text{subclass}(X, B) \\ &\rightarrow \text{extends}(A, X) \wedge \text{extends}(X, X') \wedge \text{subclass}(X', B) \\ &\rightarrow \dots \end{aligned}$$

Note that SLD-resolution would be able to find finite (i.e. inductive) proof for this query by unifying with the first or the second clause, whereas S-resolution’s rewriting reductions get caught in an infinite loop by rewriting on the third clause. Thus for this example, S-resolution is also incomplete.

In order to fully exploit S-resolution, we present and formalise a *transformation* of logic programs that ensures universal observability of S-resolution reductions. Additionally, it ensures inductive completeness for inductive fragments of programs and observational productivity for coinductive fragments of programs. An extended version of this transformation has first been presented in [13]. Informally, given a clause $p(\bar{t}) \leftarrow q_1(\bar{t}_1) \wedge \dots \wedge q_n(\bar{t}_n)$, the transformation adds extra argument to all atoms, in such a way that the extra arguments reflect the clause structure: $p(\bar{t}, \kappa(\chi_1, \dots, \chi_n)) \leftarrow q_1(\bar{t}_1, \chi_1) \wedge \dots \wedge q_n(\bar{t}_n, \chi_n)$. Note that the extra term $\kappa(\chi_1, \dots, \chi_n)$ can be intuitively read as: program clause κ has n atoms in its body.

The following definition formalises this translation for arbitrary logic programs. We use Greek letters for those parts of the logic program that are added in the transformation.

Definition 3 (Productivity transformation). Given a logic program P we assume two sets $\{\kappa_1, \kappa_2, \dots\}$ and $\{\chi_1, \chi_2, \dots\}$ of distinct and fresh function symbols and variables, respectively. Then, $\Pi(P)$ is a new logic program defined by the following equations (for programs, clauses and atoms, respectively):

$$\begin{aligned} \Pi(P) &= \Pi(\{C_1, \dots, C_n\}) = \{\Pi_{\kappa_1}(C_1), \dots, \Pi_{\kappa_n}(C_n)\} \\ \Pi_{\kappa_i}(C_i) &= \Pi_{\kappa_i}(A \leftarrow B_1 \wedge \dots \wedge B_n) = \Pi_{\kappa_i(\chi_1, \dots, \chi_n)}(A) \leftarrow \Pi_{\chi_1}(B_1) \wedge \dots \wedge \Pi_{\chi_n}(B_n) \\ \Pi_{\tau}(A) &= \Pi_{\tau}(p(t_1, \dots, t_n)) = p(t_1, \dots, t_n, \tau) \end{aligned}$$

Goal clauses have to be transformed as well in order to be resolved w.r.t. the transformed logic program:

$$\Pi(G) = \Pi(A_1 \wedge \dots \wedge A_n) = \Pi_{\chi_1}(A_1) \wedge \dots \wedge \Pi_{\chi_n}(A_n)$$

Example 8. Consider the modified version of the program from example 7, where κ_i are distinct new function symbols and χ is a new variable:

$$\begin{aligned} subclass(X, X, \kappa_1(\chi_1)) &\leftarrow class(X, \chi_1) \\ subclass(X, object, \kappa_2(\chi_1)) &\leftarrow class(X, \chi_1) \\ subclass(X, Z, \kappa_3(\chi_1, \chi_2)) &\leftarrow extends(X, Y, \chi_1) \wedge subclass(Y, Z, \chi_2) \end{aligned}$$

It can be easily proved that this new version of the program *is* universally observable, and the last argument provides the termination measure for rewriting reductions. In particular, $subclass(A, B, \chi) \not\rightarrow$. Now we can have a one step substitution reduction to $subclass(A, A, \kappa_1(\chi_1))$, and this subgoal will have one step (terminating) rewriting reduction to $class(A, \chi)$, just as with SLD-resolution.

As the above example shows, the program transformation guarantees that all rewriting reductions terminate. It additionally has two different implications for inductive and coinductive programs. For inductive programs like the one from example 7, it allows completeness of derivations; and for coinductive programs it allows lazy execution of S-resolution.

