HAL open science

Deriving Analysers by Folding/Unfolding of Natural Semantics and a Case Study: Slicing

Valérie Gouranton

▶ To cite this version:

Valérie Gouranton. Deriving Analysers by Folding/Unfolding of Natural Semantics and a Case Study: Slicing. [Research Report] RR-3413, INRIA. 1998. inria-00073277

HAL Id: inria-00073277
https://inria.hal.science/inria-00073277

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Deriving analysers by folding/unfolding of natural semantics and a case study: slicing

Valérie Gouranton

N° 3413

Avril 1998

THÈME 2

ISSN 0249-6399

Deriving analysers by folding/unfolding of natural semantics and a case study: slicing

Valérie Gouranton

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche no3413 — Avril 1998 — 20 pages

Abstract: We consider specifications of analysers expressed as compositions of two functions: a semantic function, which returns a natural semantics derivation tree, and a property defined by recurrence on derivation trees. A recursive definition of a dynamic analyser can be obtained by fold/unfold program transformation combined with deforestation. We apply our framework to the derivation of a slicing analysis for a logic programming language.

Key-words: systematic derivation, program transformation, natural semantics, proof tree, slicing analysis.

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Dérivation d'analyseurs à partir d'une sémantique naturelle par pliage/dépliage, application à l'analyse d'élagage

Résumé : Nous considérons la spécification d'un analyseur comme la composition de deux fonctions : une fonction sémantique, qui rend un arbre de dérivation de la sémantique naturelle, et une propriété définie par récurrence sur les arbres de dérivation. Une définition récursive d'un analyseur dynamique peut être obtenue par des transformations de programmes (pliage/dépliage), combinée avec des techniques de déforestation. Nous appliquons notre cadre générique à la dérivation d'une analyse d'élagage pour un langage de programmation logique.

Mots-clé : dérivation systématique, transformation de programmes, sémantique naturelle, arbre de preuve, analyse d'élagage.

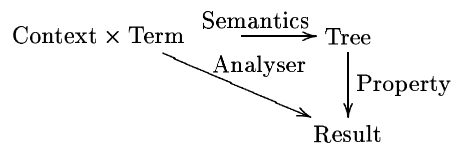
1 Introduction

A large amount of work has been devoted to program analysis during the last two decades, both on the practical side and on the theoretical issues. However, most of the program analysers that have been implemented or reported in the literature so far are concerned with one specific property, one specific language and one specific service (dynamic or static). A few generic tools have been proposed but they are generally restricted to one class of properties or languages, or limited in their level of abstraction. We believe that there is a strong need for environments supporting the design of program analysers and that more effort should be put on the *software engineering* of analysers.

We present a framework for designing analysers from operational specifications by program transformation (folding/unfolding). The analysis specification has two components: a semantics of the programming language and a definition of the property.

Natural semantics [7, 13] are a good starting point for the definition of analyses because they are both structural (compositional) and intensional. They are structural because the semantics of a phrase in the programming language is derived from the semantics of subphrases; they are intensional because the derivation tree that is associated with a phrase in the programming language contains the intermediate results (the semantics of subphrases). These qualities are significant in the context of program analysis because compositionality leads to more tractable proof techniques and intensionality makes it easier to establish the connection between the result of the analysis and its intended use.

Our semantics is defined formally as a function taking a term and an evaluation context and returning a derivation tree. The property itself is a function from derivation trees to a suitable abstract domain. The composition of these two functions defines a dynamic *a posteriori* analysis. It represents a function which initially calculates the trace of a complete execution (a derivation tree) of a program before extracting the required property. Program transformations *via* extended folding/unfolding techniques and simplification rules allow to obtain a recursive definition of the dynamic analyser (which does not call the property function). This function is in fact a dynamic *on the fly* analyser in the sense that it calculates the required property progressively during program execution. The following diagram shows the general organisation:



The key points of the approach proposed here are the following:

- The derivation is achieved in a systematic way by using functional transformations: unfolding and folding.
- It is applicable to a wide variety of languages and properties because it is based on natural semantics definitions.

As mentioned before, some of the analyses that we want to specify are dynamic and others are static. There is no real reason why these two categories of analyses should be seen as belonging to different worlds. In the paper we focus on dynamic analysis, considering that static analysis can be obtained in a second stage as an abstract interpretation of the dynamic analysis as presented in [10]. We outline this derivation in the conclusion. Note that our approach introduces a clear separation between the specification of an analysis (defined as a property on semantics derivation trees) and the algorithm that implements it.

We illustrate the framework by the formal derivation of a slicing analysis for a logic programming language. The different stages of the derivation are detailed in the following sections. Section 2

introduces contexts, terms, derivation trees and the semantics function. The abstract domain and the property function are presented in section 3. The transformation of the composition of the two specification functions (the semantics and the property) into a dynamic *on the fly* analyser is described in section 4. Related work, conclusion and avenues for further research are discussed in section 5 and section 6 respectively.

2 Natural semantics

The natural semantics of a language is a set of axioms and inference rules that define a relation between a context, a term in the programming language and a result. A natural semantics derivation tree has the form:

$$\text{Proof-Tree} = \quad [\text{RN}] \quad \frac{\text{Proof-Tree}_1 \quad ... \quad \text{Proof-Tree}_n}{\text{STT}}$$

where RN is the name of the rule used to derive STT. The conclusion STT is a statement, that is to say a triple consisting of a context, a term and a result.

In order to make formal manipulations easier, we express the construction of natural semantics derivation trees in a functional framework. Let `C` be the set of contexts, `T` the type of terms of the language and `PT` the type of derivation trees, then the semantic function S is a partial function of type:

$$\texttt{C} \times \texttt{T} \rightarrow \texttt{PT}$$

$$\begin{array}{lcl} \texttt{PT} & = & \texttt{STT} \times (\textit{list}\ \texttt{PT}) \times \texttt{RN} \\ \texttt{STT} & = & \texttt{C} \times \texttt{T} \times \texttt{NF} \\ \texttt{T} & = & \texttt{PP} \times \texttt{I} \end{array}$$

Derivation trees are made of a statement (the conclusion), a list of derivation trees (the premises) and the name of rule applied to derive the conclusion. We assume that a term is a pair of a program point and an expression. `STT` denotes the type of statements, `RN` rule names, `NF` normal forms (program results), `PP` program points and `I` expressions.

