# *Advances in Analyzing Coroutines by Abstract Conjunctive Partial Deduction*

Vincent Nys

*submitted 29 April 2015; accepted 5 June 2015*

**Abstract**

The current work describes a technique for the analysis of coroutining in Logic Programs. This provides greater insight into the execution of Logic Programs and yields a transformation from programs with nonstandard execution rules to programs with the standard execution rule of Prolog. A technique known as Compiling Control, or CC for short, was used to study these issues 30 years ago, but it lacked the tools to formalize a complete procedure. Abstract Conjunctive Partial Deduction, introduced by Leuschel, provides an appropriate setting to redefine Compiling Control for simple examples. For more elaborate examples, we extend Leuschel's framework with a new operator. Preliminary experiments with the new operator show that a wide range of programs can be analyzed, opening up possibilities for further analysis as well as program optimization.

## 1 Introduction

Since Kowalski's famous equation "Algorithm = Logic + Control" (Kowalski 1979), much research has been done into separating the logic of a program from its control. Ideally, logic programs should be written in a purely declarative style. The control flow aspect would be specified separately to obtain an efficient execution. To this end, many Logic Programming systems, such as SWI Prolog (Wielemaker et al. 2012), Ciao (Bueno et al. 1997) and Gödel (Hill and Lloyd 1994) support *delay* statements. Delay statements allow the programmer to override the default resolution strategy and specify which atoms should be resolved in particular program contexts, e.g. as soon as a program variable becomes ground. The use of delay statements causes atoms to behave as *coroutines*, in that their execution may be suspended and then later resumed, enabling intricate forms of data communication between atoms. However, the use of separate control flow statements brings with it runtime overhead. It also prevents the use of program analysis and program optimization techniques developed for logic programs under a left-to-right execution rule.

A technique known as Compiling Control (Bruynooghe et al. 1989) allows for the separation of logic and control, while avoiding the drawbacks of delay statements. The technique takes a source program, $P$, and synthesizes a program $P'$, so that computation with $P'$ under left-to-right execution mimics the computation with $P$ under a non-standard selection rule. To do this, Compiling Control (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 under a (non-standard) selection rule. The query pattern is expressed in terms of a combination of type, mode and aliasing information. 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, the resolvents, are generated. Each clause synthesizes the computation in some branch of the trace tree and 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.

Despite this, CC is a rather ad hoc approach. The connection between symbolic values and values in a concrete program was never made explicit. Furthermore, program-specific proofs were required to show that the trace tree represented all possible concrete computations. The aim of this research is to reformulate CC in a way that allows for a rigorous analysis of its correctness and completeness. Since the original work on CC, several important advances in program analysis and transformation have been made:

- General frameworks for Abstract Interpretation in Logic Programming were developed. Abstract Interpretation can formalize the notion of a "symbolic computation" used in CC. Notably, it defines an abstraction function and a concretization function to relate abstract program executions to concrete executions.
- Partial Deduction of Logic Programs was developed. Partial Deduction seems particularly close to CC, in that it builds a finite number of finite execution trees from a partially instantiated input. Like CC, it also synthesizes new programs from these finite execution trees which respect the semantics of the source program. The correctness and completeness of partial deduction were formally proven by Lloyd and Shepherdson (Lloyd and Shepherdson 1991). However, standard Partial Deduction assumes the roots of its execution trees to be atoms, while coroutines are more effectively analyzed together.
- Conjunctive Partial Deduction lifts the restriction that execution trees must start from atoms. This allows it to analyze conjunctions, preserving interactions which cannot be captured by Partial Deduction.
- Abstract Conjunctive Partial Deduction (Leuschel 2004), or ACPD for short, brings the features of the above techniques together. It provides an extension of (Conjunctive) Partial Deduction in which the analysis is based on Abstract Interpretation, rather than on concrete evaluation.

Our aim is to recast CC as a form of Abstract Conjunctive Partial Deduction. Such a reformulation has two major advantages: First, it provides a framework in which the execution of coroutining Logic Programs can be better understood. Second, it enables a transformation to from programs with delay statements to pure logic programs, allowing for additional analysis and optimization. We show that, for simple examples, recasting the approach is possible. For more complex examples, an extension to ACPD is required to overcome one of the restrictions in its original formulation.

