Improving Program Analyses by Structure Untupling

Michael Codish [a], Kim Marriott [b] and Cohavit Taboch [a]

[a] *Department of Mathematics and Computer Science, Ben-Gurion University of the Negev, Beer-Sheva Israel, E-mail:* mcodish@cs.bgu.ac.il

[b] *School of Computer Science and Software Engineering, Monash University, Clayton, Victoria, Australia, E-mail:* marriott@cs.monash.edu.au

Abstract

It is well-known that adding structural information to an analysis domain can increase the precision of the analysis with respect to the original domain. This paper presents a program transformation based on untupling and specialisation which can be applied to upgrade (logic) program analysis by providing additional structural information. It can be applied to (almost) any type of analysis and in conjunction with (almost) any analysis framework for logic programs. The approach is an attractive alternative to the more complex Pat($\mathcal{R}$) construction which automatically enhances an abstract domain $\mathcal{R}$ (in the context of abstract interpretation) with structural information.

1 Introduction

The practical utility of enhancing an abstract domain to include structural information has long been recognised. One reason is that structural information is of intrinsic interest. More usually, it is because such information has the potential to increase the precision of many other types of analysis. In the context of logic programming, precision can be improved for two reasons:

The first reason is related to *pruning*. A logic program analysis which tracks structural information (in addition to information about some property of interest) can ignore clauses whose heads are not compatible with the structural information in a particular call pattern. By ignoring these "irrelevant" clauses,

⋆ This work was carried out while the first author was visiting the Department of Computer Science and Engineering at the University of Melbourne.

precision may be improved with respect to the property of interest. The second reason is related to *sub-structure* information and is more subtle. When data entities are passed around inside structures they are described collectively. However, if the structure of the (run time) data entities corresponding to a syntactic program point is invariant under all possible executions then program analysis can provide more detailed information for this point in terms of the invariant sub-structures. A common example is the analysis of logic programs involving difference lists. To provide reasonable precision, an analysis will typically need to find a description which focuses separately on the head and on the tail of a difference list. We shall elaborate on this in Example 1 (below).

One straightforward technique for adding structural information to an abstract domain $\mathcal{R}$ is to consider its cross product with a description domain for structural information, such as the **depth**(**k**) domain described in [17] . Analyses in the new domain will give information about structure and allow pruning, but it will not provide sub-structure information since descriptions from $\mathcal{R}$ will still apply only to procedure arguments.

A more sophisticated approach in which an abstract domain $\mathcal{R}$ is enhanced with structural information to give a new abstract domain `Pat(`$\mathcal{R}$`)` is proposed in [7] and further investigated in [1]. An analysis in `Pat(`$\mathcal{R}$`)` provides structural information and improves the precision of $\mathcal{R}$ information in both of the ways described above. The main contribution of `Pat(`$\mathcal{R}$`)` (and the main difficulty) is in the way it addresses the second point: providing enhanced $\mathcal{R}$ information about sub-structures in argument positions. Essentially it does this by introducing a pattern component to descriptions and maintaining $\mathcal{R}$ information on all sub-structures of this component. This is not an easy task as it requires a high level and abstract specification of pattern information which is general enough to interact with any domain $\mathcal{R}$ and its operations. The advantage of the `Pat(`$\mathcal{R}$`)` approach is that it is generic: once the domain $\mathcal{R}$ has been defined and implemented (and supports a certain set of operations) no further design effort is required to provide an analyser for `Pat(`$\mathcal{R}$`)`. Another contribution of the work described in [7] is the experimental evaluation which demonstrates how keeping structural information can improve precision of program analyses.

One disadvantage of `Pat(`$\mathcal{R}$`)` is that its presentation is complex. Another drawback is that it is expensive. Results from [6] suggest that it slows down analysis times for the abstract domain **Pos** by a factor of between 5 and 109 for the set of benchmarks considered. This is especially distressing as the results indicate an improvement in precision for only 2 of the 11 programs considered [1] .

[1] The experiments in [6] cover 12 programs, however two of these are slight variations of the same program which behave exactly the same in a **Pos** analysis.

Perhaps for these reasons, and in spite of the recognised importance of structural information, Pat($\mathcal{R}$) is currently only implemented in the two analysis frameworks GAIA and China corresponding to [7] and [1] respectively.

This paper provides a new solution to the difficult part of the problem, namely how to use information about sub-structures to enhance an analysis over $\mathcal{R}$. Rather than enhancing the abstract domain $\mathcal{R}$ with pattern information, the program is enhanced. First, a pattern analysis is applied to the program. Then, a transformation which portrays the derived information is applied. This involves two steps: specialisation and untupling. Finally, the enhanced program is analysed over the original domain $\mathcal{R}$. The effect is an improvement in the precision of the $\mathcal{R}$ analysis similar to that obtained in the comparable part of a Pat($\mathcal{R}$) analysis.