The important issue about the type of the semantic function is that it returns the whole natural semantics derivation tree, rather than just the result of the program. This choice makes it easier to define intensional analyses. The fact that we describe the semantics in a functional framework does not prevent us from dealing with non deterministic languages, as we show for a logic programming language. This is because we can use `NF` and `C` to represent sets of possible results and contexts.

We use the notation X.`ty` to denote the field of type `TY` of X. For example, we will make intensive use of the following expressions in the rest of the paper:

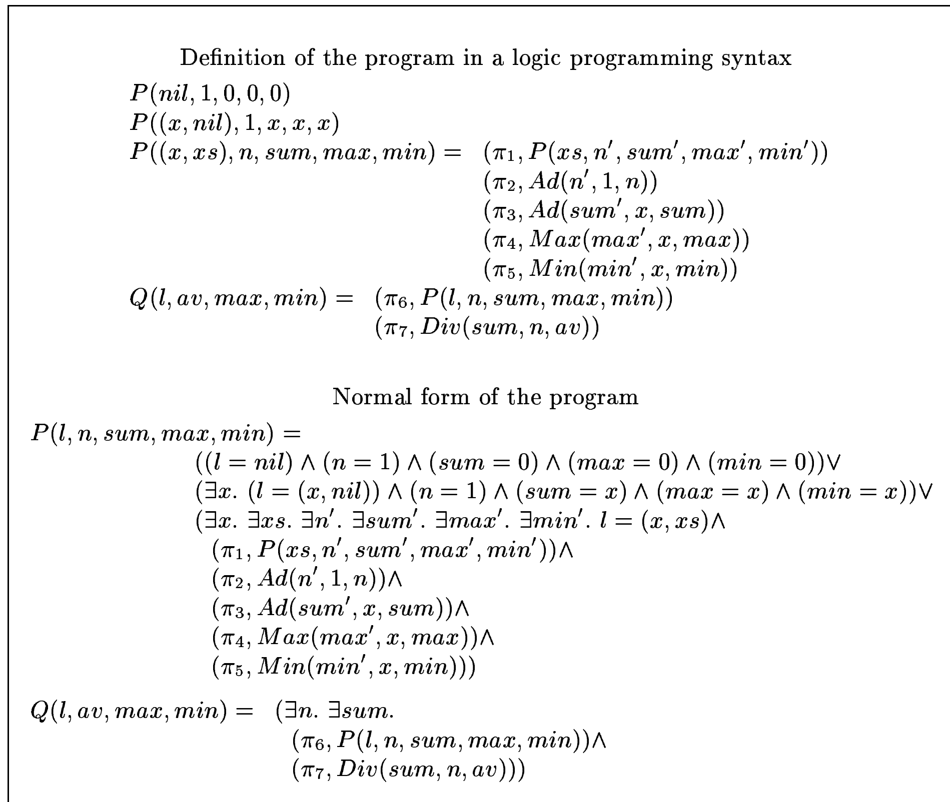
	type	meaning
PT.`stt`	`STT`	conclusion of PT
PT.`lpt`	(*list* `PT`)	premisses of PT
PT.`rn`	`RN`	name of the rule used at the root of PT
PT.`stt.c`	`C`	context of the conclusion sequent of PT
PT.`stt.t.i`	`I`	term of the conclusion sequent of PT
PT.`stt.t.pp`	`PP`	program point of the conclusion sequent of PT
PT.`stt.nf`	`NF`	normal form of the conclusion sequent of PT

The semantics of a logic programming language

We assume a program *Prog* which is a collection of predicate definitions of the form $[P_k(x_1, ..., x_n) = B_k]$. The body B_k of a predicate is in normal form and it contains only variables from $\{x_1, ..., x_n\}$. Normal forms are first order formulae (also called "goal formulae" in [15]) built up from predicate applications using only the connectives "and", "or", and "there exists". Their syntax is defined by:

$$I ::= \texttt{Op}(x_1, x_2, x_3) \mid x = t \mid U_1 \wedge U_2 \mid U_1 \vee U_2 \mid \exists x.U_1 \mid P_k(y_1, \ldots, y_n)$$

where Op stands for basic predicates[1] and P_k for user-defined predicates. U_i are terms of type T. We assume that each variable x occurring in a term $\exists x.U_1$ is unique. In a program, each subterm in this syntax is associated with a program point (using pairs).



Definition of the program in a logic programming syntax

$$
\begin{array}{ll}
P(nil, 1, 0, 0, 0) & \\
P((x, nil), 1, x, x, x) & \\
P((x, xs), n, sum, max, min) = & (\pi_1, P(xs, n', sum', max', min')) \\
 & (\pi_2, Ad(n', 1, n)) \\
 & (\pi_3, Ad(sum', x, sum)) \\
 & (\pi_4, Max(max', x, max)) \\
 & (\pi_5, Min(min', x, min)) \\
Q(l, av, max, min) = & (\pi_6, P(l, n, sum, max, min)) \\
 & (\pi_7, Div(sum, n, av))
\end{array}
$$

Normal form of the program

$$
\begin{array}{l}
P(l, n, sum, max, min) = \\
\quad ((l = nil) \wedge (n = 1) \wedge (sum = 0) \wedge (max = 0) \wedge (min = 0)) \vee \\
\quad (\exists x.\ (l = (x, nil)) \wedge (n = 1) \wedge (sum = x) \wedge (max = x) \wedge (min = x)) \vee \\
\quad (\exists x.\ \exists xs.\ \exists n'.\ \exists sum'.\ \exists max'.\ \exists min'.\ l = (x, xs) \wedge \\
\quad\ (\pi_1, P(xs, n', sum', max', min')) \wedge \\
\quad\ (\pi_2, Ad(n', 1, n)) \wedge \\
\quad\ (\pi_3, Ad(sum', x, sum)) \wedge \\
\quad\ (\pi_4, Max(max', x, max)) \wedge \\
\quad\ (\pi_5, Min(min', x, min)))
\end{array}
$$

$$
\begin{array}{ll}
Q(l, av, max, min) = & (\exists n.\ \exists sum. \\
 & \ (\pi_6, P(l, n, sum, max, min)) \wedge \\
 & \ (\pi_7, Div(sum, n, av)))
\end{array}
$$

Figure 1: A simple logic program

As an illustration of this syntax, Figure 1 presents a small program in a logic programming syntax and shows its translation into normal form. This program computes the minimum, the maximum, and the average of a list containing n integers. Some program points are omitted for the sake of readability.