## 2 Related work

The CC-transformation raised challenges for a number of researchers and a range of competing 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), the 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* 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.

## 3 Concepts

In order to illustrate the use of ACPD for the analysis of coroutines, we require a few conventions, as well as an abstract domain.

### *3.1 Conventions*

Given a program $P$, $Fun_P$ and $Pred_P$ respectively denote the sets of all functors and predicate symbols in the language underlying $P$. $Term_P$ will denote the set of all terms belonging to the concrete domain. $Atom_P$ denotes the set of all atoms which can be constructed from $Pred_P$ and $Term_P$. We refer to the set of conjunctions of atoms from $Atom_P$ as $ConAtom_P$.

### *3.2 Abstract Domain*

The abstract domain consists of two types of new constant symbols: $a_i$ and $g_j$, $i \in \mathbb{N}_0$ and $j \in \mathbb{N}_0$. The basic intuition for the symbols $a_i$ is that they represent any term of the concrete language. The symbols $g_j$ denote any ground term in the concrete language.

The subscripts $i$ and $j$ in $a_i$ and $g_j$ are used to represent aliasing. If an abstract term, abstract atom or abstract conjunction of atoms contains $a_i$ or $g_j$ 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 respectively occupied by $a_i$ and $g_j$. If an abstract term contains subterms $a_k$ and $g_k$, where $k$ is some element of $\mathbb{N}_0$, this has no particular meaning: aliasing can only be specified between abstract symbols of the same type. For instance, the abstract conjunction $perm(g_1, a_1), ord(a_1)$ denotes the concrete conjunctions $\{perm(t_1, t_2), ord(t_2) | t_1, t_2 \in Term_P, t_1 \text{ is ground}\}$.

Based on these two types of abstract constants, there is a set of abstract terms $ATerm_P$, consisting of terms which can be constructed from the abstract constants 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 $AAtom_P$.

## 4 Examples

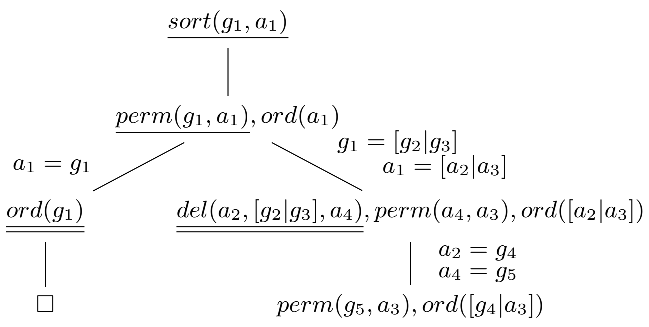
The complete formalization of our approach requires a sizeable set of abstract operations and is outside the scope of this paper. Because the effects of these operations can be readily understood, we will simply illustrate the analysis using two examples. The first of these, permutation sort, fits within Leuschel's original framework. The second, sameleaves, does not. However, an extension to Leuschel's framework makes it possible to deal with a larger class of programs to which sameleaves belongs.

### *4.1 Permutation sort*

Listing 1 contains a Prolog implementation of permutation sort. It defines the sorting operation as a random permutation, followed by a test to see whether the result is ordered. Assume an initial *sort*/2 query in which the first argument is ground. The program then generates a complete permutation of the ground argument and subsequently checks if the

```
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]).
```

Listing 1: Prolog implementation of permutation sort



$\underline{sort(g_1, a_1)}$
$\underline{perm(g_1, a_1)}, ord(a_1)$
$a_1 = g_1$
$g_1 = [g_2|g_3]$
$a_1 = [a_2|a_3]$
$\underline{\underline{ord(g_1)}}$
$\underline{\underline{del(a_2, [g_2|g_3], a_4)}}, perm(a_4, a_3), ord([a_2|a_3])$
$a_2 = g_4$
$a_4 = g_5$
$\square$
$perm(g_5, a_3), ord([g_4|a_3])$

