Analysing and Compiling Coroutines with Abstract Conjunctive Partial Deduction

Danny De Schreye, Vincent Nys(✉), and Colin Nicholson

Department of Computer Science, KU Leuven,
Celestijnenlaan 200A, 3001 Heverlee, Belgium
{danny.deschreye,vincent.nys}@cs.kuleuven.be

Abstract. We provide an approach to formally analyze the computational behavior of coroutines in Logic Programs and to compile these computations into new programs, not requiring any support for coroutines. The problem was already studied near to 30 years ago, in an analysis and transformation technique called Compiling Control. However, this technique had a strong ad hoc flavor: the completeness of the analysis was not well understood and its symbolic evaluation was also very ad hoc. We show how Abstract Conjunctive Partial Deduction, introduced by Leuschel in 2004, provides an appropriate setting to redefine Compiling Control. Leuschel's framework is more general than the original formulation, it is provably correct, and it can easily be applied for simple examples. We also show that the Abstract Conjunctive Partial Deduction framework needs some further extension to be able to deal with more complex examples.

1 Introduction

The work reported on in this paper is an initial step in a new project, in which we aim to formally analyze and automatically compile certain types of coroutining computations. Coroutines are a powerful means of supporting complex computation flows. They can be very useful for improving the efficiency of declaratively written programs, in particular for generate-and-test based programs. On the other hand, obtaining a deep understanding of the computation flows underlying the coroutines is notoriously difficult.

In this paper we restrict our attention to pure, definite Logic Programs. In this context, the problem was already studied nearly 30 years ago. Bruynooghe et al. (1986) and Bruynooghe et al. (1989) present an analysis and transformation technique for coroutines, called Compiling Control (CC for short). The purpose of the CC transformation is the following: transform a given program, P, into a program P', so that computation with P' under the standard selection rule mimics the computation with P under a non-standard selection rule. In particular, given a coroutining selection rule for a given Logic Program, the transformed program will execute the coroutining if it is evaluated under the standard selection rule of Prolog.

© Springer International Publishing Switzerland 2015
M. Proietti and H. Seki (Eds.): LOPSTR 2014, LNCS 8981, pp. 21–38, 2015.
DOI: 10.1007/978-3-319-17822-6_2

To achieve this, CC consists of two phases: an analysis phase and a synthesis phase. The analysis phase analyzes the computations of a program for a given query pattern and under a (non-standard) selection rule. The query pattern is expressed in terms of a combination of type, mode and aliasing information. The selection rule is instantiation-based, meaning that different choices in atom selection need to be based on different instantiations in these atoms. The analysis results in what is called a "trace tree", which is a finite upper part of a symbolic execution tree that one can construct for the given query pattern, selection rule and program. In the synthesis phase, a finite number of clauses are generated, so that each clause synthesizes the computation in some branch of the trace tree and such that all computations in the trace tree have been synthesized by some clause. The technique was implemented, formalized and proven correct, under certain fairly technical conditions.

Unfortunately, the CC transformation has a rather ad hoc flavor. It was very hard to show that the analysis phase of the transformation was complete, in the sense that a sufficiently large part of the computation had been analyzed to be able to capture all concrete computations that could possibly occur at run time. Even the very idea of a "symbolic execution" had an ad hoc flavor. It seemed that it should be possible to see this as an instance of a more general framework for analysis of computations.

Fortunately, since the development of CC a number of important advances have been achieved in analysis and transformation:

– General frameworks for abstract interpretation (e.g. Bruynooghe 1991) were developed. It is clear that abstract interpretation has the potential to provide a better setting for developing the CC analysis.
– Partial deduction of Logic Programs was developed (e.g. Gallagher 1986). Partial deduction seems very similar to CC, but the exact relationship was never identified. When John Lloyd and John Shepherdson formalized the issues of correctness and completeness of partial deduction in Lloyd and Shepherdson (1991), this provided a new framework for thinking about a complete analysis of a computational behavior and it was clear that some variant of this could improve the CC analysis.
– Conjunctive partial deduction (see De Schreye et al. 1999) seems even closer to CC. In an analysis for a CC transformation, one really does not want to split up the conjunctions of atoms into separate ones and then analyze the computations for these atoms separately. It is crucial that one can analyze the computation for certain atoms in conjunction (which is how conjunctive partial deduction generalizes partial deduction), so that their behavior under the non-standard selection rule may be observed.
– Finally, abstract (conjunctive) partial deduction (Leuschel 2004) brings all these features together. It provides an extension of (conjunctive) partial deduction in which the analysis is based on abstract interpretation, rather than on concrete evaluation.

In this paper we will demonstrate – mostly on the basis of examples – that abstract conjunctive partial deduction (ACPD for short) is indeed a suitable

framework to redefine CC in such a way that the flaws of the original approach are overcome. We show that for simple problems in the CC context, ACPD can, in principle, produce the transformation automatically. We also show that for more complex CC transformations, ACPD is still not powerful enough. We suggest an extension to ACPD that allows us to solve the problem and illustrate with an example that this extension is very promising.

After the preliminaries, in Sect. 3, we introduce a fairly refined abstract domain, including type, mode and aliasing information, and we show, by means of an example, how ACPD allows us to analyze a coroutine and compile the transformed program. In Sect. 4 we propose a more complex example and show why it is out of scope for ACPD. We introduce an additional abstraction in our domain and illustrate that this abstraction solves the problem. This abstraction, however, does not respect the requirements of the formalization of ACPD in Leuschel (2004). We end with a discussion.

2 Preliminaries

We assume that the reader is familiar with the basics of Logic Programming (Lloyd 1987). We also assume knowledge of the basics of abstract interpretation (Bruynooghe 1991) and of partial deduction (Lloyd and Shepherdson 1991).

In this paper, names of variables will start with a capital. Names of constants will start with a lower case character. Given a logic program P, Con_p, Var_p, Fun_p and $Pred_p$ respectively denote the sets of all constants, variables, functors and predicate symbols in the language underlying P. $Term_p$ will denote the set of all terms constructable from Con_p, Var_p and Fun_p. $Atom_p$ denotes the set of all atoms which can be constructed from $Pred_p$ and $Term_p$. We will often need to refer to conjunctions of atoms of $Atom_p$ and we denote the set of all such conjunctions as $ConAtom_p$.

We will introduce an abstract domain in the following section. The abstract domain will be based on a set of abstract constant symbols, $ACon_p$. Based on these, there is a corresponding set of abstract terms, $ATerm_p$, which consists of the terms that can be constructed from $ACon_p$ and Fun_p. $AAtom_p$ will denote the set of abstract atoms, being the atoms which can be constructed from $ATerm_p$ and $Pred_p$. Finally, $AConAtom_p$ denotes the set of conjunctions of elements of $AAtom_p$.