There are three main advantages to our approach. First, it is simple. Both in its formal specification as well as in its implementation. Second, it is efficient. By separating the pattern analysis from the $\mathcal{R}$ analysis we can focus on providing extra $\mathcal{R}$ information only at those positions where sub-structures are invariant throughout the entire analysis. In particular, when the pattern analysis does not detect any such sub-structures then there is no untupling and we will pay no additional cost during the subsequent $\mathcal{R}$ analysis. Finally, our approach is widely applicable. Since it is described as a program transformation, there is little overhead associated with its integration within (almost) any program analysis framework.

The approach described in this paper is based on the notion of *more specific logic programs* introduced in [14,13] where the authors also suggest its application to program analysis (although they did not detail the untupling transformation). In this approach, the clauses of a program are specialised with respect to the structures it manipulates. The resulting program is often more efficient than the original program and preserves its success patterns. We are not concerned with the efficiency of most specific logic programs but rather view the transformation as a means to obtain more precise analyses of success patterns for the original program. Although the transformed program does not preserve the call patterns of the original program, it is straightforward to provide improved call pattern analysis based on the improved success pattern analysis. In [14,13], the authors are also concerned with the preservation of the termination behaviour of a most specific logic program. We illustrate the application of these results to improve the precision of a termination analysis for logic programs.

2 Preliminaries

We assume a familiarity with the standard definitions and notation for logic programs [12] except that we make use of the non-ground **s**-semantics defined in [9]. We address the topic of program analysis but make no assumptions on a particular framework of analysis. Instead our technique is specified as a semantics preserving program transformation. Analysing a transformed program gives results which can then be interpreted with respect to the original program. We consider the analysis of success patterns, call patterns and termination behaviour. To clarify notion: By a *success pattern* for the program $\mathbf{P}$ we mean an element of the set

$$\mathbf{SS}(\mathbf{P}) = \left\{ \vartheta(\mathbf{p}(\bar{\mathbf{x}})) \;\middle|\; \begin{array}{l} \bar{\mathbf{x}} \text{ is a tuple of distinct variables} \\ \vartheta \text{ is a computed answer for } \mathbf{p}(\bar{\mathbf{x}}) \end{array} \right\};$$

By a *call pattern* we mean a selected atom in an SLD derivation (for the sake of simplicity we assume Prolog's leftmost selection rule); By *termination* we refer to universal termination (assuming Prolog's leftmost selection rule).

The transformation we propose relies on a simple program analysis for pattern information which is best described as an abstract interpretation [8] of the **s**-semantics [9]. This semantics is specified as the least fixed point of an immediate consequences operator which manipulates sets of (equivalence classes of) possibly non-ground atoms (modulo renaming). The **s**-semantics of a program $\mathbf{P}$ is (a bottom-up specification of) the set $\mathbf{SS}(\mathbf{P})$.

As an example program analysis, for which we illustrate an improvement when using our technique, we consider the application of abstract interpretation for the analysis of groundness dependencies using the abstract domain of positive Boolean functions (**Pos**) [4,5,18,16]. In **Pos** a formula of the form $\mathbf{X} \rightarrow \mathbf{Y}$ describes a program state in which the variable $\mathbf{Y}$ is bound to a term which will become ground if the variable $\mathbf{X}$ becomes bound to a ground term.

3 The Untupling Phase

If all of the (run time) calls to a predicate $\mathbf{p/n}$ in a logic program $\mathbf{P}$ are instances of a given structure then $\mathbf{P}$ can be transformed by *untupling* that structure. The untupling transformation enables improved analysis of substructures of the original program because by moving them to arguments they can be better described.

A sufficient condition for the calls to a predicate to be an instance of a given structure is that all (syntactic) calls in the program are instances of that

structure. As we shall see, this simple condition, when combined with program specialisation, is quite powerful.

Definition 1 (untupling) *Let* $\mathbf{P}$ *be a program such that all calls to a predicate* $\mathbf{p/n}$ *(in the clause bodies and in the external calls) are instances of an atom* $\mathbf{p}(\bar{\mathbf{s}})$ *and let* $\bar{\mathbf{v}}$ *be the sequence of the* $\mathbf{k}$ *distinct variables in* $\mathbf{p}(\bar{\mathbf{s}})$ *(in order of first occurrence). Then the* untupling *of* $\mathbf{p/n}$ *with respect to* $\mathbf{p}(\bar{\mathbf{s}})$ *is defined as follows:*