Following [14], we assume an infinite set of program variables Pvar and an infinite set of renaming variables Rvar. Terms and substitutions are constructed using program variables and renaming variables. We distinguish two kinds of substitutions: program variable substitutions (Subst) whose domain and co-domain are subsets of Pvar and Rvar respectively, and renaming variable substitutions (Rsubst) whose domain and co-domain are subsets of Rvar:

$$
\begin{array}{lcl}
\texttt{Subst} & = & \texttt{Pvar} \rightarrow \texttt{Rterm} \\
\texttt{Rsubst} & = & \texttt{Rvar} \rightarrow \texttt{Rterm}
\end{array}
$$

where Rterm represents a term constructed with renaming variables Rvar. By convention, we use $\theta \in$ Subst for a program variable substitution and $\sigma \in$ Rsubst for a renaming variable substitution. The definition of substitution composition is modified to take account the role held by renaming variables. The modification occurs when $\theta \in$ Subst and $\sigma \in$ Rsubst, we have $\sigma \circ \theta \in$ Subst defined by:

$$
\begin{array}{l}
dom(\sigma \circ \theta) = dom(\theta) \\
(\sigma \circ \theta)(x) = \sigma(\theta(x)) \text{ for all } x \in dom(\theta)
\end{array}
$$

The domain of contexts for this language is defined by C = Tree(Subst) where Tree(H) is the type of binary trees with leaves of type H. We define contexts as binary trees of substitutions to take into account the non deterministic nature of the language. So, we gather in one derivation the computation of all the substitutions of a program. A particular control strategy for the implementation

[1]We consider only ternary basic predicates here, but other arities are treated in the same way.

[Op] $C \vdash (\pi, \texttt{Op}(x_1, x_2, x_3)) \rightarrow \overline{op}(C, x_1, x_2, x_3)$

[Eq] $C \vdash (\pi, x = t) \rightarrow \overline{unif}\,(C, x, t)$

[$\wedge$] $$\frac{C \vdash U_1 \rightarrow R_1 \quad R_1 \vdash U_2 \rightarrow R_2}{C \vdash (\pi, U_1 \wedge U_2) \rightarrow R_2}$$

[$\vee$] $$\frac{C \vdash U_1 \rightarrow R_1 \quad C \vdash U_2 \rightarrow R_2}{C \vdash (\pi, U_1 \vee U_2) \rightarrow union\,(C, R_1, R_2)}$$

[$\exists$] $$\frac{\overline{Add}\ (C, x, rx) \vdash U_1 \rightarrow R_1}{C \vdash (\pi, \exists x.U_1) \rightarrow \overline{Drop}\ (R_1, x)} \quad rx \in Rvar \quad \text{fresh variable}$$

[Call] $$\frac{\overline{Ren}_k(C) \vdash B_k \rightarrow R_1}{C \vdash (\pi, P_k(y_1, ..., y_n)) \rightarrow \overline{Ext}_k(C, R_1)} \quad \text{with } [P_k(x_1, ..., x_n) = B_k] \in Prog$$

$\overline{F}(N(T_1, T_2), x_1, \ldots, x_n) = N(\overline{F}(T_1, x_1, \ldots, x_n), \overline{F}(T_2, x_1, \ldots, x_n))$
$\overline{F}(\theta, x_1, \ldots, x_n) = F(\theta, x_1, \ldots, x_n)$

$op(\theta, x_1, x_2, x_3) = \texttt{let}\ \sigma = [(\theta(x_1)\ \text{op}\ \theta(x_2))/\theta(x_3)]\ \texttt{in}\ \sigma \circ \theta$
if $\theta(x_1)$ and $\theta(x_2)$ are ground and $\theta(x_3) \in \texttt{Rvar}$, $\perp$ otherwise

$unif(\theta, x, t) = \texttt{let}\ \sigma = \text{mgu}(\theta(x), \theta(t))\ \texttt{in}\ \sigma \circ \theta$ if $\theta(x)$ and $\theta(t)$ can be unified ,
$\perp$ otherwise

$union(N(T_1, T_2),\ N(U_1, U_2),\ N(V_1, V_2)) = N(union(T_1, U_1, V_1),\ union(T_2, U_2, V_2))$
$union(\theta, U, V) = N(U, V)$

$Add(\theta, pv, rv) = \theta[rv/pv]$ with $v \neq pv \Rightarrow \theta[rv/pv](v) = \theta(v)$ and $\theta[rv/pv](pv) = rv$

$Drop(\theta, pv) = \theta_{/pv}$ with $v \neq pv \Rightarrow \theta_{/pv}(v) = \theta(v)$ and $\theta_{/pv}(pv) = \perp$

$Ren_k(\theta) = [\theta(y_i)/x_i]$

$Ext_k(\theta, \theta') = \sigma \circ \theta$ with $\theta' = \sigma \circ [\theta(y_i)/x_i]$