3 An Example of a CC Transformation, Using ACPD

In this section, we provide the intuitions behind our approach by means of a simple example. We use permutation sort as an illustration. The intention is to transform this program so that calls to *perm*/2 and *ord*/1 are interleaved.

Example 1 (Permutation sort).

```
sort(X,Y) ⟵ perm(X,Y), ord(Y).
perm([],[]).
perm([X|Y],[U|V]) ⟵
    del(U,[X|Y],W),perm(W,V).
```

```
del(X,[X|Y],Y).
del(X,[Y|U],[Y|V]) ⟵ del(X,U,V).
ord([]).
ord([X]).
ord([X,Y|Z]) ⟵
    X ≤ Y, ord([Y|Z]).
```

We now introduce the abstract domain. This domain consists of two types of new constant symbols: g and $a_i, i \in \mathbb{N}$. The symbol g denotes any ground term in the concrete language. The basic intuition for the symbols a_i is that they are intended to represent variables of the concrete domain. However, as we want the abstract domain to be closed under substitution (if an abstract term denotes some concrete term, then it should also denote all of its instances), an abstract term a_i will actually represent any term of the concrete language.

The subscript i in a term a_i is used to represent aliasing. If an abstract term, abstract atom or abstract conjunction of atoms contains a_i several times (with the same subscript), the denoted concrete terms, atoms or conjunctions of atoms contain the *same* term in all positions corresponding to those occupied by a_i. For instance, the abstract conjunction $perm(g, a_1), ord(a_1)$ denotes the concrete conjunctions $\{perm(t_1, t_2), ord(t_2) | t_1, t_2 \in Term_p \text{ and } t_1 \text{ is ground}\}$.

In addition to g and a_i, we will include all concrete constants in the abstract domain, so $Con_p \subseteq ACon_p$. This is not essential for the approach: we could develop a sound and effective ACPD for the CC transformation based on the abstract constants g and $a_i, i \in \mathbb{N}$, alone. However, including Con_p in $ACon_p$ makes the analysis more precise: some redundant paths in the analysis are avoided.

Definition 1 (Abstract domain). *The abstract domain consists of:*

- $ACon_p = Con_p \cup \{g\} \cup \{a_i | i \in \mathbb{N}\}$.
- $ATerm_p$*,* $AAtom_p$ *and* $AConAtom_p$ *are defined as the sets of the terms, atoms and conjunctions of atoms constructable from* $ACon_p$*,* Fun_p *and* $Pred_p$*.*

Next, we define the semantics of the abstract domain, through a concretization function γ. With slight abuse of notation, we use the same symbol γ to denote the concretization functions on $ATerm_p$, $AAtom_p$ and $AConAtom_p$.

In order to formalize the semantics of the aliasing, we need two auxiliary concepts: the subterm selection sequence and the aliasing context.

Definition 2 (Subterm selection sequence). *Let t be a term, atom or conjunction of atoms (either concrete or abstract).*

- $i \in \mathbb{N}_0$ *is a subterm selection sequence for* t*, if* $t = f(t_1, ..., t_n)$ *and* $i \leq n$*. The subterm of* t *selected by* i *is* t_i*.*
- $i_1.i_2.....i_n$ *is a subterm selection sequence for* t*, if* $t = f(t_1, ..., t_n)$*,* $i_1 \leq n$*,* $i_1 \in \mathbb{N}_0$ *and* $i_2.....i_n$ *is a subterm selection sequence for* t_{i_1}*. With an inductively defined notation, we denote by* $t_{i_1.i_2.....i_k}$ *the subterm of* $t_{i_1....i_{k-1}}$ *selected by* i_k*, with* $1 < k \leq n$*. We also refer to* $t_{i_1.i_2.....i_n}$ *as the subterm of* t *selected by* $i_1.i_2.....i_n$*.*

Note that, in this definition, we assume that a conjunction of atoms A_1, A_2,...,A_n is denoted as $\wedge(A_1, A_2, ..., A_n)$.

Example 2 (Subterm selection sequence). Let $t = f(g(h(X), 5), f(h(a), Y))$*, then* $t_{1.1.1} = X$*,* $t_{2.1.1} = a$*.*

Definition 3 (Aliasing context). *Let t be an abstract term, atom or conjunction of atoms. The aliasing context of t, denoted $AC(t)$, is the finite set of pairs (sss_1, sss_2) of subterm selection sequences of t, such that $t_{sss_1} = t_{sss_2} = a_i$ for some $i \in \mathbb{N}$.*

Example 3 (Aliasing context). Let $t = p(f(a_2, g), a_1, a_2, g(h(a_1)))$*, then* $AC(t) = \{(1.1, 3), (2, 4.1.1)\}$.

Definition 4 (Concretization function). *The concretization function* $\gamma : ATerm_p \cup AAtom_p \cup AConAtom_p \rightarrow 2^{Term_p} \cup 2^{Atom_p} \cup 2^{ConAtom_p}$ *is defined as:*

- $\gamma(c) = \{c\}$*, for any* $c \in Con_p$
- $\gamma(g) = \{t \in Term_p | t \text{ is ground}\}$
- $\gamma(a_i) = Term_p,\ i \in \mathbb{N}$
- $\gamma(f(at_1, ..., at_n)) = \{f(t_1, ..., t_n) | t_i \in \gamma(at_i), i = 1...n$*, and let t denote* $f(t_1, ..., t_n)$*, then for all* $(sss_1, sss_2) \in AC(f(at_1, ..., at_n)) : t_{sss_1} = t_{sss_2}\}$

Example 4 (Concretization function). $\gamma(p(f(a_2, g), a_1, a_2, q(h(a_1)))) = \{p(f(t_1, t_2), t_3, t_1, q(h(t_3))) | t_1, t_3 \in Term_p, t_2$ *ground term of* $Term_p\}$

The abstract domain introduced above is infinitely large. There are two causes for this. Terms can be nested unboundedly deep, therefore infinitely many different terms exist. In addition, there are infinitely many $a_i, i \in \mathbb{N}$, symbols.

If so desired, the abstract domain can be refined, so that it becomes finite. This is done by using depth-k abstraction and by defining an equivalence relation on $\{a_i | i \in \mathbb{N}\}$. For the purpose of this paper, the infinite size of the abstract domain is not a problem.

Let us return to the permutation sort example. ACPD requires a top-level abstract atom (or conjunction) to start the transformation. Let $sort(g, a_1)$ be this atom. In the context of the $\mathcal{A}$-coveredness condition of partial deduction, our initial set $\mathcal{A}$ is $\{sort(g, a_1)\}$.

Below, we construct a finite number of finite, abstract partial deduction derivation trees for abstract (conjunctions of) atoms. The construction of these trees assumes an "abstract unification" and an "abstract unfold" operation. Their formal definitions can be found in Annex (2014). For now, we only show their effects in abstract partial derivation trees.