Figure 1: Abstract tree for $sort(g_1, a_1)$

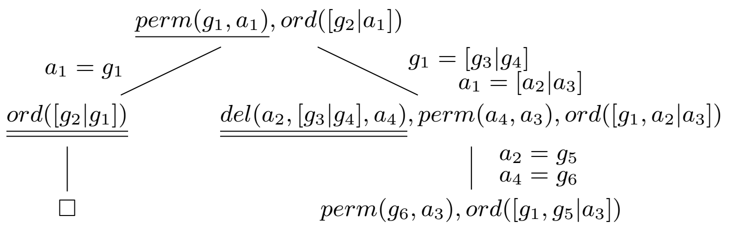
result is ordered. A more efficient approach — though still not efficient in an absolute sense — is to interleave the generation of the permutation with the check for ordering. Each time an element is added to the permutation, it can be compared to its predecessor (if any). If the check fails, there is no point in completing the permutation.

ACPD requires a top-level abstract atom (or conjunction) to start the transformation. Let $sort(g, a_1)$ be this atom. As in standard partial deduction, we initialize a set of analyzed roots $\mathcal{A}$ as the singleton set of the top-level query, i.e. $\{sort(g, a_1)\}$.

Next, we construct a finite number of finite, ACPD derivation trees for abstract (conjunctions of) atoms. The construction of these trees relies on operations for abstract unification and abstract resolution.

To specify the control flow, we assume an "oracle" which decides on the selection rule applied in the abstract derivation trees. This oracle has three functions:

- to decide whether an obtained goal should be unfolded further, or whether it should be kept residual (to be split and added to $\mathcal{A}$).



$\underline{perm(g_1, a_1)}, ord([g_2|a_1])$
$a_1 = g_1$
$g_1 = [g_3|g_4]$
$a_1 = [a_2|a_3]$
$\underline{\underline{ord([g_2|g_1])}}$
$\underline{\underline{del(a_2, [g_3|g_4], a_4)}}, perm(a_4, a_3), ord([g_1, a_2|a_3])$
$a_2 = g_5$
$a_4 = g_6$
$\square$
$perm(g_6, a_3), ord([g_1, g_5|a_3])$

Figure 2: Abstract tree for $perm(g_1, a_1), ord([g_2|a_1])$

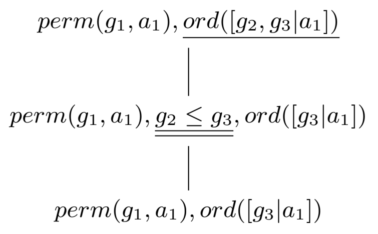

$perm(g_1, a_1), \underline{ord([g_2, g_3|a_1])}$
$perm(g_1, a_1), \underline{\underline{g_2 \le g_3}}, ord([g_3|a_1])$
$perm(g_1, a_1), ord([g_3|a_1])$

Figure 3: Abstract tree for $perm(g_1, a_1), ord([g_2, g_3|a_1])$

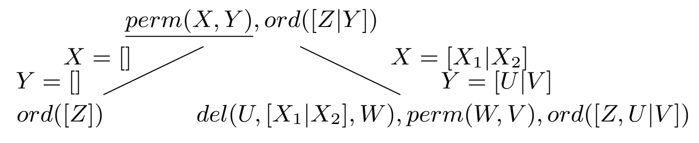

$\underline{perm(X, Y)}, ord([Z|Y])$
$X = []$
$Y = []$
$ord([Z])$
$X = [X_1|X_2]$
$Y = [U|V]$
$del(U, [X_1|X_2], W), perm(W, V), ord([Z, U|V])$

Figure 4: Concrete tree corresponding to Figure 2

- to decide which atom of the current goal should be selected for unfolding.
- to decide whether to "fully evaluate" a selected atom.

The third 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 our setting, however, we want to know the effect that solving the atom has on the remainder of the goal. Therefore, we assume that a full 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. In (Vidal 2011), a similar functionality is integrated in a hybrid approach to conjunctive partial deduction.

In Figures 1 through 3, 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. These are indicated next to the branches of the derivation trees.