Example 9. Consider this modified version of example 4:

$$\begin{aligned} p(f(X), \kappa_1(\chi)) &\leftarrow p(X, \chi) \\ q(X, \kappa_2(\chi)) &\leftarrow q(X, \chi) \end{aligned}$$

The infinite S-resolution reductions for $q(X)$ now become both universally observable (all rewriting derivations terminate) and observationally productive:

$$q(X, \chi) \xrightarrow{\chi = \kappa_1(\chi')} q(X, \kappa_1(\chi')) \rightarrow q(X, \chi') \xrightarrow{\chi' = \kappa_1(\chi'')} q(X, \kappa_1(\chi'')) \rightarrow q(X, \chi'') \dots$$

The S-resolution can now detect that the above reduction is productive in its second argument, i.e. an infinite term $\kappa_1(\kappa_1(\dots))$ will be computed at infinity as a substitution to χ in the query $q(X, \chi)$.

The transformation above has a clear proof-relevant interpretation [13]. Each rule gets a unique extra argument, and when a rule is applied, the instantiation of the last term of an atom can be understood as recording a proof evidence. In the end, the last term of each atom in the goal will be instantiated with a term encoding a *proof* for the original atom A in the given program. For this reason, when we consider programs resulting from this transformation, we call terms in the last position inside atoms *proof terms*.

We now present our original results showing that the productivity transformation of programs does not change their declarative – inductive or coinductive – semantics.

Theorem 4. *For any given logic program P , $\Pi(P)$ is productive.*

Proof. In order to show that $\Pi(P)$ is strongly normalizing, a decreasing measure on goal clauses needs to be established. Such a measure is the total number of function symbols in all the proof terms of the goal. It is easy to see that each clause *reduces* this number by at least 1, since all the clauses in $\Pi(P)$ have the following shape:

$$p(\dots, \kappa_i(\chi_1, \dots, \chi_m)) \leftarrow p_1(\dots, \chi_1) \wedge \dots \wedge p_m(\dots, \chi_m) \quad \square$$

In light of its proof-relevant interpretation, the productivity transformation is expected not to change the intended meaning of the program since what it does is basically recording proof evidences. Indeed, we prove that Π does not affect the declarative semantics of logic programs: the following result states that the transformation is sound and complete with respect to both the inductive and coinductive models.

Theorem 5 (Soundness and completeness of the productivity transformation). *Given a logic program P and an atom A , each of the following implications hold for some proof term π :*

$$\begin{aligned} A \in M_P &\iff \Pi_\pi(A) \in M_{\Pi(P)} && \text{(inductive soundness and completeness)} \\ A \in M_P^{co} &\iff \Pi_\pi(A) \in M_{\Pi(P)}^{co} && \text{(coinductive soundness and completeness)} \end{aligned}$$

Proof. The full proof can be found in the Appendix A, and exploits the well-known identities $M_P = T_P \uparrow \omega$, and $M_P^{co} = T_P \downarrow \omega$ that hold for any logic program P when the complete Herbrand base⁹ is considered [21], and the two lemmas stating the following claims:

$$\begin{aligned} A \in T_P \uparrow n &\iff \Pi_\pi(A) \in T_{\Pi(P)} \uparrow n \text{ for some proof term } \pi \in B_{\Pi(P)}^{co} \\ A \in T_P \downarrow n &\iff \Pi_\pi(A) \in T_{\Pi(P)} \downarrow n \text{ for some proof term } \pi \in B_{\Pi(P)}^{co}. \end{aligned}$$

Both lemmas are proved by induction over n ; incidentally, the proofs show that π actually coincides with a proof tree for $A \in M_P$ and $A \in M_P^{co}$, respectively. \square

In light of the results above, we can safely apply the transformation Π to ensure productivity without changing the inductive and coinductive semantics of logic programs. This allows us to fruitfully exploit structural resolution as an inference engine together with abstract compilation.

6 Conclusions

Abstract compilation is a technique which uses logic programming for advanced static type analysis of object-oriented programs. To support analysis involving recursive types and programs, abstract compilation requires to consider the coinductive interpretation of the generated logic programs. We have identified a class of recursive methods for which the implementation of the inference engine based on Co-LP [22] does not work. We overcome this limitation by considering an inference engine based on structural resolution [19, 11]. S-resolution can only work if logic programs are universally observable; this property can be achieved by means of program transformation. We have proposed a new translation scheme for abstract compilation based on this transformation. We have proved that such a transformation preserves the semantics of the programs generated by abstract compilation.