(1) A new clause $\mathbf{p}(\bar{\mathbf{s}}) \leftarrow \mathbf{p}'(\bar{\mathbf{v}})$ *where* $\mathbf{p}'$ *is a fresh predicate symbol not occurring in* $\mathbf{P}$*, is introduced.*
(2) Each clause for $\mathbf{p/n}$ *of the form* $\mathbf{p}(\bar{\mathbf{t}}) \leftarrow \mathbf{body}$ *in the original program* $\mathbf{P}$ *is replaced by a clause* $\vartheta(\mathbf{p}'(\bar{\mathbf{v}}') \leftarrow \mathbf{body})$ *with* $\vartheta = \mathbf{mgu}(\bar{\mathbf{s}}', \bar{\mathbf{t}})$ *where* $\mathbf{p}(\bar{\mathbf{s}}') \leftarrow \mathbf{p}'(\bar{\mathbf{v}}')$ *is a fresh copy of* $\mathbf{p}(\bar{\mathbf{s}}) \leftarrow \mathbf{p}'(\bar{\mathbf{v}})$*. If* ϑ *does not exist then the clause is redundant and is omitted from the program.*
(3) All calls to $\mathbf{p/n}$ *in the clauses of* $\mathbf{P}$ *are unfolded to corresponding calls to* $\mathbf{p}'/\mathbf{k}$ *(using the clause* $\mathbf{p}(\bar{\mathbf{s}}) \leftarrow \mathbf{p}'(\bar{\mathbf{v}})$*).*

The clauses introduced by Rule 1 of Definition 1 (those of the form $\mathbf{p}(\bar{\mathbf{s}}) \leftarrow \mathbf{p}'(\bar{\mathbf{v}})$) are referred to as "entry points". They might seem redundant in the transformed program. However, they are useful for interpreting the analysis of the transformed program in terms of the original program.

The untupling transformation results in a new program which is equivalent in its answers (for the original predicates) and has the same termination behaviour. This follows because a resolution step for a call to $\mathbf{p/n}$ with a clause $\mathbf{C}$ in the original program has the same affect as two resolution steps in the transformed program: the first using the entry point clause for $\mathbf{p/n}$ and the second using the transformed clause corresponding to $\mathbf{C}$.

Example 1 *Consider the following program which specifies a preorder traversal on the nodes of a binary tree using a difference list structure.*

$$\mathbf{preorder}(\mathbf{T}, \mathbf{Xs}) \leftarrow \mathbf{preorder_dl}(\mathbf{T}, \mathbf{Xs} - [\,]).$$

$$\mathbf{preorder_dl}(\mathbf{nil}, \mathbf{X} - \mathbf{X}).$$
$$\mathbf{preorder_dl}(\mathbf{tree}(\mathbf{L}, \mathbf{X}, \mathbf{R}), [\mathbf{X}|\mathbf{Xs}] - \mathbf{Zs}) \leftarrow$$
$$\quad \mathbf{preorder_dl}(\mathbf{L}, \mathbf{Xs} - \mathbf{Ys}),$$
$$\quad \mathbf{preorder_dl}(\mathbf{R}, \mathbf{Ys} - \mathbf{Zs}).$$

Assuming that there are no external calls to the predicate $\mathbf{preorder_dl}/2$*, a simple syntactic check reveals that all calls to the predicate* $\mathbf{preorder_dl}/2$ *are instances of* $\mathbf{preorder_dl}(\mathbf{A}, \mathbf{B} - \mathbf{C})$*. Hence untupling gives the following*

program:

$\%$ **entry points** :
$\mathbf{preorder(A, B)} \leftarrow \mathbf{preorder'(A, B)}.$
$\mathbf{preorder_dl(A, B - C)} \leftarrow \mathbf{preorder_dl'(A, B, C)}.$

$\%$ **untupled clauses** :
$\mathbf{preorder'(T, Xs)} \leftarrow \mathbf{preorder_dl'(T, Xs, [\,])}.$

$\mathbf{preorder_dl'(nil, X, X)}.$
$\mathbf{preorder_dl'(tree(L, X, R), [X|Xs], Zs)} \leftarrow$
$\quad \mathbf{preorder_dl'(L, Xs, Ys)},$
$\quad \mathbf{preorder_dl'(R, Ys, Zs)}.$

To demonstrate the advantage of untupling for program analysis consider the following example.

Example 2 *We consider a groundness analysis with the domain* **Pos** *for the* **preorder**/2 *relation in Example 1 and its untupled version. A bottom-up* **Pos** *analysis using the original program gives no useful groundness dependencies. However when analysing the untupled version we obtain the following* **Pos** *description for the success patterns of* **preorder**/2*:*

$\mathbf{preorder(A, B)} \leftarrow \ \mathbf{A \leftrightarrow B}$

It is already in the first stage of the analysis, when describing the respective single facts in the two programs, where a difference is made. With the original program, the structure $\mathbf{X - X}$ *(in the fact) gives no* **Pos** *information, while in the untupled program we specify a groundness dependency between the second and third arguments of* **preorder_dl'**/3*.*

Unfortunately, it is rare for the syntactic calls to a predicate in a program to have non-trivial shared structure. The following illustrates this point using a common variant of the program from Example 1. We remove this limitation in the next section.

Example 3 *Consider the following variant of the program from Example 1 which calls an explicit* **append_dl**/3 *relation. Because the recursive calls to* **preorder_dl**/2 *have a free variable in their second arguments, untupling can-*

not be directly used to improve program analysis for this program.