Figure 2: Natural semantics of a logic programming language

$$
\begin{array}{l}
\mathcal{S}(C,T) = \texttt{case}\ \ T\ \texttt{of} \\
\quad (\pi, \texttt{Op}\,(x_1,x_2,x_3)) : ((C,T,\overline{op}(C,x_1,x_2,x_3)), \texttt{nil}, \texttt{Op}) \\
\quad (\pi, \texttt{Eq}(x,t)) : ((C,T,\overline{unif}\,(C,x,t)), \texttt{nil}, \texttt{Eq}) \\
\quad (\pi, \texttt{And}\ (U_1,U_2)) :\ \texttt{let}\ \ PT_1 = \mathcal{S}(C,U_1) \\
\qquad\qquad R_1 = PT_1.\texttt{stt.nf} \\
\qquad\qquad PT_2 = \mathcal{S}(R_1,U_2) \\
\qquad\qquad R_2 = PT_2.\texttt{stt.nf} \\
\qquad \texttt{in}\ \ ((C,T,R_2),[PT_1,PT_2],\wedge) \\
\quad (\pi, \texttt{Or}\ (U_1,U_2)) :\ \texttt{let}\ \ PT_1 = \mathcal{S}(C,U_1) \\
\qquad\qquad R_1 = PT_1.\texttt{stt.nf} \\
\qquad\qquad PT_2 = \mathcal{S}(C,U_2) \\
\qquad\qquad R_2 = PT_2.\texttt{stt.nf} \\
\qquad \texttt{in}\ \ ((C,T,union\,(C,R_1,R_2)),[PT_1,PT_2],\vee) \\
\quad (\pi, \texttt{Exists}\ (x,U_1)) :\ \texttt{let}\ \ PT_1 = \mathcal{S}(\overline{Add}\ (C,x,rx),U_1) \\
\qquad\qquad R_1 = PT_1.\texttt{stt.nf} \\
\qquad \texttt{in}\ \ ((C,T,\overline{Drop}\ (R_1,x)),[PT_1],\exists) \\
\quad (\pi, \texttt{Call}\,(P_k(y_1,\ldots,y_n))) :\ \texttt{let}\ \ PT_1 = \mathcal{S}(\overline{Ren_k}(C),B_k) \\
\qquad\qquad R_1 = PT_1.\texttt{stt.nf} \\
\qquad \texttt{in}\ \ ((C,T,\overline{Ext_k}(C,R_1)),[PT_1],\texttt{Call}) \\
\qquad\qquad \text{with}\ \ [P_k(x_1,\ldots,x_n) = B_k]\ \in \text{Prog}
\end{array}
$$

Figure 3: The semantics function of a logic programming language

of the language corresponds to a particular ordering of the leaves of substitutions trees. For instance, the list of results of the usual depth-first evaluation strategy of Prolog is precisely the leaves of the substitution tree produced by our semantics ordered from left to right. We write $N(T_1,T_2)$ for a tree with subtrees T_1 and T_2.

The natural semantics of a simple logic programming language using the usual inference rule presentation is presented in Figure 2. In the figure, $\overline{F}(T)$ denotes the application of a function F to all the substitutions of a tree T and its result is also a tree. The function *op* represents the interpretation of operator `Op`. The substitution $unif(\theta,x,t)$ of `Subst` is defined for the unification of x and t *via* θ (rule `Eq`). The rule $\wedge$ is not surprising, the first formula U_1 of the conjonction is evaluated and the result R_1 is taken as context for the evaluation of the second formula U_2 of the conjonction; the result R_2 is the final result. For the rule $\vee$, the subtrees corresponding to the sub-formulae of the disjonction are evaluated independently. The function $union(T_1,T_2,T_3)$ is needed to build a new substitution tree joining the trees T_2 and T_3 produced by two subgoals. Its first argument is the initial substitution, which is used to identify the points where the joins have to be introduced (these points are the leaves of T_1). The rule $\exists$ uses two functions *Add* and *Drop*. *Add* is used to add a program variable in a substitution (the new program variable is attached to a free renaming variable) and *Drop* removes a variable from a substitution.

For the rule `Call`, two definitions of substitutions are needed. $Ren_k(C)$ creates a new substitution to execute the body of a clause (it amounts to a variable renaming) because the body B_k of a clause contains formal parameters x_i and C contains program variables y_i. $Ext_k(C,R_1)$ propagates the result of a predicate in the calling substitutions because C contains variables y_i and R_1 contains formal parameters x_i. From the definition of Ren_k, we see that the body B_k of a predicate is evaluated in an environment defining exactly the formal parameters of the predicate P_k.

The formal definitions of the functions introduced informally before are presented in the bottom of Figure 2.

The semantics in functional form is presented in Figure 3. The semantics function of Figure 3 takes two arguments (the context C and the term T) and it returns a derivation tree. The derivation tree contains the conclusion $(C, T, \mathcal{F}^k(C, \overline{R}, E))$, where $\mathcal{F}^k$ is the result of the program in functional form, the list of subtrees and the name k of the rule used to derive the conclusion. The body of the function is a list of cases selected by pattern matching on the form of the term. The function is defined by recurrence on the term. The set of definitions Prog is used as an implicit parameter of the semantics.

3 Specification of a slicing property

Slicing[2] a program consists in constructing a reduced version of the program (called a program *slice*) containing only those statements that affect a given set of variables at given program points (this set is called *the slicing criterion*). In program debugging, slicing makes it possible for a software engineer to focus on the relevant parts of the code. Slicing is also useful for testing, program understanding and in maintenance activities. Because of this diversity of applications, different variations on the notion of slicing have been proposed, as well as a number of methods to compute slices. First, a program slice can either be *executable* or not. Producing an executable slice makes it possible to apply further treatments to the result of the analysis. Another important distinction is between *static* and *dynamic* slicing. In the first case, the slice is computed without any assumption on the inputs, whereas the latter relies on some specific input data. Slicing algorithms can also be distinguished by their direction. *Backward* slicing identifies the statements of a program that may have some impact on the criterion whereas *forward* slicing returns the statements which may be influenced by the criterion. In this paper, we consider *dynamic backward slicing* with executable slices. Static slicing algorithms can be derived by abstract interpretation of dynamic slicing analysers ; this construction is sketched in the conclusion. We can describe forward slicing analysers in a similar way but slicing analyses producing non executable slices do not fit well into our framework since the specification of the analysis is a relation between the semantics of the original program and the semantics of the slice as presented in [10] .

Slicing was originally proposed by Weiser for imperative languages [28] and its application to logic programming [22] and functional programming [17] have been studied recently. In fact, the concept of slicing itself is very general: it is not tied to one specific style of programming[3] and it can lead to dynamic as well as static analysers [24].

A slicing analysis for a logic programming language (with programs in normal form) according to a program point and a set of variables of interest consists in keeping only the sub-goals of disjunctions of each clause (a clause defines a predicate) being able to affect the value of the variables of interest. If all sub-goals of a formula of the disjunction are dropped, then this formula is dropped. If all formulae of the disjunction of goals are dropped, then the clause is dropped. In the opposite case, the head of the clause defining the predicate is kept.