While these results show that there are cases where S-resolution works better than Co-LP as inference engine for abstract compilation, we leave for further development the possibility of extending such a claim to prove that structural resolution always leads to analysis results which, if not improved, are at least comparable to those obtained with Co-LP as inference engine of abstract compilation. In particular, the recent results [20, 18] show that it is possible to integrate Co-LP loop detection into S-resolution, and thus to identify regular patterns and infer regular terms like Co-LP does. For example, it is possible to infer the answer $X = \text{cons}(0, X)$ for a query of Example 2 rather than giving a lazy answer $(X = \text{cons}(0, X'), X' = \text{cons}(0, X''), \dots)$. A potential use of coinductive proofs has been investigated for type class inference in Haskell [14], where irregular patterns arising in rewriting reductions (in the sense

⁹For the first identity the standard inductive Herbrand base could be equivalently considered.

of Definition 2) were considered. Future work will include investigation on the existence of a unifying approach to different coinductive methods in logic programming, type inference and abstract compilation.

We are developing a prototype implementation [10] based on a productivity checker for Logic Programming which provides an effective procedure for semi-deciding coinductive soundness of infinite S-resolution derivations, and, hence will result in a better assessment of the scalability of our proposed approach.

Another interesting direction for further development is the study of the “completeness” of our approach with respect to type inference problems. This would allow us to better understand for which classes of programs the resolution (successfully) terminates in case of type inference queries; our prototype implementation could help us identify classes of real programs, by conducting experiments on, possibly simplified, code taken from widely available open source projects.

References

- [1] K. Y. Ahn & A. Vezzosi (2016): *Executable Relational Specifications of Polymorphic Type Systems using Prolog*. In: *FLOPS 2016*, pp. 109–125, doi:10.1007/978-3-319-29604-3_8.
- [2] D. Ancona & A. Corradi (2014): *Sound and complete subtyping between coinductive types for object-oriented languages*. In: *ECOOP 2014*, pp. 282–307, doi:10.1007/978-3-662-44202-9_12.
- [3] D. Ancona & A. Corradi (2016): *A formal account of SSA in Java-like languages*. In: *FTfJP@ECOOP 2016*, p. 2, doi:10.1145/2955811.2955813.
- [4] D. Ancona & A. Corradi (2016): *Semantic subtyping for imperative object-oriented languages*. In: *OOPSLA 2016*, pp. 568–587, doi:10.1145/2983990.2983992.
- [5] D. Ancona & G. Lagorio (2009): *Coinductive Type Systems for Object-Oriented Languages*. In: *ECOOP 2009*, pp. 2–26, doi:10.1007/978-3-642-03013-0_2.
- [6] D. Ancona & G. Lagorio (2011): *Idealized coinductive type systems for imperative object-oriented programs*. *RAIRO - Theoretical Informatics and Applications* 45(1), pp. 3–33, doi:10.1051/ita/2011009.
- [7] D. Ancona & G. Lagorio (2012): *Static single information form for abstract compilation*. In: *IFIP TCS 2012*, pp. 10–27, doi:10.1007/978-3-642-33475-7_2.
- [8] D. Ancona et al. (2010): *Abstract Compilation of Object-Oriented Languages into Coinductive CLP(X): Can Type Inference Meet Verification?* In: *FoVeOOS 2010, Revised Selected Papers*, pp. 31–45, doi:10.1007/978-3-642-18070-5_3.
- [9] B. Courcelle (1983): *Fundamental Properties of Infinite Trees*. *Theor. Comput. Sci.* 25, pp. 95–169, doi:10.1016/0304-3975(83)90059-2.
- [10] E. Komendantskaya, M. Schmidt & Y. Li (2017): *Implementation of Structural Resolution and Coalgebraic Logic Programming*. Available at <https://github.com/coalp>.
- [11] E. Komendantskaya et al. (2017): *A productivity checker for logic programming*. Post-proc. *LOPSTR’16*. Available at <http://arxiv.org/abs/1608.04415>.
- [12] M. H. van Emden & M. A. N. Abdallah (1985): *Top-Down Semantics of Fair Computations of Logic Programs*. *J. Log. Program.* 2(1), pp. 67–75, doi:10.1016/0743-1066(85)90005-6.
- [13] P. Fu & E. Komendantskaya (2016): *Operational semantics of resolution and productivity in Horn clause logic*. *Formal Aspects of Computing*, pp. 1–22, doi:10.1007/s00165-016-0403-1.
- [14] P. Fu, E. Komendantskaya, T. Schrijvers & A. Pond (2016): *Proof Relevant Corecursive Resolution*. In: *FLOPS’16, LNCS 9613*, Springer, pp. 126–143, doi:10.1007/978-3-319-29604-3_9.
- [15] G. Gupta et al. (2007): *Coinductive Logic Programming and Its Applications*. In: *ICLP 2007*, pp. 27–44, doi:10.1007/978-3-540-74610-2_4.