$\mathbf{preorder(T, Xs)} \leftarrow \mathbf{preorder_dl(T, Xs - [\,])}.$

$\mathbf{preorder_dl(nil, X - X)}.$
$\mathbf{preorder_dl(tree(L, X, R), DL)} \leftarrow$
$\quad \mathbf{preorder_dl(L, DL1)},$
$\quad \mathbf{append_dl(DL1, [X|D] - D, DL2)},$
$\quad \mathbf{preorder_dl(R, DL3)},$
$\quad \mathbf{append_dl(DL2, DL3, DL)}.$
$\mathbf{append_dl(A - B, B - C, A - C)}.$

4 The Specialisation Phase

If all of the success patterns for a predicate $\mathbf{p/n}$ in a logic program $\mathbf{P}$ are instances of a given structure then $\mathbf{P}$ can be specialised so that all of the calls to $\mathbf{p/n}$ are instances of that structure. This transformation is described in [13] where the resulting program is termed a "*more specific*" version of $\mathbf{P}$. In particular, the syntactic calls to a predicate $\mathbf{p/n}$ in a more specific version of $\mathbf{P}$ may be more specific and so are more likely to lead to non-trivial untupling hence, facilitating improved program analysis.

Definition 2 (specialisation) *Let* $\mathbf{C} = \mathbf{h} \leftarrow \mathbf{b_1}, \ldots, \mathbf{b_n}$ *be a clause in a program* $\mathbf{P}$*, let* $\mathbf{a_1}, \ldots, \mathbf{a_n}$ *be atoms (renamed apart from each other and from* $\mathbf{C}$*) such that for* $\mathbf{i} = 1..\mathbf{n}$ *any success pattern for* $\mathbf{b_i}$ *is an instance of* $\mathbf{a_i}$ *and let* $\vartheta = \mathbf{mgu}(\langle \mathbf{b_1}, \ldots, \mathbf{b_n} \rangle, \langle \mathbf{a_1}, \ldots, \mathbf{a_n} \rangle)$*. Then* $\mathbf{C}$ *is replaced by* $\vartheta(\mathbf{C})$ *to obtain a specialised version of* $\mathbf{P}$*. If* $\mathbf{mgu}(\langle \mathbf{b_1}, \ldots, \mathbf{b_n} \rangle, \langle \mathbf{a_1}, \ldots, \mathbf{a_n} \rangle)$ *does not exist or if for some* $1 \leq \mathbf{i} \leq \mathbf{n}$*,* $\mathbf{b_i}$ *has no success patterns then* $\mathbf{C}$ *is redundant and does not occur in the specialised version of* $\mathbf{P}$*.*

Example 4 *Consider the program from Example 3 and notice that any success pattern for this program is an instance of one of the following atoms:*

$\mathbf{preorder(A, B)}.$
$\mathbf{preorder_dl(A, B - C)}.$
$\mathbf{append_dl(A - B, B - C, A - C)}.$

Hence, the program can be specialised resulting in:

$$\mathbf{preorder}(\mathbf{T}, \mathbf{Xs}) \leftarrow \mathbf{preorder_dl}(\mathbf{T}, \mathbf{Xs} - [\,]).$$

$$\mathbf{preorder_dl}(\mathbf{nil}, \mathbf{X} - \mathbf{X}).$$
$$\mathbf{preorder_dl}(\mathbf{tree}(\mathbf{L}, \mathbf{X}, \mathbf{R}), \mathbf{H} - \mathbf{T}) \leftarrow$$
$$\quad \mathbf{preorder_dl}(\mathbf{L}, \mathbf{H} - [\mathbf{X}|\mathbf{T}1]),$$
$$\quad \mathbf{append_dl}(\mathbf{H} - [\mathbf{X}|\mathbf{T}1], [\mathbf{X}|\mathbf{T}1] - \mathbf{T}1, \mathbf{H} - \mathbf{T}1),$$
$$\quad \mathbf{preorder_dl}(\mathbf{R}, \mathbf{T}1 - \mathbf{T}),$$
$$\quad \mathbf{append_dl}(\mathbf{H} - \mathbf{T}1, \mathbf{T}1 - \mathbf{T}, \mathbf{H} - \mathbf{T}).$$

$$\mathbf{append_dl}(\mathbf{A} - \mathbf{B}, \mathbf{B} - \mathbf{C}, \mathbf{A} - \mathbf{C}).$$

This more specific version can now be usefully untupled for improved program analysis because of the additional structure in the recursive calls. Untupling results in the following clauses (omitting the entry points for brevity):

$$\mathbf{preorder}'(\mathbf{T}, \mathbf{Xs}) \leftarrow \mathbf{preorder_dl}'(\mathbf{T}, \mathbf{Xs}, [\,]).$$

$$\mathbf{preorder_dl}'(\mathbf{nil}, \mathbf{X}, \mathbf{X}).$$
$$\mathbf{preorder_dl}'(\mathbf{tree}(\mathbf{L}, \mathbf{X}, \mathbf{R}), \mathbf{H}, \mathbf{T}) \leftarrow$$
$$\quad \mathbf{preorder_dl}'(\mathbf{L}, \mathbf{H}, [\mathbf{X}|\mathbf{T}1]),$$
$$\quad \mathbf{append_dl}'(\mathbf{H}, [\mathbf{X}|\mathbf{T}1], \mathbf{T}1),$$
$$\quad \mathbf{preorder_dl}'(\mathbf{R}, \mathbf{T}1, \mathbf{T}),$$
$$\quad \mathbf{append_dl}'(\mathbf{H}, \mathbf{T}1, \mathbf{T}).$$

$$\mathbf{append_dl}'(\mathbf{A}, \mathbf{B}, \mathbf{C}).$$

It is interesting to note that the calls to **append_dl**/3 *are now redundant and can be removed.*

As shown in [13] this type of specialisation preserves the success patterns of the original program but not necessarily its calls and termination behaviour.