Next, we need an "oracle" that decides on the selection rule applied in the abstract derivation trees. This oracle mainly has two functions:

- to decide whether an obtained goal should be unfolded further, or whether it should be kept residual (to be split and added to $\mathcal{A}$),
- to decide which atom of the current goal should be selected for unfolding.

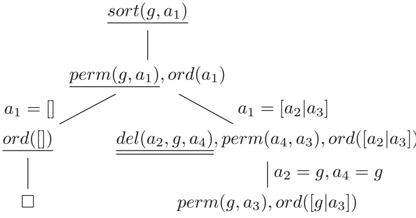

Fig. 1. Abstract tree for $sort(g, a_1)$

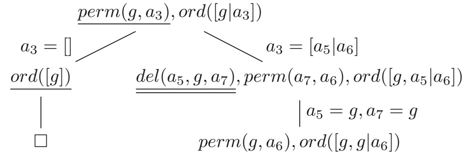

Fig. 2. Abstract tree for $perm(g, a_3)$, $ord([g|a_3])$

In fact, we will use a third type of decision that the oracle may make: it may decide to "fully evaluate" a selected atom. This type of decision is not commonly supported in partial deduction. What it means is that we decide not to transform a certain predicate of the original program, but merely keep its original definition in the transformed program. In partial deduction, this can be done by never selecting these atoms, including them in $\mathcal{A}$ and including their original definition in the transformed program.

In our setting, however, we want to know the effect that solving the atom has on the remainder of the goal. Therefore, we will assume that an abstract interpretation over our abstract domain computes the abstract bindings that solving the atom results in. These are applied to the remainder of the goal. Note that this cannot easily be done in standard partial deduction, as fully evaluating an atom during (concrete) partial deduction may not terminate. In Vidal (2011), a similar functionality is integrated in a hybrid approach to conjunctive partial deduction.

For now, we simply assume the existence of the oracle. Figures 1, 2 and 3 show the abstract partial derivation trees that ACPD may build for permutation sort and top level $\mathcal{A} = \{sort(g, a_1)\}$.

In these figures, in each goal, the atom selected for abstract unfolding is underlined. If an atom is underlined twice, this expresses that the atom was selected for full abstract interpretation.

Both unfolding and full abstract evaluation may create bindings. Our abstract unification only collects bindings made on the a_i terms. Bindings created on g terms are not relevant.

In the left branch of the tree in Fig. 1 we see the effect of including the concrete constants in the abstract domain. As a result, the binding for a_1 is $[]$, instead of g. If we had not included Con_p in $ACon_p$, then $ord(g)$ would have required a full analysis, using the three clauses for $ord/1$.

A goal with no underlined atom indicates that the oracle selects no atom and decides to keep the conjunction residual. After the construction of the tree in Fig. 1, ACPD adds the abstract conjunction $perm(g, a_3), ord([g|a_3])$ to $\mathcal{A}$. ACPD starts a new tree for this atom. This tree is shown in Fig. 2.

The tree is quite similar to the one in Fig. 1. The main difference is that, in the residual leaf, the ord atom now has a list argument with two g elements.

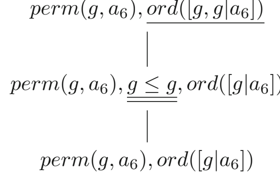

Fig. 3. Abstract tree for $perm(g, a_6), ord([g, g|a_6])$

This pattern does not yet exist in the current $\mathcal{A}$ and is therefore added to $\mathcal{A}$. A third abstract tree is computed for $perm(g, a_6), ord([g, g|a_6])$, shown in Fig. 3.

In Fig. 3, the residual leaf $perm(g, a_6), ord([g|a_6])$ is a renaming of the conjunction $perm(g, a_3), ord([g|a_3])$, which is already contained in $\mathcal{A}$. Therefore, ACPD terminates the analysis, concluding $\mathcal{A}$-coveredness for $\mathcal{A} = \{sort(g, a_1), \wedge(perm(g, a_3), ord([g|a_3])), \wedge(perm(g, a_6), ord([g, g|a_6]))\}$.

In standard (concrete) conjunctive partial deduction, the analysis phase would now be completed. In ACPD, however, we need an additional step. In the abstract partial derivation trees, we have not collected the concrete bindings that unfolding would produce. These are required to generate the resolvents. Therefore, we need an additional step, constructing essentially the same three trees again, but now using concrete terms and concrete unification.

We only show one of these concrete derivation trees in Fig. 4. It corresponds to the tree in Fig. 2. We define the root of a concrete derivation tree corresponding to an abstract tree as follows.

Definition 5 (Concrete conjunctions in the root). *Let $acon \in \mathcal{A}$, then the conjunction in the root of the corresponding concrete tree, denoted as $c(acon)$, is obtained by replacing any g or a_i symbol in $acon$ by a fresh free variable, ensuring that multiple occurrences of a_i, with the same subscript i, are replaced by identical variables.*

When unfolding the concrete tree, every abstract unfolding of the abstract tree is mimicked, using the same clauses, over the concrete domain.

The step of full abstract interpretation of the $del(a_5, g, a_7)$ atom in Fig. 2 has no counterpart in Fig. 4. The atom $del(U, [X_1|X_2], W)$ is kept residual and the $del/3$ clauses are added to the transformed program.

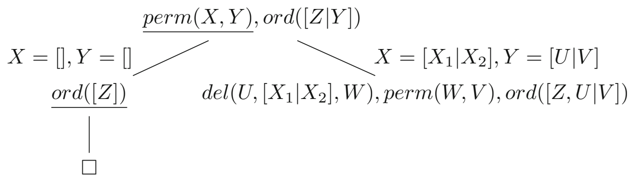

Fig. 4. Concrete tree corresponding to Fig. 2

More specifically, using a renaming $p_1(X, Y, Z)$ for $\wedge(perm(X, Y), ord([Z|Y]))$ and $p_2(W, V, Z, U)$ for $\wedge(perm(W, V), ord([Z, U|V]))$, we synthesize the following resolvents from the tree in Fig. 4:
$p_1([], [], Z) \longleftarrow .$
$p_1([X_1|X_2], [U|V], Z) \longleftarrow del(U, [X_1|X_2], W), p_2(W, V, Z, U).$
From the counterparts of the trees in Figs. 1 and 3, we obtain the following additional resultants:
$sort([], []).$
$sort([X_1|X_2], [Y_1|Y_2]) \longleftarrow del(Y_1, [X_1|X_2], Z), p_1(Z, Y_2, Y_1).$
$p_2(U, V, W, X)W \leq X, p_1(U, V, X).$

This transformation inherits correctness results from ACPD. In particular, $\mathcal{A}$-closedness and independence guarantee the completeness and correctness of the analysis. In addition, the transformation preserves all computed answers (in both directions) and finite failure of the transformed program implies finite failure of the original.