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 Figure 1, ACPD adds the abstract conjunction $perm(g_5, a_3), ord([g_4|a_3])$ to $\mathcal{A}$. ACPD starts a new tree for (a renaming of) this atom. This tree is shown in Figure 2.

The tree looks similar to the one in Figure 1 without its root. The main difference is that, in the residual leaf of the second tree, the *ord* atom now has a list argument with two $g$ elements. 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 $perm(g_1, a_1), ord([g_2, g_3|a_1])$, shown in Figure 3.

In Figure 3, the leaf $perm(g_1, a_1), ord([g_3|a_1])$ is a renaming of $perm(g_1, a_1), ord([g_2|a_1])$, which is already contained in $\mathcal{A}$. Therefore, ACPD terminates the analysis, concluding $\mathcal{A}$ covers all possible roots for $\mathcal{A} = \{sort(g_1, a_1), \wedge(perm(g_1, a_1), ord([g_2|a_1])), \wedge(perm(g_1, a_1), ord([g_2, g_3|a_1]))\}$.

In concrete conjunctive partial deduction, the analysis phase would now be completed. In ACPD, however, we need an additional step. In the abstract 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 by resolving the same clauses, but now using concrete terms and concrete unification.

We only show one of these concrete derivation trees in Figure 4.

### *4.2 Sameleaves*

```
sameleaves(T1,T2) :- collect(T1,T1L),collect(T2,T2L),eq(T1L,T2L).
eq([],[]).
eq([H|T1],[H|T2]) :- eq(T1,T2).
collect(node(X),[X]).
collect(tree(L,R),C) :- collect(L,CL), collect(R,CR), append(CL,CR,C).
append([],L,L).
append([H|T],L,[H|TR]) :- append(T,L,TR).
```

Listing 2: Prolog implementation of Sameleaves

Listing 2 shows a program which cannot be analyzed using the technique outlined above. Assume a top-level query of the form $sameleaves(g_1, g_2)$. Trying to analyze this program in the same way as the permutation sort program leads to an infinite derivation. This is due to the fact that trees may be nested to an arbitrary depth. Adding each newly derived conjunction to $\mathcal{A}$ would prevent the procedure from finding a fixpoint for $\mathcal{A}$. We could solve this by cutting the goal into two smaller conjunctions and adding these to $\mathcal{A}$. However, all these atoms behave as generators or testers and depend on each other. By splitting the conjunction, we would no longer be able to analyze the coroutining behaviour.

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 any instantiation of $acon$ is also 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 conjunctions with a growing number of atoms by an abstract conjunct. Therefore, we need to extend ACPD. We introduce a new abstraction, *multi*, which makes it possible to represent growing conjunctions with a repeating internal structure.

To specify the relationship of an abstracted set of conjunctions with the surrounding context, as well as its internal structure, we introduce the "parameterized renaming". The parameterized renaming of a conjunction $C$ is obtained by replacing every $a_j$ or $g_k$ occurring in $C$ by its respective parameterized naming, $a_{Id,i,j}$ or $g_{Id,i,k}$. Here, $Id$ is a unique identifier for the *multi* abstraction with which the renaming will be associated, as several *multi* abstractions may be needed to complete the analysis. The $i$ is symbolic and is used to specify data shared between abstracted conjunctions. Finally, $j$ and $k$ fulfill the same function as an aliasing index, but within the local scope of a *multi* abstraction.

The multi abstraction is then defined as a construct of the form $multi(p(C), First, Consecutive, Last)$, where:

- $p(C)$ is the parameterized conjunction for some $C \in AConAtom_P$.
- $First$ is a conjunction of equalities $a_{Id,1,j} = b_j$ and $g_{Id,1,k}$, where $b_j$ and $b_k$ occur outside the abstraction and all left-hand sides of the equalities are distinct.
- $Consecutive$ is a conjunction of equalities $a_{Id,i+1,j} = a_{Id,i,j'}$ and $g_{Id,i+1,k} = g_{Id,i,k'}$, where all left-hand sides of the equalities are distinct.
- $Last$ is a conjunction of equalities $a_{Id,\mathcal{L},j} = b_j$ and $g_{Id,\mathcal{L},k} = b_k$, where $b_j$ and $b_k$ occur outside the abstraction occurs and all left-hand sides of the equalities are distinct.