5 Deriving Pattern Information

The derivation of pattern information is easily described as an abstract interpretation. In [13] and [15] Marriott *et al.* suggest several techniques to compute more specific versions of a program. For our purposes, a modification of the *singleton abstraction scheme* abstract interpretation of [15] suffices. The difference is that in our context it is appropriate to assume that the underlying semantics is the **s**-semantics [9] instead of Fitting's more complex Kripke-Kleene semantics [10] used in [15]. An advantage of our technique is its simplicity, although some of the other techniques described in [13] will find more specific versions of the program. Like other methods, we cannot

guarantee to find the most specific logic program.

We define an abstract semantics operator $\mathbf{T_P^{msg}}$ in terms of the non-ground immediate consequences operator $\mathbf{T_P}$ of the **s**-semantics. For any set of atoms **I**, our operator is given as

$$\mathbf{T_P^{msg}}(\mathbf{I}) = \sqcup \mathbf{T_P}(\mathbf{I})$$

where $\sqcup$ is a least upper bound which combines the set of atoms for the predicate $\mathbf{p/n}$, replacing them by their (single) most specific generalisation.

It is straightforward to prove that the least fixed point of this operator is finitely computable and provides an approximation of the program's success patterns, in the sense that any success pattern is an instance of an element of the approximation.

Example 5 *Consider the program from Example 3. The analysis of its success patterns results in the description*

$$\mathbf{lfp}(\mathbf{T_P^{msg}}) = \left\{ \begin{array}{l} \mathbf{preorder}(\mathbf{A}, \mathbf{B}). \\ \mathbf{preorder_dl}(\mathbf{A}, \mathbf{B} - \mathbf{C}). \\ \mathbf{append_dl}(\mathbf{A} - \mathbf{B}, \mathbf{B} - \mathbf{C}, \mathbf{A} - \mathbf{C}). \end{array} \right\}$$

6 Combining the Phases

A simple yet powerful technique to enhance almost any program analysis with structural information is obtained by combining the simple analysis for structure and the two transformations described above. First, we specialise the program with respect to the common structures in its success patterns. Next we untuple the calls in this more specific version of the program. Finally, we perform the analysis of interest on this program.

While untupling on its own is of limited usefulness, what gives this combined technique its strength is that common structures in the success patterns of the original program are common patterns in the (syntactic) calls of its more specific version.

Since both specialisation and untupling preserve success patterns, the analysis of the resulting program is correct with respect to the original program for success patterns. There are a number of standard ways to enhance the technique to also provide analysis of call patterns. In particular, in a bottom-up approach by applying the technique in combination with Magic Sets (see for example [2]); or in a top-down approach by using the improved success patterns (with the transformed program) to derive improved call patterns (with

the original program).

Another interesting application of the combined approach is to improve the results of termination analysis. This application must be considered with care as specialisation can cause an infinite derivation in the original program to become a failing derivation. In [13], the authors specify an infinite derivation preserving specialisation which must be used for this application.

As an example of a program where untupling improves termination analysis consider the following.

Example 6 *The following program checks if two binary trees have the same frontier with an "interleaved" traversal of the two trees so that it fails as soon as the two frontiers are in conflict.*

$$\begin{array}{l}
\mathbf{sameleaves}(\mathbf{Tree}1, \mathbf{Tree}2) \leftarrow \\
\quad \mathbf{sameleaves_forest}([\mathbf{Tree}1], [\mathbf{Tree}2]).
\end{array}$$

$$\begin{array}{l}
\mathbf{sameleaves_forest}([\,], [\,]). \\
\mathbf{sameleaves_forest}(\mathbf{Trees}1, \mathbf{Trees}2) \leftarrow \\
\quad \mathbf{getleaf}(\mathbf{Trees}1, \mathbf{Leaf}, \mathbf{NewTrees}1), \\
\quad \mathbf{getleaf}(\mathbf{Trees}2, \mathbf{Leaf}, \mathbf{NewTrees}2), \\
\quad \mathbf{sameleaves_forest}(\mathbf{NewTrees}1, \mathbf{NewTrees}2).
\end{array}$$

$$\begin{array}{l}
\mathbf{getleaf}([\mathbf{leaf}(\mathbf{A})|\mathbf{Trees}], \mathbf{A}, \mathbf{Trees}). \\
\mathbf{getleaf}([\mathbf{tree}(\mathbf{A}, \mathbf{B})|\mathbf{Trees}], \mathbf{L}, \mathbf{NewStack}) \leftarrow \\
\quad \mathbf{getleaf}([\mathbf{A}, \mathbf{B}|\mathbf{Trees}], \mathbf{L}, \mathbf{NewStack}).
\end{array}$$