4 A More Complex Example, Introducing the *multi* Abstraction

In Sect. 3 we have shown that ACPD is indeed sufficient to formally revisit CC for a simple example. However, for more complex examples, ACPD still lacks expressivity. Consider the following prime number generator.

Example 5 (Prime numbers)

```
primes(N,P) ? integers(2,I),sift(I,P),len(P,N).
integers(N,[]).
integers(N,[N|I]) ? M is N+1, integers(M,I).
sift([N|Is],[N|Ps]) ? filter(N,Is,F), sift(F,Ps).
sift([],[]).
divides(N,M) ? X is M mod N, X is 0.
not_divide(N,M) ? X is M mod N, X > 0.
filter(N,[M|I],F) ? divides(N,M), filter(N,I,F).
filter(N,[M|I],[M|F]) ? not_divide(N,M), filter(N,I,F).
filter(N,[],[]).
len([],0).
len([H|T],N) ? M is N - 1, len(T,M).
```

The program is intended to be called with a goal $primes(N, P)$, with N a positive integer and P a free variable. The $integers/2$ predicate generates growing lists of integer numbers. $filter/3$ represents the removal of all multiples of a single integer N from a list. $sift/2$ recursively filters out multiples of an initial list element which is prime.

The complete ACPD style analysis is available in Annex (2014). We only present some relevant parts.

The top level goal for the abstract analysis is $primes(g, a_1)$, so that the initial set $\mathcal{A}$ is $\{primes(g, a_1)\}$. A first abstract derivation tree describes the initialization for the computation. It contains a branch leading to an empty goal (success branch) and a branch with the leaf: $\wedge(integers(g, a_3), filter(g, a_3, a_5), sift(a_5, a_4), len(a_4, g))$, which is added to $\mathcal{A}$.

Next, we construct an abstract derivation tree for the latter conjunction. This gives a successful branch with an empty conjunction in the leaf, a branch ending in a renamed version of the above conjunction, and a third branch, with the following leaf, which is added to $\mathcal{A}$: $\wedge(integers(g, a_4), filter(g, a_4, a_5), filter(g, a_5, a_7), sift(a_7, a_6), len(a_6, g))$.

At this point it becomes clear that an analysis following only the steps shown in Sect. 3 will not terminate. The two abstract conjunctions, most recently added to $\mathcal{A}$, are identical – up to renaming of a_i's – except that the latter conjunction contains two atoms $filter(g, a_i, a_j)$, instead of just one. A further analysis, building additional derivation trees, will result in the construction of continuously growing conjunctions, with continuously increasing numbers of $filter/3$ atoms.

We could solve this by cutting the goal into two smaller conjunctions and adding these to $\mathcal{A}$. However, all these atoms are generators or testers in the coroutine and depend on eachother. By splitting the conjunction, we would no longer be able to analyze the coroutine.

One of the restrictions imposed by ACPD is that for any abstract conjunction of atoms, $acon \in AConAtom_p$, there exists a concrete conjunction, $con \in ConAtom_p$, such that: for all $con_i \in \gamma(acon)$: con_i is an instance of con. In practice, this means that an abstract conjunction is not allowed to represent a set of concrete conjunctions whose elements have a distinct number of conjuncts. However, in order to solve the problem observed in our example, we need the ability to represent a set of conjunctions, with a growing number of atoms, by an abstract atom. Therefore, we need to extend ACPD.

We extend our abstract domain and introduce a new abstraction, $multi/4$, which makes it possible to represent growing conjunctions, with a number of copies of a single abstract atom.

To define this abstraction is rather difficult. This is because we do not only want to be able to represent a conjunction of multiple, identically instantiated atoms, but also their aliasing with the context in which they occur, as well as the aliasing between consecutive atoms in the conjunction.

We first introduce a parameterized naming scheme for a_i constants and apply this to abstract atoms.

Definition 6 (Parameterized naming and parameterized abstract atom). *Let $A \in AAtom_p$. By $Id(A)$, we denote a unique identifier associated with A.*

Let $a_j \in ACon_p, j \in \mathbb{N}$, such that a_j occurs in A, then the parameterized naming of a_j is the symbol $a_{Id(A),i,j}$.

Let $A \in AAtom_p$. The parameterized atom for A, $p(A)$, is obtained by replacing every a_j occurring in A by its parameterized naming, $a_{Id(A),i,j}$.

The new abstraction $multi/4$ will depend on the context (the abstract conjunction) in which it occurs. This context may contain abstract constants, a_j. It may also contain parameterized namings of abstract constants, $a_{Id(A),i,j}$. This is due to the fact that a $multi/4$ abstraction will typically contain parameterized namings and that an abstract conjunction will be allowed to contain multiple $multi/4$ abstractions. Therefore, the context of one $multi/4$ abstraction may contain another $multi/4$ abstraction.

Definition 7 (Context). *A context is an abstract conjunction and is denoted as C. Given a context C, we denote $a(C) = \{a_j \in ACon_p | a_j \text{ occurs in } C\}$, we denote $pa(C) = \{a_{Id(A),i,j} | a_{Id(A),i,j} \text{ occurs in } C\}$.*

Definition 8 (multi abstraction). *A multi abstraction is a construct of the form $multi(p(A), First, Consecutive, Last)$, where:*

- *$p(A)$ is the parameterized atom for some $A \in AAtom_p$.*
- *$First$ is a conjunction of equalities $a_{Id(A),1,j} = b_j$, where $b_j \in a(C) \cup pa(C)$ and all left-hand sides of the equalities are distinct.*
- *$Consecutive$ is a conjunction of equalities $a_{Id(A),i+1,j} = a_{Id(A),i,j'}$, where $j, j' \in \mathbb{N}$ and all left-hand sides of the equalities are distinct.*
- *$Last$ is a conjunction of equalities $a_{Id(A),k,j} = b_j$, where $b_j \in a(C) \cup pa(C)$ and all left-hand sides of the equalities are distinct.*

Example 6 (multi/4 abstraction). We return to the primes example, with the two abstract conjunctions already added to $\mathcal{A}$. We can rename the indices of the a_j constants in one of these conjunctions in order to make the contexts in which the $filter(g, a_i, a_j)$ atoms occur identical for both conjunctions, e.g.: $\wedge(integers(g, a_3), filter(g, a_3, a_5), sift(a_5, a_4), len(a_4, g))$ and $\wedge(integers(g, a_3), filter(g, a_3, a_6), filter(g, a_6, a_5), sift(a_5, a_4), len(a_4, g))$.