Let us take the program in normal form of Figure 1 to illustrate dynamic backward slicing. We assume that we are interested only in the value of the variable av at the program point π_7. The pair $\{(\pi_7, av)\}$ is called the slicing criterion. The dynamic slice of the program is extracted for one particular input. For instance, if we execute the predicate Q with nil as the initial value of l, we get:

$$\begin{array}{ll} P(nil, 1, 0, 0, 0) & \\ Q(l, av, max, min) = & (\pi_6, P(l, n, sum, max, min)) \\ & (\pi_7, Div(sum, n, av)) \end{array}$$

The predicate P is not recursively called and the first disjunctive part is satisfied, the third clause of P is never executed. The definition of the predicate Q is kept because all its clauses are useful to

[2] More precisely "backward slicing".

[3] Even if the details of the resulting analyses are of course.

compute the variable av. If we consider the execution of the program with $(2,(3,nil))$ as initial value of l, we recursively call the predicate P, we get:

$$
\begin{array}{lll}
P((x,nil),1,x,x,x) & & \\
P((x,xs),n,sum,max,min) = & (\pi_1, P(xs,n',sum',max',min')) \\
& (\pi_2, Ad(n',1,n)) \\
& (\pi_3, Ad(sum',x,sum))
\end{array}
$$

$$
\begin{array}{ll}
Q(l,av,max,min) = & (\pi_6, P(l,n,sum,max,min)) \\
& (\pi_7, Div(sum,n,av)
\end{array}
$$

Only a part of the third clause of the predicate P is kept, the program points π_4 and π_5 are dropped because they are not useful in computing av.

Assuming a set of pairs (π_i, v_i), where π_i is a program point and v_i a variable, a backward slicing analysis produces the slice computing for each point π_i the same values as the initial program for the variable v_i.

In our framework, a property is expressed by a function which takes at least an argument being the co-domain of the semantics function (a derivation tree of type `PT`) and the result of the property is an abstract domain. The slicing property takes an additional argument to represent the slicing criterion (of type $\mathcal{P}(\texttt{PP} \rightarrow \texttt{Var})$) and the type of the result is $\mathcal{P}(\texttt{PP})$ because slices are represented by sets of program points. The slicing criterion is represented in our approach by the mapping from program points to relevant variables. Because of the slicing property, we need extra information. We introduce a set of variables of interest according to a program point (this set represents the value of variable that must be preserved for computing the corresponding term). The initial value of the set is $\emptyset$. The property propagates this information of type $\mathcal{P}(\texttt{Var})$ and finally the type of the property is:

$$\alpha_{sl} : \texttt{PT} \times \mathcal{P}(\texttt{PP} \rightarrow \texttt{Var}) \times \mathcal{P}(\texttt{Var}) \rightarrow \mathcal{P}(\texttt{PP}) \times \mathcal{P}(\texttt{Var})$$

The slicing property α_{sl} for the logic programming language is presented in Figure 4. The property takes as arguments a derivation tree PT, plus two additional parameters $RV \in \texttt{PP} \rightarrow \mathcal{P}(\texttt{Pvar})$ and $D \in \mathcal{P}(\texttt{Pvar})$. The second argument RV (for Relevant Variables) is the slicing criterion mentioned above. A program point π associated with a non-empty set $RV(\pi)$ is called an observation point. The third argument D of the property represents the set of variables whose values must be preserved in the output context[4] (normal form) of the term, *i.e.* the set of variables that must be preserved in the result R_i of the evaluation of the derivation tree PT_i. In the initial call, D is the empty set. The function α_{sl} is called recursively on the intermediate derivation trees (PT_i) of the natural semantics and sets of observation variables.

The result of the property is a pair (S,N) with $S \in \mathcal{P}(\texttt{PP})$ and $N \in \mathcal{P}(\texttt{Pvar})$. S is the set of program points of the term T that must be kept in the slice and N is the set of variables whose value must be preserved in the input context[5]. A program point must be kept in the slice if it can influence an observation point or the value of a variable of D in the output context. The same condition applies to decide which variables must be preserved in the input context. If the program point can be removed from the slice, the result of the property is $(\emptyset, D)$, which means that no program point is added to the slice and the variables whose values must be preserved in the input context are the variables that are necessary in the output context. Otherwise, the first component of the result of the property is $\bigcup_i S_i \cup \{\pi\}$ because π has to be added to the program points collected in the subterms of T. The second component N of the result is the set of variables whose value must be preserved in the input

[4]For a forward property this argument would characterise the input context rather than the output context.

[5]For a forward property this argument would characterise the output context rather than the input context.

context C. It contains at least the set D and the variables $RV(\pi)$ of slicing criterion, thus we factorise that by setting $D' = D \cup RV(\pi)$ in beginning of the slicing definition.

We assume that the definitions of S_i and N_i are not mutually recursive. The definition of the sets of observation variables (third argument of α_{sl}) do not use $N_j,\ j > i$. Note that this is a characteristic feature of a backward analysis.

In Figure 4, the relation $Indep(C, D_1, D_2)$ is used to ensure that two sets of variables D_1 and D_2 are independent, which is the case when they do not share any renaming variables (in any substitution of the context C). The relation *Indep* appears in the first two cases as a necessary condition to exclude the term from the slice. If the relation holds, then the (renaming variable) substitution resulting from the evaluation of the term cannot have any impact on the variables of D. The relation $UF(C, x, t)$ is satisfied if the unification of x and t cannot fail for any substitution of C. It is a prerequisite for excluding `Eq`(x, t) from the slice because a failure is recorded in the substitution tree as the $\perp$ substitution[6]; as a consequence, it has an impact on all the variables. This condition was not included in the `Op` case, assuming that the logic programming language is equipped with mode annotations ensuring that operators are always called with their first two arguments ground and the last one free[7]. In both the `Op` and the `Eq` cases, the set of necessary variables (at the input of the program point) is D' added to all the program variables of the term: the set $\{x_1, x_2, x_3\}$ for `Op` (x_1, x_2, x_3) and the set of program variables occurring in t increased with x for the rule `Eq` (x, t). The formal definitions of *Indep* and *UF* are presented in the bottom of Figure 4.