Consider an attempt to prove termination for this program using a term-size norm — for example using the analyser described in [3] (which is also available online at `http://www.cs.bgu.ac.il/~mcodish/TerminWeb`*). Termination is not detected because in the recursive call to* $\mathbf{getleaf}/3$ *the first argument does not decrease in size.*

The structure analysis described in this paper determines that all of the success patterns for $\mathbf{getleaf}/3$ *are instances of the atom* $\mathbf{getleaf}([\mathbf{A}|\mathbf{B}], \mathbf{C}, \mathbf{D})$.

Specialisation and untupling give:

$$\begin{array}{l}\mathbf{sameleaves'}(\mathbf{Tree}1, \mathbf{Tree}2) \leftarrow \\ \quad \mathbf{sameleaves_forest'}([\mathbf{Tree}1], [\mathbf{Tree}2]).\end{array}$$

$$\begin{array}{l}\mathbf{sameleaves_forest'}([\,], [\,]). \\ \mathbf{sameleaves_forest'}([\mathbf{X}0|\mathbf{X}1], [\mathbf{X}2|\mathbf{X}3]) \leftarrow \\ \quad \mathbf{getleaf'}(\mathbf{X}0, \mathbf{X}1, \mathbf{Leaf}, \mathbf{NewTrees}1), \\ \quad \mathbf{getleaf'}(\mathbf{X}2, \mathbf{X}3, \mathbf{Leaf}, \mathbf{NewTrees}2), \\ \quad \mathbf{sameleaves_forest'}(\mathbf{NewTrees}1, \mathbf{NewTrees}2).\end{array}$$

$$\begin{array}{l}\mathbf{getleaf'}(\mathbf{leaf}(\mathbf{A}), \mathbf{Trees}, \mathbf{A}, \mathbf{Trees}). \\ \mathbf{getleaf'}(\mathbf{tree}(\mathbf{A}, \mathbf{B}), \mathbf{Trees}, \mathbf{L}, \mathbf{NewStack}) \leftarrow \\ \quad \mathbf{getleaf'}(\mathbf{A}, [\mathbf{B}|\mathbf{Trees}], \mathbf{L}, \mathbf{NewStack}).\end{array}$$

Observe that the size of the first argument in the recursive call to $\mathbf{getleaf'}/4$ *is strictly smaller than the size of the corresponding first argument in the head. This makes the difference for termination analysis.*

7 Comparison with `Pat(`$\mathcal{R}$`)`

As far as we are aware, `Pat(`$\mathcal{R}$`)`, is the first generic approach for enhancing an analysis domain $\mathcal{R}$ with structural information. `Pat(`$\mathcal{R}$`)` was introduced in [7]. Further details on the implementation can be found in [11] (Appendix B) while performance results for `Pat(`$\mathcal{R}$`)` are given in [6]. Additional insight into the `Pat(`$\mathcal{R}$`)` domain and its analysis can be found in [1].

The essential idea behind `Pat(`$\mathcal{R}$`)` is to describe a tuple of terms (the arguments to a procedure) as a tuple of "patterns" augmented with additional information. A pattern is a description of the structures in a term. It is similar to a $\mathbf{depth}(\mathbf{k})$ description [17], however, there is no fixed $\mathbf{k}$. In principle, a pattern for a term can simply be viewed as a generalisation of the term and with that view a pattern describes its set of instances. A least upper bound operation on tuples is defined as their most specific generalisation. The additional information augmenting a `Pat(`$\mathcal{R}$`)` description consists of an enhanced $\mathcal{R}$ component which describes the sub-structures in the tuple of patterns (rather than describing the arguments of the tuple) and a *same-values* component which specifies when certain sub-structures are equal.

Like `Pat(`$\mathcal{R}$`)`, our approach is generic in the sense that it may be used with any analysis domain. However in contrast to `Pat(`$\mathcal{R}$`)`, as it is expressed in terms of a program transformation, our technique can be applied within (almost) any analysis framework and for (almost) any type of analysis which relies on the approximation of success and/or call patterns. There is also a disadvantage in

basing the analysis on a program transformation. It implies that the technique will not apply to analyses which assume more detailed notions of observables (such as how structures are shared on the heap in the implementation). This is true also of the `Pat(`$\mathcal{R}$`)` analysis described by Bagnara [1].

There are many similarities between the two approaches. Both enhance the analysis of information expressed in $\mathcal{R}$ through pattern information without burdening the designer of the analysis with the need to implement new operations. Both approaches provide $\mathcal{R}$ information about sub-structures in a tuple of terms as well as the usual $\mathcal{R}$ information about the terms in the tuple.