- [16] P. Johann et al. (2015): *Structural Resolution for Logic Programming*. In: *Tech. Comm. of ICLP 2015*. Available at <http://arxiv.org/abs/1507.06010>.
- [17] E. Komendantskaya & P. Johann (2015): *Structural Resolution: a Framework for Coinductive Proof Search and Proof Construction in Horn Clause Logic*. CoRR abs/1511.07865. Available at <http://arxiv.org/abs/1511.07865>.
- [18] E. Komendantskaya & Y. Li (2017): *Productive Corecursion in Logic Programming*. In: *Under Review*.
- [19] E. Komendantskaya et al. (2016): *Coalgebraic logic programming: from Semantics to Implementation*. *J. Logic and Computation* 26(2), p. 745, doi:10.1093/logcom/exu026.
- [20] Y. Li (2017): *Structural Resolution with Coinductive Loop Detection*. In E. Komendantskaya & J. Power, editors: *Post-proceedings of CoALP-Ty'16*, Open Publishing Association. Available at <http://arxiv.org/abs/1703.08336>.
- [21] J. W. Lloyd (1987): *Foundations of Logic Programming, 2nd Edition*. Springer, doi:10.1007/978-3-642-83189-8.
- [22] L. Simon et al. (2006): *Coinductive Logic Programming*. In: *ICLP 2006*, pp. 330–345, doi:10.1007/11799573_25.
- [23] M. Sulzmann & P. J. Stuckey (2008): *HM(X) type inference is CLP(X) solving*. *J. Funct. Program.* 18(2), pp. 251–283, doi:10.1017/S0956796807006569.

A Proofs of Soundness and Completeness of the Productivity Translation

Definition 6. Given a logic program P , we use the following notations [21] for iterative application of the immediate consequence operator T_P :

$$\begin{aligned} T_P \uparrow 0 &= \emptyset & T_P \downarrow 0 &= B_P^{co} \\ T_P \uparrow n + 1 &= T_P(T_P \uparrow n) & T_P \downarrow n + 1 &= T_P(T_P \downarrow n) \end{aligned}$$

Lemma 7 (Inductive soundness and completeness of Π). *Given a logic program P , a (ground) atom $A \in B_P$ and a number $n \in \mathbb{N}$, the following implication holds for some proof term $\pi \in B_{\Pi(P)}$:*

$$A \in T_P \uparrow n \iff \Pi_\pi(A) \in T_{\Pi(P)} \uparrow n$$

Proof. The proof goes by induction over n . The base case is trivial since $T_P \uparrow 0 = \emptyset$. When $n > 0$ we prove the two implications as follows.

Case (\implies). If $n > 0$, since $T_P \uparrow n = T_P(T_P \uparrow n - 1)$, from the definition of T_P there must be a clause $C = A' \leftarrow B_1 \wedge \dots \wedge B_m$ in P and a (grounding) substitution σ such that $A = \sigma(A')$ and $\{\sigma(B_1), \dots, \sigma(B_m)\} \subseteq T_P \uparrow n - 1$. Since such a clause is in P , its translation $\Pi_\kappa(C)$ must be in $\Pi(P)$ for some constant κ :

$$\Pi_\kappa(C) = \Pi_\kappa(A' \leftarrow B_1 \wedge \dots \wedge B_m) = \Pi_{\kappa(\chi_1, \dots, \chi_m)}(A') \leftarrow \Pi_{\chi_1}(B_1) \wedge \dots \wedge \Pi_{\chi_m}(B_m)$$