Inside this abstraction, the parameterized renaming $p(C)$ represents an arbitrary number of renamings of $C$. The bindings are used to specify how specific conjunctions captured

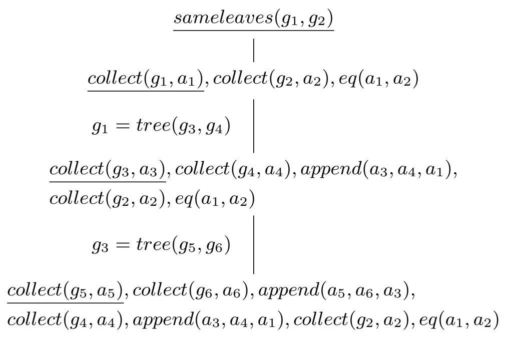

$\underline{sameleaves(g_1, g_2)}$
$\underline{collect(g_1, a_1)}, collect(g_2, a_2), eq(a_1, a_2)$
$g_1 = tree(g_3, g_4)$
$\underline{collect(g_3, a_3)}, collect(g_4, a_4), append(a_3, a_4, a_1),$
$collect(g_2, a_2), eq(a_1, a_2)$
$g_3 = tree(g_5, g_6)$
$\underline{collect(g_5, a_5)}, collect(g_6, a_6), append(a_5, a_6, a_3),$
$collect(g_4, a_4), append(a_3, a_4, a_1), collect(g_2, a_2), eq(a_1, a_2)$

Figure 5: Infinite derivation strategy for sameleaves

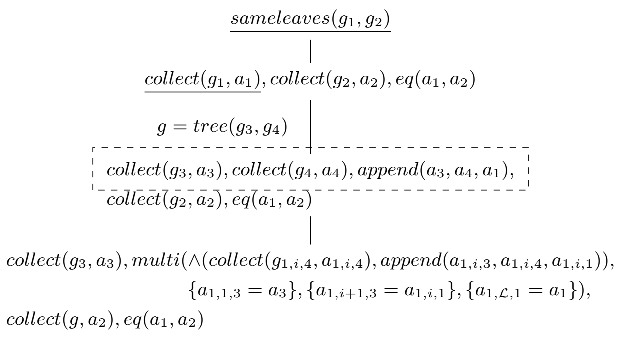

$\underline{sameleaves(g_1, g_2)}$
$\underline{collect(g_1, a_1)}, collect(g_2, a_2), eq(a_1, a_2)$
$g = tree(g_3, g_4)$
$collect(g_3, a_3), collect(g_4, a_4), append(a_3, a_4, a_1),$
$collect(g_2, a_2), eq(a_1, a_2)$
$collect(g_3, a_3), multi(\wedge(collect(g_{1,i,4}, a_{1,i,4}), append(a_{1,i,3}, a_{1,i,4}, a_{1,i,1})),$
$\{a_{1,1,3} = a_3\}, \{a_{1,i+1,3} = a_{1,i,1}\}, \{a_{1,\mathcal{L},1} = a_1\}),$
$collect(g, a_2), eq(a_1, a_2)$