In general, `Pat(`$\mathcal{R}$`)` is more powerful than untupling since in most cases the structural information required for clause pruning is not found in the untupling approach. The following example illustrates how pruning can lead to more precise $\mathcal{R}$ information.

Example 7 *Consider the goal `r(X)` and the program:*

$$\begin{array}{ll} \mathbf{r}(\mathbf{X}) \leftarrow \mathbf{p}(\mathbf{b},\mathbf{X}). & \mathbf{p}(\mathbf{a},\mathbf{a}). \\ \mathbf{r}(\mathbf{X}) \leftarrow \mathbf{p}(\mathbf{a},\mathbf{X}). & \mathbf{p}(\mathbf{b},\mathbf{b}). \\ & \mathbf{p}(\mathbf{c},_). \end{array}$$

Analysis with `Pat(`**Pos**`)` *will use pattern information about the first argument of `p/2` to ignore the contribution of the non-matching clauses in the analysis. Thus, it will determine that* **X** *is ground at the end of both clauses for `r/1` and hence that the goal `r(X)` grounds `X`. Analysis with our untupling approach cannot determine this: The specialisation and untupling transformations leave the program unchanged and subsequent analysis with* **Pos** *will not determine that* **X** *is ground because of the contribution of the program's last clause.*

However, the extra precision of `Pat(`$\mathcal{R}$`)` comes at a potentially high price: the new operations manipulate $\mathcal{R}$ descriptions for every sub-structure in the patterns of the current description. This can be prohibitive, especially for a domain such as `Pat(`**Pos**`)` where the worst-case complexity of the **Pos** operations is exponential in the number of variables. In many cases, the extra $\mathcal{R}$ precision does not actually lead to better analysis information since it is lost during subsequent analysis. When (two) successive `Pat(`$\mathcal{R}$`)` approximations for a given program point contain different patterns in matching positions, these are generalised by the least upper bound operation and $\mathcal{R}$ information is maintained only for the sub-structures which are consistent in the (two) approximations. Thus, by the end of the analysis, $\mathcal{R}$ information is provided only for the sub-structures in patterns which are invariant throughout the computation.

The main advantage of the untupling approach is that by performing a simple pattern analysis before applying the $\mathcal{R}$ analysis, it is possible to identify these

invariant patterns in the descriptions of a predicate's calls and returns. The subsequent pattern-enhanced $\mathcal{R}$ analysis will focus only on the sub-structures in *these* patterns.

Thus we claim that the untupling approach is inherently more efficient than `Pat(`$\mathcal{R}$`)`. The same work performed in the initial pattern analysis of the untupling approach is performed also in the pattern component of the `Pat(`$\mathcal{R}$`)` analysis. The program transformation applied in the untupling approach (specialisation and untupling) is linear in the size of the program and in practice fast. Finally, the work involved in the $\mathcal{R}$ analysis of the untupled program focuses only on the invariant sub-structures identified also in the `Pat(`$\mathcal{R}$`)` analysis.

To provide further insight on the tradeoff between precision and efficiency between the two approaches, we take a closer look at analysis results using the two. Unfortunately, we did not have access to a `Pat(`$\mathcal{R}$`)` analyser. Our comparison is based on the results described in [6] and on several `Pat(`$\mathcal{R}$`)` analyses which we have performed manually based on the formal specifications given in [6] and [7].

The experiments described in [6] indicate that a `Pat(`**Pos**`)` analysis is between 5 and 109 times slower than a corresponding **Pos** analysis for a collection of 11 benchmarks. An improvement in precision is obtained for only 2 of the these 11 benchmarks: `press.pl` and `read.pl`. The `Pat(`**Pos**`)` analyses for these programs are respectively 5.75 and 109.68 times slower than the corresponding **Pos** analyses.

Working through the `Pat(`**Pos**`)` analyses for these two programs it becomes apparent that the only predicate for which the enhanced **Pos** information in `Pat(`**Pos**`)` makes a difference in precision in the final result of the analysis is `factorize/3` in `press.pl`. This is also the only predicate among the 200 defined in these two programs which involves the use of difference lists. All of the other enhanced **Pos** information for sub-structures is either lost because sub-structures at corresponding positions in the `Pat(`**Pos**`)` descriptions are not invariant or else because the corresponding sub-structures are always passed around together. With the exception of the `factorize/3` predicate all of the gains in precision of **Pos** information in the `Pat(`**Pos**`)` analysis for these two programs are due to the pruning of clauses based on the pattern information.

We have also tested the untupling approach on the benchmarks described in [6]. For `press.pl` there is an improvement over **Pos** for the predicate `factorize/3` but the more precise information derived for this predicate does not propagate through the analysis to other predicates due to the lack of pattern information which enables `Pat(`**Pos**`)` to apply clause pruning. For all of the programs, the cost of the **Pos** after untupling increases by at most 35%.