Now we can generalize these two abstract conjunctions using the $multi/4$ abstraction: Let $A = filter(g, a_3, a_6)$. Then, the abstract conjunction is: $\bigwedge(integers(g, a_3), multi(filter(g, a_{Id(A),i,3}, a_{Id(A),i,6}), \wedge(a_{Id(A),1,3} = a_3),$ $\wedge\, (a_{Id(A),i+1,3} = a_{Id(A),i,6}), \wedge(a_{Id(A),k,6} = a_5)), sift(a_5, a_4), len(a_4, g))$

Here, expressions such as $\wedge(a_{Id(A),1,3} = a_3)$ represent conjunctions with only one conjunct.

Conversely, abstract conjunctions containing $multi/4$ abstractions, such as the one above, represent infinitely many abstract conjunctions without the $multi/4$ abstraction. In the example, these contain either one or multiple $filter(g, a_i, a_j)$ atoms.

In what follows, we will omit the $Id(A)$ subscript in the parameterized namings $a_{Id(A),i,j}$ and just refer to $a_{i,j}$ instead. The $Id(A)$ subscript is only relevant for abstract conjunctions containing multiple $multi/4$ abstractions, a case which we will not consider for the moment.

In order to describe the abstract conjunctions represented by an abstract conjunction containing a $multi/4$ abstraction, we need the ability to map parameterized namings back to ordinary a_j constants. This requires the following concepts.

Definition 9 (concrete index assignment mapping). *Let $n \in \mathbb{N}$. The concrete index assignment mapping, $R(i,n)$, is a mapping defined on any syntactic construct, S, containing parameterized namings $a_{i,j}$. $R(i,n)$ replaces every occurrence of a parameterized naming $a_{i,j}$ in S by the parameterized naming $a_{n,j}$.*

Example 7 (concrete index assignment mapping). $R(i,1)(filter(g,a_{i,3},a_{i,6})) = filter(g,a_{1,3},a_{1,6})$. $R(i,k)(filter(g,a_{i,3},a_{i,6})) = filter(g,a_{k,3},a_{k,6})$.

Definition 10 (double-index mapping). *The double-index mapping, ψ, is a mapping defined on any syntactic construct, S, containing parameterized namings $a_{i,j}$. ψ replaces every occurrence of a parameterized naming $a_{i,j}$ in S by a_{i_j}, where i_j denotes a fresh element of $\mathbb{N}$, not occurring in any a_i yet.*

Example 8 (double-index mapping). $\psi(filter(g,a_{i,3},a_{i,6})) = filter(g,a_{i_3},a_{i_6})$, *with i_3, i_6 fresh elements of $\mathbb{N}$.*

Definition 11 (substitution corresponding to equality constraints). *Let $Constraint$ be a conjunction of equality constraints, $a_{i,j} = b_j$, with $a_{i,j}$ parameterized namings, and such that all left-hand sides of equalities are mutually distinct. The substitution corresponding to $Constraint$ is the substitution $\Theta_{Constraint} = \{\psi(a_{i,j})/\psi(b_j) | a_{i,j} = b_j \in Constraint\}$.*

Note that this definition is meant to deal with the conjunctions of equalities in the $First$, $Consecutive$ and $Last$ arguments of the $multi/4$ abstraction.

Example 9 (substitutions corresponding to equality constraints). For the conjunctions of equality constraints in Example 6, the corresponding substitutions are: $\Theta_{First} = \{a_{1_3}/a_3\}$, $\Theta_{Consecutive} = \{a_{(i+1)_3}/a_{i_6}\}$, $\Theta_{Last} = \{a_{k_6}/a_5\}$.

With these notions, we can now describe the abstract conjunctions represented by a $multi/4$ abstraction.

Definition 12 (Abstract conjunctions represented by multi/4). *The abstract conjunctions represented by $multi(p(A), First, Consecutive, Last)$ are:*

- $\psi(R(i,1)(p(A)))\Theta_{First} \circ \Theta_{R(k,1)(Last)}$, *and*
- $\psi(R(i,1)(p(A)))\Theta_{First} \wedge \psi(R(i,2)(p(A)))\Theta_{R(i,1)(Consecutive)} \wedge$
  $\ldots \wedge \psi(R(i,k)(p(A)))\Theta_{R(i,k-1)(Consecutive)} \circ \Theta_{Last}$, *with $k > 1$.*

Example 10 (Abstract conjunctions represented by $multi/4$). For the $multi/4$ abstraction in Example 6, $multi(filter(g,a_{i,3},a_{i,6}), \wedge(a_{1,3} = a_3), \wedge(a_{i+1,3} = a_{i,6}), \wedge(a_{k,6} = a_5))$, *the represented abstract conjunctions are:*

- $filter(g,a_3,a_5)$, *and*
- $filter(g,a_3,a_{1_6}) \wedge filter(g,a_{1_6},a_{2_3}) \wedge \ldots \wedge filter(g,a_{(k-1)_6},a_5), k > 1$.

Next, we need to define the abstract unfolding of a $multi/4$ abstraction. Unfolding a $multi/4$ abstraction makes a case split. Either the $multi/4$ abstraction represents only one abstract atom, or it represents more than one. In both cases the bindings with the context and, in the latter case, the bindings between consecutive atoms, need to be respected.

Definition 13 (Abstract unfold of $multi/4$**).** *Abstract unfold of multi produces a branching in the abstract derivation tree. An abstract atom* $multi(p(A), First, Consecutive, Last)$ *is replaced in one branch by* $\psi(R(i,1)(p(A)))\Theta_{First} \circ \Theta_{R(k,1)(Last)}$ *and in a second branch by* $\psi(R(i,1)(p(A)))\Theta_{First} \wedge multi(p(A), NewFirst, Consecutive, Last)$*, where* $NewFirst = \wedge\{a_{1,j} = a_{1_{j'}} | a_{(i+1),j} = a_{i,j'} \in Consecutive\}$.

Example 11 (Abstract unfold of $multi/4$*). Again returning to Example 6, abstract unfold of* $multi(filter(g, a_{i,3}, a_{i,6}), \wedge(a_{1,3} = a_3), \wedge(a_{i+1,3} = a_{i,6}), \wedge(a_{k,6} = a_5))$ *produces in one branch* $filter(g, a_3, a_5)$ *and in the other branch* $filter(g, a_3, a_{1_6}) \wedge multi(filter(g, a_{i,3}, a_{i_6}), \wedge(a_{1,3} = a_{1,6}), \wedge(a_{i+1,3} = a_{i,6}), \wedge(a_{k,6} = a_5))$.

A few comments on this definition are in order. First, the definition of $NewFirst$ may seem strange, because both sides of the equalities have a "1" index. However, note that on the left-hand side of the equality, it is in a parameterized naming, $a_{1,j}$, referring to the first atom represented by the $multi/4$, while on the right-hand side, it is in an abstract atom $a_{1_{j'}}$, referring to an atom that was just moved outside of the $multi/4$. Second, it is important to remember that the abstract constants a_{1_j} are produced by a $\psi(a_{1,j})$ call and that their index 1_j needs to be a fresh index, not yet occurring in the expressions. This is particularly important in cases where we perform several abstract unfoldings of $multi/4$ in sequence. At each unfold, new fresh subscripts need to be introduced.