For the rule `And`, both branches are processed in turn (the second branch first since our property is computed in a backward direction). The property is first called with PT_2 and D' and the result is (S_2, N_2); then the property is computed with PT_1 and N_2, we have (S_1, N_1) as the result. The program point π can be removed from the slice when both S_1 and S_2 are empty sets. When the program point is kept, the result of the operator `And` is then $(S_1 \cup S_2 \cup \{\pi\}, N_1)$ because the information about program points of both branches is kept and the set N_1 represents the variables must be preserved in the input context since we consider a backward direction.

The treatment of `Or` is different: the term is systematically kept in the slice because it always influences the values of all the variables (through the introduction of subtrees in the derivation tree). Both branches are computed independently and the result gathers the information of these two branches.

The rules for `Exists` and `Call` are not surprising. We assume that the variable x in `Exists`(x, U_1) is unique in a normalised program; so x can be removed from the set of necessary variables yielded by the analysis of U_1 (hence $N_1 - \{x\}$).

In the rule for `Call`, first the derivation tree corresponding to the predicate P_k is computed with the set $\{x_i \mid \neg Indep(C, D', \{y_i\})\}$ of variables to be preserved (*i.e.* the formal parameters x_i of P_k bounded to arguments y_i which are not independent from the set D'). The test in the rule for `Call` is similar to the test in the `Op` case. We could make more sophisticated choices to avoid including all the variables $y_1..., y_n$ in the set of the necessary variables.

[6]Note that $\perp$ is an absorbing element for the semantics of the language. For instance $op(\perp, x_1, x_2, x_3) = \perp$ and $unif(\perp, x, t) = \perp$.

[7]Otherwise an extra condition based on *UF* can be added as in the `Eq` case.

```
αsl (PT, RV, D) =
 let    π  =  PT.stt.t.pp
        C  =  PT.stt.c
        D' =  D ∪ RV(π)
 in     case (PT.lpt, PT.rn) of

(nil, Op) :  let Op (x1, x2, x3)  =  PT.stt.t.i
             in  if   RV(π) = ∅ and Indep(C, D, {x3})
                 then (∅, D)
                 else ({π}, D' ∪ {x1, x2, x3})
(nil, Eq) :  let   Eq (x, t)  =  PT.stt.t.i
             in     if   RV(π) = ∅  and UF(C, x, t) and Indep(C, D, Pv(t) ∪ {x})
                    then (∅, D)
                    else ({π}, D' ∪ Pv(t) ∪ {x})
([PT1, PT2], ∧) :  let    (S2, N2)  =  αsl (PT2, RV, D')
                          (S1, N1)  =  αsl (PT1, RV, N2)
                   in     if RV(π) = ∅ and S1 ∪ S2 = ∅
                          then (∅, D)
                          else ( S1 ∪ S2 ∪ {π}, N1)
([PT1, PT2], ∨) :  let    (S2, N2)  =  αsl (PT2, RV, D')
                          (S1, N1)  =  αsl (PT1, RV, D')
                   in    ( S1 ∪ S2 ∪ {π}, N1 ∪ N2)
       (PT1, ∃) :  let    Exists (x, U1)  =  PT.stt.t.i
                          (S1, N1)  =  αsl (PT1, RV, D')
                    in     if RV(π) = ∅ and S1 = ∅
                           then (∅, D)
                           else (S1 ∪ {π}, N1 − {x})
(PT1, Call) :  let    Call(Pk(y1, ..., yn))  =  PT.stt.t.i
                      (S1, N1)  =  αsl (PT1, RV, {xi | ¬Indep(C, D', {yi})})
               in     if   RV(π) = ∅ and S1 ∪ N1 = ∅ and Indep(C, D, {y1, ..., yn})
                      then (∅, D)
                      else (S1 ∪ {π}, D' ∪ {y1, ..., yn})
```

$UF(C, x, t) = \forall \theta \in C.\ \theta \neq \perp \Rightarrow \exists \sigma = mgu(\theta(x), \theta(t))$

$Pv(t) =$ set of program variables occurring in t

$Rv(rt) =$ set of renaming variables occurring in rt

$Indep(C, D_1, D_2) = \forall \theta \in C.\ \theta \neq \perp \Rightarrow \{Rv(\theta(x)) \mid x \in D_1\} \cap \{Rv(\theta(x)) \mid x \in D_2\} = \emptyset$

Figure 4: Slicing property

4 Derivation of the dynamic *on the fly* analyser

We have presented in section 2 the semantics function $\mathcal{S}$ and the property α_{sl} in functional form in section 3. The general organisation is described by the following diagram:

$$\begin{array}{ccc} \mathtt{C} \times \mathtt{T} & \xrightarrow{\mathcal{S}} & \mathtt{PT} \\ & \searrow^{\nu_a} & \downarrow \alpha_{sl} \\ & & \mathtt{D}_a \end{array}$$

The composition of the property α_{sl} and the semantics $\mathcal{S}$ is a function of type $\mathtt{C} \times \mathtt{T} \rightarrow \mathtt{D}_a$, where $\mathtt{D}_a$ is the domain of abstract values, the result of the analysis. This function computes successively the derivation tree related to a program, then the property of interest for this tree. It corresponds to a dynamic analysis *a posteriori* that inspects the trace produced after the program execution. It is interesting to formally describe dynamic analysers, because they are useful for instrumentation or debugging. We could also prefer dynamic analyses which, calculate their result *on the fly* i.e. during program execution. Their advantage is that they do not have to memorise traces before analysing them.

The derivation of a dynamic *on the fly* analyser from a dynamic analyser *a posteriori* presents similarities with a well-known program transformation within the framework of functional programming. The program transformation is called deforestation [27] and its purpose is to eliminate the intermediate data structures induced by the composition of recursive functions. Here, the intermediate structure is the derivation tree of the natural semantics. We use folding and unfolding transformations to carry out deforestation. The three principal operations are the following:

- unfoldings: we set $\nu_a(C,T) = \alpha_{sl}(\mathcal{S}\,(C,T))$ and replace in the expression the calls to the recursive functions α_{sl} and $\mathcal{S}$ by their definition.
- applications of laws on the operators of the language (like the conditional ones, the expressions `case` and `let`).
- foldings which consist in replacing the occurrences of $\alpha_{sl}(\mathcal{S}\,(C',T')))$ from calls to $\nu_a(C',T')$.