In particular, with untupling, in many of the cases where no additional **Pos** information is derived by a `Pat(Pos)` there is no extra cost, aside from that related to the derivation of pattern information itself. This is in contrast to the experience for `Pat(Pos)` where analysis times are considerably slower even when there is no improvement in **Pos** precision.

In [7] the authors stress the major strength of `Pat(`$\mathcal{R}$`)` as a technique to improve the analysis of programs which manipulate difference lists. This is true also of our approach. In our second experiment we apply **Pos** analysis with and without untupling to analyse six programs involving difference lists. For all of these **Pos** with untupling was more precise than **Pos** and gave the most accurate **Pos** information possible. Thus, in this case untupling with **Pos** is as precise as `Pat(Pos)`.

8 Conclusion

We have illustrated the application of untupling and more specific logic programs to improve program analyses. The effect is similar to that obtained using the abstract domain construction `pat(`$\mathcal{R}$`)` proposed in [7]. The idea to apply specialisation and untupling to program analysis is mentioned in [14,13]. However the main focus in that work is on the application of the technique as a program optimisation. The technique has not previously been applied to the domain of program analysis and, to the best of our knowledge, its relationship to `Pat(`$\mathcal{R}$`)` has not been previously observed.

Our approach is very simple and easy to implement. Our investigation has also provided insight into the two different ways that pattern information can improve the results of an analysis over a domain $\mathcal{R}$. Namely, through pruning and through sub-structure information. This also clarifies the contribution of the `Pat(`$\mathcal{R}$`)` domain.

Our approach is more efficient than a `Pat(`$\mathcal{R}$`)` analysis but may be less precise. It may also be more difficult to precisely handle non-logical built-ins such as *var*. Our conclusion to this end is that pattern information is crucial to obtain the effects of clause pruning but enhancing $\mathcal{R}$ information to focus on sub-structure should be restricted to positions which maintain invariant patterns throughout the computation. A combination of the ideas described here with a straightforward cross product approach to augmenting an analysis with structural information will, we believe, provide most of the benefits of `Pat(`$\mathcal{R}$`)` but should be less expensive. Another approach would be to perform polyvariant specialisation of the program as suggested in [17]. The potential disadvantage of this approach is an explosion in code size.

A simple Prolog implementation of the untupling transformation (with its associated pattern analysis) can be obtained from

`ftp://ftp.cs.bgu.ac.il/pub/people/mcodish/untupler.pl`.

The pattern analysis is expressed as a simple bottom-up interpreter. It handles a representative set of Prolog built-ins. However, it is only a toy. Because of the way built-ins are handled the implementation is applicable only to improve program analyses based on properties which are closed under instantiation. As such, the `Pat(`$\mathcal{R}$`)` analysis currently maintained in the CHINA analyser is surely more robust and more powerful. Our conclusion to this end is therefore that the results of this paper indicate that it is worthwhile to consider the application of the proposed technique in real world analysers.

References

[1] R. Bagnara. *Data-Flow Analysis for Constraint Logic-Based Languages.* PhD thesis, University of Pisa, 1997.

[2] M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. *Journal of Theoretical Computer Science*, 124:93–125, 1994.

[3] M. Codish and C. Taboch. A semantic basis for termination analysis of logic programs. *Journal of Logic Programming*, 41:103–123, 1999.

[4] A. Cortesi, G. Filé, and W. Winsborough. *Prop* revisited: Propositional formula as abstract domain for groundness analysis. In *Proc. 6th Annual IEEE Symp. Logic in Computer Science*, pages 322–327, Los Alamitos, California, 1991. IEEE Computer Society Press.

[5] A. Cortesi, G. Filé, and W. Winsborough. Optimal groundness analysis using propositional logic. *Journal of Logic Programming*, 27(2):137–167, 1996.

[6] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Conceptual and software support for abstract domain design: Generic structural domain and open product. Technical Report CS-93-13, Brown University, Department of Computer Science, 1993. Available from ftp://ftp.cs.brown.edu/pub/techreports/93/cs93-13.ps.Z.

[7] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming. In ACM, editor, *Proceedings of 21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 227–239, New York, NY, USA, 1994. ACM Press.

[8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Jan. 1977.

[9] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Comput. Sci.*, 69(3):289–318, 1989.

[10] M. C. Fitting. A Kripke-Kleene semantics for logic programming. *Journal of Logic Programming*, 4:295–312, 1985.

[11] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(1):35–101, Jan. 1994.

[12] J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, $2^{\mathbf{nd}}$ edition, 1987.

[13] K. Marriott, L. Naish, and J. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1(2), 1990.

[14] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 909–923, Seatle, 1988. ALP, IEEE, The MIT Press.

[15] K. Marriott and H. Søndergaard. Bottom-up dataflow analysis of normal logic programs. *The Journal of Logic Programming*, 13(2 & 3):181–204, July 1992.

[16] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.

[17] T. Sato and H. Tamaki, Enumeration of success patterns in logic programs, *Theoretical Computer Science* **34** (1984) 227–240.

[18] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of *Prop*. *Journal of Logic Programming*, 23(3):237–278, June 1995.