By inductive hypothesis, for some proof terms π_1, \dots, π_m :

$$\{\sigma(B_1), \dots, \sigma(B_m)\} \subseteq T_P \uparrow n - 1 \implies \{\Pi_{\pi_1}(\sigma(B_1)), \dots, \Pi_{\pi_m}(\sigma(B_m))\} \subseteq T_{\Pi(P)} \uparrow n - 1$$

Since σ is computed with respect to the original program P it does not change the proof terms π_i , thus the substitution can be pushed out of the transformation:

$$\{\sigma(\Pi_{\pi_1}(B_1)), \dots, \sigma(\Pi_{\pi_m}(B_m))\} \subseteq T_{\Pi(P)} \uparrow n - 1$$

Through one application of $T_{\Pi(P)}$ we can apply the clause $\Pi_{\kappa}(C)$ to the atoms above with the substitution mapping every variable χ_i to the proof term π_i :

$$\sigma(\Pi_{\kappa(\pi_1, \dots, \pi_m)}(A')) \in T_{\Pi(P)} \uparrow n$$

Finally, we conclude by moving the substitution σ again, recalling that $\sigma(A') = A$ and $\kappa(\pi_1, \dots, \pi_m)$ is ground:

$$\sigma(\Pi_{\kappa(\pi_1, \dots, \pi_m)}(A')) = \Pi_{\sigma(\kappa(\pi_1, \dots, \pi_m))}(\sigma(A')) = \Pi_{\kappa(\pi_1, \dots, \pi_m)}(A) \in T_{\Pi(P)} \uparrow n$$

Case (\Leftarrow). The opposite implication can be proved in quite a similar way. From the definitions of T_P and $T_P \uparrow n$ we know the two following clauses are in P and $\Pi(P)$, respectively:

$$\begin{aligned} C_i = A' \leftarrow B_1 \wedge \dots \wedge B_m \in P \\ \Pi_{\kappa_i}(C_i) = \Pi_{\kappa_i(\chi_1, \dots, \chi_m)}(A') \leftarrow \Pi_{\chi_1}(B_1) \wedge \dots \wedge \Pi_{\chi_m}(B_m) \in \Pi(P) \end{aligned}$$

and, also, there is a substitution σ such that $A = \sigma(A')$ and, for $i = 1, \dots, n$:

$$\sigma(\Pi_{\chi_i}(B_i)) = \Pi_{\sigma(\chi_i)}(\sigma(B_i)) \in T_{\Pi(P)} \uparrow n - 1$$

Then, by inductive hypothesis:

$$\sigma(B_i) \in T_P \uparrow n - 1$$

Finally, applying T_P to $\{B_1, \dots, B_m\}$ with clause C_i and substitution σ , we can conclude that $\sigma(A') = A \in T_P \uparrow n$. □

Lemma 8 (Coinductive soundness and completeness of Π). *Given a logic program P , a (ground) atom $A \in B_P^{co}$ and a number $n \in \mathbb{N}$, the following implication holds for some proof term $\pi \in B_{\Pi(P)}^{co}$:*

$$A \in T_P \downarrow n \iff \Pi_{\pi}(A) \in T_{\Pi(P)} \downarrow n$$

(note that both A and π could be infinite)

Proof. The proof goes by induction over n . The base case is trivial since $T_P \downarrow 0 = B_P^{co}$ by definition. The inductive case is entirely similar to the proof of lemma 7, the only difference being the use of $T_P \downarrow n$ rather than $T_P \uparrow n$. □

Proof of Theorem 5 (*Soundness and completeness of the productivity transformation*). The following well-known results [21] about the least and greatest fixed points hold for any logic program P when the complete Herbrand base is considered:

$$\begin{aligned} M_P &= T_P \uparrow \omega \\ M_P^{co} &= T_P \downarrow \omega \end{aligned}$$

The theorem therefore directly follows from lemmas 7 and 8, that proved the strict relation between T_P and $T_{\Pi(P)}$. □