The goal of these transformations is to remove all the calls to the property extraction function α_{sl}, to obtain a closed definition of $\nu_a(C,T)$. The function obtained is then a dynamic *on the fly* analyser since it does not build the intermediate derivation trees any more.

The partial correction of the transformation by folding/unfolding is obvious. The total correction is not assured in general because some inopportune foldings can introduce cases of non-termination. The Improvement Theorem in [19] can be extented to a method (*the extended improved unfold-fold method*) presented in [20] which makes it possible to show the total correction of the method proposed in this paper.

Dynamic slicing analyser

The definition of the dynamic slicing analyser for the logic programming language is the following:

$$\mathcal{SL}_d\,(C,T,RV,D) = \alpha_{sl}\,(\mathcal{S}\,(C,T),RV,D)$$

First, we use an unfolding technique applied to the semantics and the property functions. The transformation rules used for the derivation of the dynamic *on the fly* analyser by unfolding are the following rules:

[**Tr₁**]	$\texttt{case}\ E_i\ \texttt{of}\ E_1 : R_1\ \vdots\ E_k : R_k$	$\rightarrow\ R_i$
[**Tr₂**]	$\texttt{case}\ E\ \texttt{of}\ E_1 : R\ \vdots\ E_k : R$	$\rightarrow\ R$
[**Tr₃**]	$f(\texttt{case}\ E\ \texttt{of}\ E_1 : R_1\ \vdots\ E_k : R_k)$	$\rightarrow\ \texttt{case}\ E\ \texttt{of}\ E_1 : f(R_1)\ \vdots\ E_k : f(R_k)$
[**Tr₄**]	$f(\texttt{case}\ E\ \texttt{of}\ E_1 : R_1\ \vdots\ E_k : R_k\ ,\ \texttt{case}\ E\ \texttt{of}\ E_1 : R'_1\ \vdots\ E_k : R'_k)$	$\rightarrow\ \texttt{case}\ E\ \texttt{of}\ E_1 : f(R_1, R'_1)\ \vdots\ E_k : f(R_k, R'_k)$
[**Tr₅**]	$\texttt{case}(\texttt{case}\ E\ \texttt{of}\ E_1 : R_1\ \vdots\ E_k : R_k)\ \texttt{of}\ C'_1 : R'_1\ \vdots\ C'_l : R'_l)$	$\rightarrow\ \texttt{case}\ E\ \texttt{of}\ E_1 : \texttt{case}\ R_1\ \texttt{of}\ C'_1 : R'_1\ \vdots\ C'_l : R'_l\ \ \vdots\ \ E_k : \texttt{case}\ R_k\ \texttt{of}\ C'_1 : R'_1\ \vdots\ C'_l : R'_l$
[**Tr₆**]	$\texttt{let}\ x\ =\ E_1\ \texttt{in}\ E_2$	$\rightarrow\ E_2[E_1/x]$

We detail the derivation for two rules: `Op` and `And` (the other cases are obtained in the same way). We unfold $\mathcal{S}(C,T)$ in the definition of α_{sl} and we note $\{\texttt{case}T\texttt{of}\ldots\}$ the definition of $\mathcal{S}(C,T)$. We have:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathcal{SL}_d(C,T,RV,D) =} \\
\texttt{let} & \pi\ =\ \{\texttt{case}T\texttt{of}\ldots\}.\texttt{stt.t.pp} & (1) \\
 & C\ =\ \{\texttt{case}T\texttt{of}\ldots\}.\texttt{stt.c} & (2) \\
 & D'\ =\ D \cup RV(\pi) & \\
\texttt{in} & \texttt{case}\ (\{\texttt{case}T\texttt{of}\ldots\}.\texttt{lpt}, \{\texttt{case}T\texttt{of}\ldots\}.\texttt{rn})\ \texttt{of} & (3) \\
 & (\texttt{nil}, \texttt{Op})\ : B_{Op} & \\
 & ([PT_1, PT_2], \wedge)\ : B_{\wedge} & \\
 & \ldots &
\end{array}
$$

where B_{Op} and $B_{\wedge}$ are respectively the rules corresponding to the operators `Op` and `And`.

Applying the rules:
(1) [**Tr₃**] with $f =$ `.stt.t.pp` and [**Tr₂**]
(2) [**Tr₃**] with $f =$ `.stt.c`
(3) [**Tr₃**] with $f =$ `.lpt` and $f =$ `.rn`
we get:

```
SL_d (C, T, RV, D) =
 let   π  = T.pp
       D' = D ∪ RV(π)
 in    case (case T of (π, Op (x_1, x_2, x_3)) :  nil
                       (π, And (U_1, U_2)) :    let  PT_1 = S(C, U_1)
                                                     R_1  = PT_1.stt.nf
                                                     PT_2 = S(C, U_2)
                                                in   [PT_1, PT_2],
                       ...
             case T of (π, Op (x_1, x_2, x_3)) :  Op
                       (π, And (U_1, U_2)) :    And
                       ...
             ) of  (4)
       (nil, Op)  : B∧
       ...
```

where $M.\mathtt{pp} = \{\pi \mid (\pi, x) \in M\}$ if $M \in \mathcal{P}(\mathtt{PP} \times \mathtt{Var})$.

Applying the rule [$\mathbf{Tr_4}$] with $f = id$ to (4), we have:

```
SL_d (C, T, RV, D) =
 let   π  = T.pp
       D' = D ∪ RV(π)
 in    case (case T of (π, Op (x_1, x_2, x_3)) :  (nil, Op)
                       (π, And (U_1, U_2)) :    let  PT_1 = S(C, U_1)
                                                     R_1  = PT_1.stt.nf
                                                     PT_2 = S(C, U_2)
                                                in   ([PT_1, PT_2], And)
                       ...
             ) of
       (nil, Op)  : B_Op
       ([PT_1, PT_2], ∧)  : B∧
       ...
```