Figure 6: Generalization to *multi*

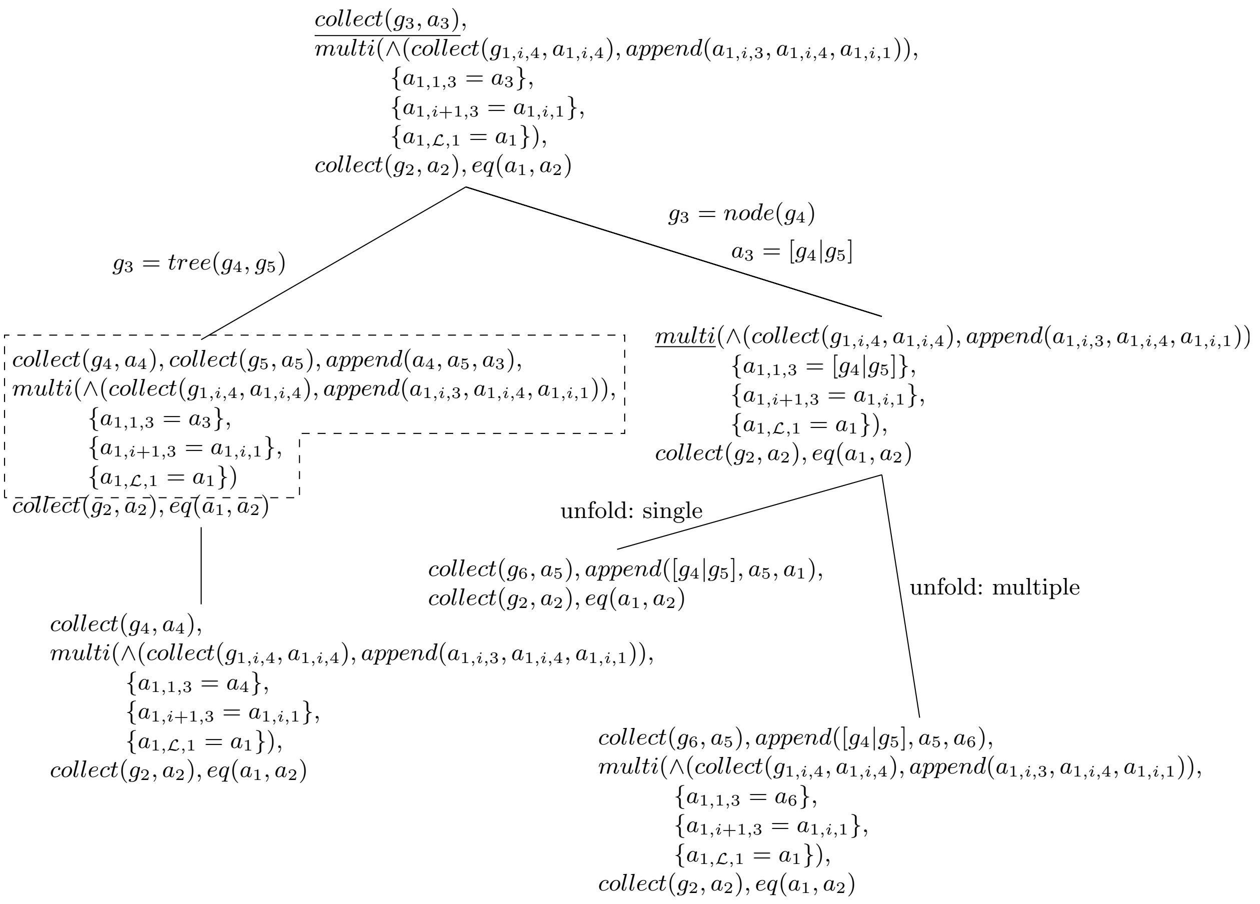

$\underline{collect(g_3, a_3)},$
$\underline{multi}(\wedge(collect(g_{1,i,4}, a_{1,i,4}), append(a_{1,i,3}, a_{1,i,4}, a_{1,i,1})),$
$\{a_{1,1,3} = a_3\},$
$\{a_{1,i+1,3} = a_{1,i,1}\},$
$\{a_{1,\mathcal{L},1} = a_1\}),$
$collect(g_2, a_2), eq(a_1, a_2)$
$g_3 = node(g_4)$
$a_3 = [g_4|g_5]$
$g_3 = tree(g_4, g_5)$
$collect(g_4, a_4), collect(g_5, a_5), append(a_4, a_5, a_3),$
$multi(\wedge(collect(g_{1,i,4}, a_{1,i,4}), append(a_{1,i,3}, a_{1,i,4}, a_{1,i,1})),$
$\{a_{1,1,3} = a_3\},$
$\{a_{1,i+1,3} = a_{1,i,1}\},$
$\{a_{1,\mathcal{L},1} = a_1\})$
$collect(g_2, a_2), eq(a_1, a_2)$
$\underline{multi}(\wedge(collect(g_{1,i,4}, a_{1,i,4}), append(a_{1,i,3}, a_{1,i,4}, a_{1,i,1}))$
$\{a_{1,1,3} = [g_4|g_5]\},$
$\{a_{1,i+1,3} = a_{1,i,1}\},$
$\{a_{1,\mathcal{L},1} = a_1\}),$
$collect(g_2, a_2), eq(a_1, a_2)$
unfold: single
unfold: multiple
$collect(g_6, a_5), append([g_4|g_5], a_5, a_1),$
$collect(g_2, a_2), eq(a_1, a_2)$
$collect(g_4, a_4),$
$multi(\wedge(collect(g_{1,i,4}, a_{1,i,4}), append(a_{1,i,3}, a_{1,i,4}, a_{1,i,1})),$
$\{a_{1,1,3} = a_4\},$
$\{a_{1,i+1,3} = a_{1,i,1}\},$
$\{a_{1,\mathcal{L},1} = a_1\}),$
$collect(g_2, a_2), eq(a_1, a_2)$
$collect(g_6, a_5), append([g_4|g_5], a_5, a_6),$
$multi(\wedge(collect(g_{1,i,4}, a_{1,i,4}), append(a_{1,i,3}, a_{1,i,4}, a_{1,i,1})),$
$\{a_{1,1,3} = a_6\},$
$\{a_{1,i+1,3} = a_{1,i,1}\},$
$\{a_{1,\mathcal{L},1} = a_1\}),$
$collect(g_2, a_2), eq(a_1, a_2)$

Figure 7: Generalization of *multi* (left branch) and unfolding of *multi* (right branches)

by the abstraction are bound to their surrounding environment. Specifically, the *First* bindings indicate how information is shared between the environment and the first conjunction captured. The subscript "1" used in this set of equalities is a reminder that these equalities pertain to the first captured conjunction. The *Consecutive* bindings indicate which values are shared between consecutive conjunctions captured by the abstraction. Because the abstraction represents a linear structure, we can relate values in some conjunction $i$ to those in its successor, $i+1$. The *Last* bindings are similar to the *First* bindings, but pertain to the last rather than the first captured conjunction. The subscript "$\mathcal{L}$" is also a reminder of the conjunction to which these bindings apply.