Finally, we need to define abstract generalization with $multi/4$, allowing us to replace conjunctions of identically instantiated and similarly aliased abstract atoms by a *multi* construct.

Definition 14 (Abstract generalization with $multi/4$**).** *Let* $A \in AAtom_p$*. Let* $A_1, ..., A_k \in AAtom_p$ *and let* $\bigwedge_{l=1,k} A_l$ *occur in a context of abstract atoms* C*. Let* $a(C)$ *and* $pa(C)$ *respectively be the abstract constants and the parameterized namings occurring in* C*. Let* $r_l, l = 1, k$*, be renamings of* A*, such that* $r_l(A) = A_l$*. In particular, for any* a_i *occurring in* A*,* $r_l(a_i)$ *occurs at the same subterm selection sequence position in* A_l*.*

$Gen(\bigwedge_{l=1,k} A_l) = multi(p(A), First, Consecutive, Last)$ *is the abstract generalization with* $multi/4$ *of* $\bigwedge_{l=1,k} A_l$ *in* C *if:*

- *for any* $b_j \in a(C) \cup pa(C), a_{1,j} = b_j \in First$ *if and only if* $r_1(a_j) = b_j$
- $a_{i+1,j} = a_{i,j'} \in Consecutive$ *if and only if* $r_{i+1}(a_j) = r_i(a_{j'})$
- *for any* $b_j \in a(C) \cup pa(C)$*,* $a_{k,j} = b_j \in Last$ *if and only if* $r_k(a_j) = b_j$

We can extend the above definition to allow generalizations $Gen(\wedge_{l=1,k} A_l \wedge multi(p(A), First, Consecutive, Last)) = multi(p(A), First', Consecutive, Last)$ and generalizations $Gen(multi(p(A), First, Consecutive, Last), \wedge \bigwedge_{l=1,k} A_l) = multi(p(A), First, Consecutive, Last')$. We omit the details for these generalizations. We illustrate abstract generalization with $multi/4$ in our running example below.

Let us return to the prime numbers example. Observing the growing number of $filter/3$ atoms in our last conjunction (w.r.t. the conjunction already present

in $\mathcal{A}$), we perform the generalization: $Gen(\wedge(filter(g, a_4, a_5), filter(g, a_5, a_7))) = multi(filter(g, a_{1,i,4}, a_{1,i,5}), \wedge(a_{1,1,4} = a_4), \wedge(a_{1,i+1,4} = a_{1,i,5}), \wedge(a_{1,k,5} = a_7))$. Here, we include the $Id(A)$ again, because we will have multiple $multi/4$ abstractions. We arbitrarily select $Id(A)$ to be 1.

Then we construct a new abstract derivation tree for this conjunction, including – among others – an abstract unfold of $multi/4$ and abstract generalizations with $multi/4$. In Fig. 5, we show this abstract tree.

After abstract unfolding of $integers(g, a_1)$, the tree contains an abstract unfolding of $multi(filter(g, a_{1,i,1}, a_{1,i,2}), \wedge(a_{1,1,1} = [g|a_4]), \wedge(a_{1,i+1,1} = a_{1,i,2}), \wedge(a_{1,k,2} = a_2))$. This unfolding can lead to one instance of $filter/3$ or several. If there is only one filter, a full evaluation of $divides(g, g)$ eventually leads to an empty goal. A full evaluation of $does_not_divide(g, g)$, on the other hand, leads to a new generalization which produces a renaming of the root of this tree.

Eventually, the analysis ends up with a final set $\mathcal{A}$:

$$\{ \wedge (primes(g, a_1)),$$
$$\wedge (integers(g, a_1), multi(filter(g, a_{1,i,1}, a_{1,i,2}), \wedge(a_{1,1,1} = a_1), \wedge (a_{1,i+1,1} = a_{1,i,2}), \wedge(a_{1,k,2} = a_2)), sift(a_2, a_3), len(a_3, g)),$$
$$\wedge (multi(filter(g, a_{1,i,1}, a_{1,i,2}), \wedge(a_{1,1,1} = []), \wedge(a_{1,i+1,1} = a_{1,i,2}), \wedge (a_{1,k,2} = a_2)), sift(a_2, a_3), len(a_3, g)),$$
$$\wedge (integers(g, a_4), multi(filter(g, a_{2,i,4}, a_{2,i,6}), \wedge(a_{2,1,4} = a_4), \wedge (a_{2,i+1,4} = a_{2,i,6}), \wedge(a_{2,k,6} = a_6)), multi(filter(g, a_{1,i,1}, a_{1,i,2}), \wedge (a_{1,1,1} = [g|a_{2,k,6}]), \wedge(a_{1,i+1,1} = a_{1,i,2}), \wedge(a_{1,k,2} = a_2)), sift(a_2, a_3), len(a_3, g))\}$$

$integers(g, a_1)$, $multi(filter(g, a_{1,i,1}, a_{1,i,2}), \{a_{1,1,1} = a_1\}, \{a_{1,i+1,1} = a_{1,i,2}\}, \{a_{1,k,2} = a_2\})$, $sift(a_2, a_3), len(a_3, g)$

$a_1 = []$

$multi(filter(g, a_{1,i,1}, a_{1,i,2}), \{a_{1,1,1} = []\}, \{a_{1,i+1,1} = a_{1,i,2}\}, \{a_{1,k,2} = a_2\})$, $sift(a_2, a_3), len(a_3, g)$

$a_1 = [g|a_4]$

$integers(g, a_4)$, $multi(filter(g, a_{1,i,1}, a_{1,i,2}), \{a_{1,1,1} = [g|a_4]\}, \{a_{1,i+1,1} = a_{1,i,2}\}, \{a_{1,k,2} = a_2\})$, $sift(a_2, a_3), len(a_3, g)$

$integers(g, a_4), filter(g, [g|a_4], a_2), sift(a_2, a_3), len(a_3, g)$

$integers(g, a_4), filter(g, [g|a_4], a_5)$, $multi(filter(g, a_{1,i,1}, a_{1,i,2}), \{a_{1,1,1} = a_5\}, \{a_{1,i+1,1} = a_{1,i,2}\}, \{a_{1,k,2} = a_2\})$, $sift(a_2, a_3), len(a_3, g)$

$a_2 = [g|a_5]$

$integers(g, a_4), divides(g, g), filter(g, a_4, a_2), sift(a_2, a_3), len(a_3, g)$

$integers(g, a_4), does_not_divide(g, g), filter(g, a_4, a_5), sift([g|a_5], a_3), len(a_3, g)$

$integers(g, a_4), filter(g, a_4, a_2), sift(a_2, a_3), len(a_3, g)$

$integers(g, a_4), filter(g, a_4, a_5), sift([g|a_5], a_3), len(a_3, g)$

$a_3 = [g|a_6]$

$integers(g, a_4), filter(g, a_4, a_5), filter(g, a_5, a_7), sift(a_7, a_6), len([g|a_6], g)$

$integers(g, a_4), filter(g, a_4, a_5), filter(g, a_5, a_7), sift(a_7, a_6), len(a_6, g)$

generalize

$integers(g, a_4)$, $multi(filter(g, a_{1,i,4}, a_{1,i,5}), \{a_{1,1,1} = a_4\}, \{a_{1,i+1,4} = a_{1,i,5}\}, \{a_{1,k,2} = a_7\})$, $sift(a_7, a_6), len(a_6, g)$

Fig. 5. Abstract unfolding of $integers(g, a_1), multi(filter(g, a_{1,i,1}, a_{1,i,2}), \wedge(a_{1,1,1} = a_1), \wedge(a_{1,i+1,1} = a_{1,i,2}), \wedge(a_{1,k,2} = a_2)), sift(a_2, a_3), len(a_3, g)$

All non-empty leaves in the abstract derivation trees for these atoms are (renamings of) elements of $\mathcal{A}$. This shows $\mathcal{A}$-coveredness and the abstract phase of the analysis terminates.

Similar to what was observed for permutation sort in Sect. 3, we still need an extra analysis to collect the concrete bindings, so that the resultants can be generated. Special care is required for the $multi/4$ abstraction. There are three issues: how to represent $multi/4$ in the concrete domain, how to deal with the concrete counterparts of abstract generalization with $multi/4$ and abstract unfolding of $multi/4$.

Definition 5, in Sect. 3, defined the concrete counterparts of the conjunctions in $\mathcal{A}$. We extend it to $multi(A)$:

Definition 15 (Concrete conjunction for $multi(A, First, Consecutive, Last)$**).** *Let* $A \in AAtom_p$, *then* $c(multi(p(A), First, Consecutive, Last)) = multi([c(A)|T])$, *with* T *a fresh variable.*

It may seem strange that in the concrete analysis phase we omit the three arguments $First$, $Consecutive$ and $Last$. These arguments are needed in the abstract analysis to correctly capture the data flow and to correctly model the unfolding under the coroutining selection rule. In the concrete analysis phase, as we are completely mimicking the unfolding in the corresponding abstract trees, we are still performing the correct selection. Moreover, the only point of the concrete analysis phase is to collect the bindings produced by unfolding the concrete clauses. The extra arguments are not needed for this purpose.

Example 12 $c(multi(filter(g, a_{1,1,1}, a_{1,1,2}), \wedge(a_{1,1,1} = a_1), \wedge(a_{1,i+1,1} = a_{1,i,2}), \wedge(a_{1,k,2} = a_2))) = multi([filter(X, I1, F1)|T])$

For the abstract generalization with $multi/4$, we define the concrete counterpart as follows.

Definition 16 (Concrete generalization). *Let* $A \in AAtom$.

- *If the abstract generalization with multi/4 is of the type* $Gen(\bigwedge_{i=1,n} A) = multi(A, First, Consecutive, Last)$, *then the corresponding node in the concrete derivation contains* $c(\bigwedge_{i=1,n} A)$. *The concrete generalization is defined as* $ConGen(c(\bigwedge_{i=1,n} A)) = multi(c([A, \ldots, A]))$, *with* n *members in the list.*
- *If the abstract generalization with multi/4 is of the type* $Gen((\bigwedge_{i=1,n} A) \wedge multi(A, First, Consecutive, Last)) = multi(A, First', Consecutive, Last)$, *then the corresponding node in the concrete derivation contains* $c(\bigwedge_{i=1,n} A) \wedge multi(List)$, *where List is a list of at least one* $c(A)$. *The concrete generalization is defined as* $ConGen(c(\bigwedge_{i=1,n} A) \wedge multi(List)) = multi([c(A), \ldots, c(A)|List])$ *with* n *new members added to List.*
- *The third case,* $Gen(multi(A, First, Consecutive, Last) \wedge (\bigwedge_{i=1,n} A)) = multi(A, First, Consecutive, Last')$, *is treated similarly to the previous case, but the concrete atoms are appended to the existing list.*

Example 13 (Concrete generalization). Let $integers(A, B), filter(C, B, D), filter(E, D, F), sift(F, G), len(G, H)$ *occur in a concrete conjunction in a concrete derivation tree, where abstract generalization with* $multi/4$ *is performed on the corresponding abstract conjunction. Then, as a next step in the concrete derivation tree, this conjunction is replaced by* $integers(A, B), multi([filter(C, B, D), filter(E, D, F)]), sift(F, G), len(G, H)$.

Note that this "generalization" actually does not generalize anything. It only brings the information in a form that can be generalized.

The actual generalization happens implicitly in the move to the construction of the next concrete derivation tree. If our conjunction is a leaf of the concrete derivation tree, then the corresponding abstract conjunction is added to the set $\mathcal{A}$. *Let* $\wedge(integers(g, a_4), multi(filter(g, a_{1,i,4}, a_{1,i,5}), \wedge(a_{1,1,4} = a_4), \wedge(a_{1,i+1,4} = a_{1,i,5}), \wedge(a_{1,k,5} = a_7)), sift(a_7, a_6), len(a_6, g))$, *for instance, be the corresponding abstract conjunction that is added to* $\mathcal{A}$. *Then, a new concrete tree is built for a concrete conjunction corresponding to this abstract one.*

In this example, the root of that concrete tree is:

$$\wedge(integers(A, B), multi([filter(C, B, D)|T]), sift(E, F), len(F, G))$$

Finally, we still need to define the counterpart of abstract unfold of $multi/4$ in the concrete tree. To do this, we add the following definition of $multi/1$ to the original program P.

```
multi([H]) ? H.
multi([H|T]) ? H, multi(T).
```

It should be clear that concrete unfolding of concrete $multi/1$ atoms with the above definition for $multi/1$ gives us the desired counterpart of the case split performed in abstract unfold of $multi/1$ if we apply the same bindings used in the abstract unfold.

With the concepts above, we construct a concrete derivation tree, mimicking the steps in the abstract derivation tree – but over the concrete domain – for every conjunction in the set $\mathcal{A}$. Collecting all the resultants from these concrete trees, we get the transformed program. A working Prolog program can be found in Annex (2014). Transformations of permutation sort, graph coloring and lucky numbers are available from the same source.

5 Discussion

In this paper, we have presented an approach to formally analyze the computations, for logic programs, performed under coroutining selection rules, and to compile such computations into new logic programs. On the basis of an example, we have shown that simple coroutines, in which the execution of a single, atomic generator is interleaved with a single, atomic tester, can be successfully analyzed and compiled within the framework of ACPD (Leuschel 2004). These "simple" coroutines essentially correspond to the *strongly regular* logic programs of Vidal (2011), based on Hruza and Stepanek (2003).