The problem of a growing recursion is illustrated in Figure 5 and generalization using the *multi* abstraction is shown in Figure 6. Because the subconjunction $collect(g_3, a_3)$, $collect(g_4, a_4), append(a_3, a_4, a_5)$ obeys a pattern common to all growing conjunctions, the recursion can be detected algorithmically. After generalization, the leaf of Figure 6 is added to $\mathcal{A}$ in the standard way. The left branch of Figure 7 demonstrates how further unfolding followed by generalization can lead back to a renaming of this state. The first branch to the right shows how the bindings pass information between the *multi* abstraction and its environment: When $a_3$ is bound to $[g_4|g_5]$ in the environment (with $g_5$ the most specific abstraction of the empty list), it also becomes bound in the *First* bindings. The second level of branching shows how the *multi* abstraction is unfolded: In the first case, it represents a single conjunction to which both the *First* and *Last* bindings are applied. In the second, it represents a larger conjunction whose first abstracted subconjunction is moved outside the abstraction. The current *First* bindings are applied to this subconjunction and new *First* bindings are then computed based on the current *Consecutive* bindings.

The complete analysis of sameleaves is considerably longer and involves some interesting patterns of computation. However, the operations shown are enough find a fixpoint for $\mathcal{A}$ and complete the analysis.

## 5 Current status

We first presented an earlier version of this research at LOPSTR 2014 and a corresponding paper was published in (De Schreye et al. 2014). An extended journal paper was submitted at the end of June, 2015. The latter version contains the formalizations for most of the operations illustrated by example in this paper. It details an earlier variant of the *multi* abstraction discussed here, which could not yet represent the growing sequence of atoms found in the sameleaves program. The variant discussed here can do this, because it abstracts linearly growing sequences of abstract conjunctions, rather than linearly growing sequences of atoms. We conjecture that the abstraction is now general enough to deal with any growing sequence of abstract atoms under an instantiation-based computation rule.