To achieve this, we defined an expressive abstract domain, capturing modes, types and aliasing. In the paper, we have focused on the intuitions, more than on the full formalization, as space restrictions would not allow both. However, we have developed the formal definitions for the ordering on the abstract domain, abstract unification, abstract unfold and others. Because the approach – for simple coroutines – fits fully within the ACDP framework, it inherits the correctness results from ACPD.

We have proposed an extension to our abstract domain: the *multi*/4-abstraction. A *multi*/4 atom can represent (sets of) conjunctions of one or more concrete atoms. We have defined abstract unfold and abstract generalisation operations for this abstraction. We have shown, in an example, that this abstraction and these operations allow us to extend ACPD, enabling it to perform a complete analysis, and to compile the more complex coroutines.

On a more general level, our work provides a new, rational reconstruction of the CC-transformation (Bruynooghe et al. 1986), avoiding ad hoc features of the CC approach. In addition, the work presents a new application for ACPD.

As a rule, coroutining improves the efficiency of declarative programs by testing partial solutions as quickly as possible. In addition, a program may become more flexible when the transformation is applied. For instance, a generate-and-test based program for the graph coloring problem which was transformed in the course of this research was originally meant to be called with a ground list of nations and a list of free variables of the correct length. A transformed variant of this program can be run in the same way, but the top-level predicate can also be called with a ground list of nations and a free variable. This is because SLD resolution sends the original program down an infinite branch of the search tree. The transformed program checks results earlier and, as a result, infers that both top-level arguments must be lists of the same size. In this scenario, compiling control transforms an infinite computation into a finite one.

The CC-transformation raised challenges for a number of researchers and a range of compediting transformation and synthesis techniques. A first reformulation of the CC-transformation was proposed in the context of the "programs-as-proofs" paradigm, in Wiggins (1990). It was shown that CC-transformations, to a limited extent, could be formalized in a proof-theoretic program synthesis context.

In Boulanger et al. (1993), CC-transformation was revisited on the basis of a combination of abstract interpretation and constraint processing. This improved the formalization of the technique, but it did not clarify the relation with partial deduction.

The seminal survey paper on Unfold/Fold transformation, Pettorossi and Proietti (1994), showed that basic CC-transformations are well in the scope of Unfold/Fold transformation. In later works (e.g. Pettorossi and Proietti 2002), the same authors introduced list-introduction into the Unfold/Fold framework, whose function is very similar to that of the *multi*/4 abstraction in our approach. Also related to our work are Puebla et al. (1997), providing alternative transformations to improve the efficiency of dynamic scheduling, and Vidal (2011) and

Vidal (2012), which also provide a hybrid form of partial deduction, combining abstract and concrete levels.

As an alternative approach to the one proposed in this paper, one could also apply the first Futamura projection. Given a meta-interpreter implementing a dynamic selection strategy, one could attempt to transform a program P by partially evaluating P and the meta-interpreter. This would require an appropriate analysis, for instance abstract partial deduction.

There are a number of issues that are open for future research. First, we aim to investigate the generality of the $multi/4$ abstraction. Although it seems to work well in a number of examples, we will study more complex ones. We also want to revisit the ACPD framework, in order to extend it to the new abstraction we aim to support. This will involve a new formalization of ACPD, capable of supporting analysis and compilation of coroutines in full generality. This will also formally establish the correctness results for the more general cases, such as the one presented in Sect. 4. Obviously, we also want to have a full implementation of these concepts and to show that the analysis and compilation can be fully automated.

Acknowledgements. We thank the anonymous reviewers for their very useful suggestions.

References

Annex.: Definitions, concepts and elaboration of an example (2014). https://perswww.kuleuven.be/~u0055408/tag/lopstr14.html

Boulanger, D., Bruynooghe, M., De Schreye, D.: Compiling control revisited: a new approach based upon abstract interpretation for constraint logic programs. In: LPE, pp. 39–51 (1993)

Bruynooghe, M.: A practical framework for the abstract interpretation of logic programs. J. Logic Program. **10**(2), 91–124 (1991)

Bruynooghe, M., De Schreye, D., Krekels, B.: Compiling control. In: Proceedings of the 1986 Symposium on Logic Programming. IEEE Society Press, Salt Lake City (1986)

Bruynooghe, M., De Schreye, D., Krekels, B.: Compiling control. J. Logic Program. **6**(1), 135–162 (1989)

De Schreye, D., Glück, R., Jørgensen, J., Leuschel, M., Martens, B., Sørensen, M.H.: Conjunctive partial deduction: foundations, control, algorithms, and experiments. J. Logic Program. **41**(2), 231–277 (1999)

Gallagher, J.P.: Transforming logic programs by specialising interpreters. In: ECAI, pp. 313–326 (1986)

Hruza, J., Stepanek, P.: Speedup of logic programs by binarization and partial deduction. arXiv preprint arXiv:cs/0312026 (2003)

Leuschel, M.: A framework for the integration of partial evaluation and abstract interpretation of logic programs. ACM Trans. Program. Lang. Syst. (TOPLAS) **26**(3), 413–463 (2004)

Lloyd, J.: Foundations of Logic Programming. Springer-Verlag, Berlin (1987)

Lloyd, J.W., Shepherdson, J.C.: Partial evaluation in logic programming. J. Logic Program. **11**(3), 217–242 (1991)

Pettorossi, A., Proietti, M.: Transformation of logic programs: foundations and techniques. J. Logic Program. **19**, 261–320 (1994)

Pettorossi, A., Proietti, M.: The list introduction strategy for the derivation of logic programs. Formal Aspects Comput. **13**(3–5), 233–251 (2002)

Puebla, G., de la Banda, M.J.G., Marriott, K., Stuckey, P.J.: Optimization of logic programs with dynamic scheduling.In: ICLP, vol. 97, pp. 93–107 (1997)

Vidal, G.: A hybrid approach to conjunctive partial evaluation of logic programs. In: Alpuente, M. (ed.) LOPSTR 2010. LNCS, vol. 6564, pp. 200–214. Springer, Heidelberg (2011)

Vidal, G.: Annotation of logic programs for independent and-parallelism by partial evaluation. Theor. Pract. Logic Program. **12**(4–5), 583–600 (2012)

Wiggins, G.A.: The improvement of prolog program efficiency by compiling control: a proof-theoretic view. Department of Artificial Intelligence, University of Edinburgh (1990)

http://www.springer.com/978-3-319-17821-9

Logic-Based Program Synthesis and Transformation
24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014. Revised Selected Papers
Proietti, M.; Seki, H. (Eds.)
2015, XII, 333 p. 61 illus., Softcover
ISBN: 978-3-319-17821-9