We have applied the technique to analyze a range of well-known "toy" problems, notably the permutation sort, graph coloring, prime generator, N-queens and sameleaves example. A manually synthesized standard Prolog program based on this analysis is available for the first four problems. Such a synthesis can also be performed for the sameleaves program. The reason it has not yet been written is simply because the analysis is quite long.

To demonstrate that the approach can be automated, we have developed an implementation which performs the complete analysis phase. It requires the user to specify the decisions made by the oracle: For the selection rule, it uses delay declarations. For the calls which require a full evaluation, it requires the user to map an atom to be solved onto an output atom. Also, the analyzer requires the user to specify which conjunctions should be generalized into a *multi* abstraction. This is done using a grammar of conjunctions, with each conjunction produced by the grammar a possible instantiation of the *multi* abstraction. With this information, the analyzer is capable of performing the complete analysis phase for any example we have found, including the sameleaves program.

## 6 Future work

First and foremost, we want to investigate the generality of the *multi* abstraction. No examples which give rise to nested *multi* abstractions have been examined yet. These are certain to occur in complex, real-world programs. Likewise, use of the *multi* abstraction to represent *disjunctions* as well as conjunctions is a worthwhile avenue for future research. Eventually, we hope to formally characterize the class of programs which can be analyzed using our approach.

We also plan to re-examine ACPD as formulated by Leuschel, in order to fully integrate the *multi* abstraction. In this way, we hope to retain all correctness results of ACPD for the analysis and compilation of coroutines in full generality.

Improvements to the implementation can also be made. We hope to integrate the current prototype with an existing system for abstract interpretation. This should free the user from the need to specify the effects of a full evaluation of abstract atoms. Similarly, we would like to investigate how the grammatical description of growing conjunctions can be derived automatically. With both improvements, the entire analysis and synthesis could be done algorithmically using only the program source code and delay statements. We believe this will open up interesting new paths for further program analysis and optimization.

## References

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

Bruynooghe, M., De Schreye, D., and Krekels, B. 1989. Compiling control. *The Journal of Logic Programming 6,* 1, 135–162.

Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-Garcia, P., and Puebla, G. 1997. The ciao prolog system. *Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM) 95*, 96.

De Schreye, D., Nys, V., and Nicholson, C. 2014. Analysing and compiling coroutines with abstract conjunctive partial deduction. In *Logic-Based Program Synthesis and Transformation.* Springer, 21–38.

Hill, P. and Lloyd, J. W. 1994. *The Gödel programming language.* MIT press.

Kowalski, R. 1979. Algorithm=logic+ control. *Communications of the ACM 22,* 7, 424–436.

Leuschel, M. 2004. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS) 26,* 3, 413–463.

Lloyd, J. W. and Shepherdson, J. C. 1991. Partial evaluation in logic programming. *The Journal of Logic Programming 11,* 3, 217–242.

Pettorossi, A. and Proietti, M. 1994. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming 19*, 261–320.

Pettorossi, A. and Proietti, M. 2002. The list introduction strategy for the derivation of logic programs. *Formal aspects of computing 13,* 3-5, 233–251.

Puebla, G., de la Banda, M. J. G., Marriott, K., and Stuckey, P. J. 1997. Optimization of logic programs with dynamic scheduling. In *ICLP*. Vol. 97. 93–107.

Vidal, G. 2011. A hybrid approach to conjunctive partial evaluation of logic programs. In *Logic-Based Program Synthesis and Transformation*, M. Alpuente, Ed. Lecture Notes in Computer Science, vol. 6564. Springer Berlin Heidelberg, 200–214.

Vidal, G. 2012. Annotation of logic programs for independent and-parallelism by partial evaluation. *Theory and Practice of Logic Programming 12,* 4-5, 583–600.

Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. 2012. Swi-prolog. *Theory and Practice of Logic Programming 12,* 1-2, 67–96.

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