Applying the rule [$\mathbf{Tr_5}$], we get:

```
SL_d (C, T, RV, D) =
 let   π  = T.pp
       D' = D ∪ RV(π)
 in    case T of (π, Op (x_1, x_2, x_3)) :  case (nil, Op) of (nil, Op)          : B_Op
                                                              ([PT_1, PT_2], ∧)  : B∧
                                                              ...
                 (π, And (U_1, U_2)) :    let  PT_1 = S(C, U_1)
                                               R_1  = PT_1.stt.nf
                                               PT_2 = S(C, U_2)
                                          in   case ([PT_1, PT_2], And) of (nil, Op)          : B_Op
                                                                           ([PT_1, PT_2], ∧)  : B∧
                                                                           ...
                 ...
```

Applying the rule [$\mathbf{Tr_1}$] for each case, and replacing B_{Op} by its definition, we get:

```
SL_d (C, T, RV, D) =
 let   π  = T.pp
       D' = D ∪ RV(π)
 in    case T of (π, Op (x_1, x_2, x_3)) :  let Op (x_1, x_2, x_3) = {caseTof ...}.stt.t.i   (5)
                                            in  if   RV(π) = ∅ and Indep(C, D, {x_3})
                                                then (∅, D)
                                                else ({π}, D' ∪ {x_1, x_2, x_3})
                 (π, And (U_1, U_2)) :    let  PT_1 = S(C, U_1)
                                               R_1  = PT_1.stt.nf
                                               PT_2 = S(C, U_2)
                                          in   B∧
                 ...
```

Applying the rule [**Tr3**] to (5), the rule [**Tr6**] twice with $x = \pi$ and $x = PT_2$, we get the result of the unfolding step presented in the Figure 5.

```
SL_d (C, T, RV, D) =
 let   D' = D ∪ RV(T.pp)
 in    case T of
     (π, Op (x1, x2, x3)) :  if   RV(π) = ∅ and Indep(C, D, {x3})
                             then (∅, D)
                             else ({π}, D' ∪ {x1, x2, x3})
     (π, And (U1, U2)) :    let   PT1 = S (C, U1)
                                  R1 = PT1.stt.nf
                                  (S2, N2) = α_sl (S (R1, U2), RV, D')
                                  (S1, N1) = α_sl (S (C, U1), RV, N2)
                            in    if RV(π) = ∅ and S1 ∪ S2 = ∅
                                  then (∅, D)
                                  else ( S1 ∪ S2 ∪ {π}, N1)
```

Figure 5: Unfoldings of semantics and property functions

```
SL_d (C, T, RV, D) =
 let   D' = D ∪ RV(T.pp)
 in    case T of
     (π, Op (x1, x2, x3)) :  if   RV(π) = ∅ and Indep(C, D, {x3})
                             then (∅, D)
                             else ({π}, D' ∪ {x1, x2, x3})
     (π, And (U1, U2)) :    let   PT1 = S (C, U1)
                                  R1 = PT1.stt.nf
                                  (S2, N2) = SL_d (R1, U2, RV, D')
                                  (S1, N1) = SL_d (C, U1, RV, N2)
                            in    if RV(π) = ∅ and S1 ∪ S2 = ∅
                                  then (∅, D)
                                  else ( S1 ∪ S2 ∪ {π}, N1)
```

Figure 6: Dynamic (*on the fly*) slicing analysis

To obtain a dynamic *on the fly* analyser, we must apply folding steps that allows us to remove the calls of the function α_{sl}. Figure 6 presents the result of these foldings. The fact that $\mathcal{SL}_d$ itself calls $\mathcal{S}$ shows that it is a dynamic analysis.

Example

The predicate Q of Figure 1 is analysed with the slicing criterion $\{(\pi_7, av)\}$, the arguments are the following:

$$T = Q(l, av, max, min)$$
$$RV(\pi) = (\text{if } \pi = \pi_7 \text{ then } \{av\} \text{ else } \emptyset)$$

and the results are:

$$\mathcal{SL}_d\,([nil/l, x/av, y/max, z/min], T, RV, \emptyset) = (\{\pi_6, \pi_7\}, \{l, av, max, min\})$$
$$\mathcal{SL}_d\,([(2,(3,nil))/l, x/av, y/max, z/min], T, RV, \emptyset) = (\{\pi_1, \pi_2, \pi_3, \pi_6, \pi_7\}, \{l, av, max, min\})$$

The first result of the analysis corresponds to the execution of the program with $Q(nil, x, y, z)$ as the initial call; the slice includes only the body of Q because the third clause of P is never executed[8].

[8]More precisely, in our semantics the third clause is always evaluated with $\perp$ as the input substitution. The definition of *Indep* shows that $Indep(\perp, D_1, D_2)$ is always *true*.

The second result corresponds to a call with $Q((2,(3,nil)),x,y,z)$ and the slice includes, as expected, all of the program except the program points π_4 and π_5.

5 Related work

The fold/unfold transformation framework used here is based on seminal work by Burstall and Darlington [2, 6]. The application of the technique to the derivation of programs has also been investigated in [5], which presents the synthesis of several sorting algorithms. The initial specification is expressed in terms of sets and predicate logic constructs. Our transformations are also reminiscent of the deforestation technique [3, 9, 27]: in both cases the goal is to transform a composition of recursive functions into a single recursive definition.

Generic frameworks for program analysis have been proposed in the context of logic programming languages [14] and data flow analysis [25, 29]. They rely on abstract interpretations of denotational semantics [14, 25] or interpreters [29] and genericity is achieved by parameterising the abstract domains and choosing appropriate abstract functions. The implementation details of the analysis algorithm can be factorised. While these tools may attain a higher degree of mechanisation than our framework, they do not offer to the user the same level of abstraction: they take as input the *specification of an abstract interpreter* rather than the *specification of a property*. Despite this difference of point of view, all these works are obviously inspired by the same goals. The framework introduced in [23] is closer to the spirit of the work presented in this paper but the technique itself is quite different. Programs are represented as models in a modal logic and a data flow analysis can be specified as a property in the logic. An efficient data flow analyser can be generated by partially evaluating a specific model checker with respect to the specifying modal formula. In comparison with this work, our framework trades mechanisation against generality: it is not limited to data flow analyses but the derivation process by fold/unfold transformations is not fully automatic.

Few papers have been devoted to the semantics of program slicing so far. A relationship between the behaviour of the original program and the behaviour of the slice is proved in [18]. The semantics of the language is expressed in terms of program dependence graphs; thus the programs are first analysed in order to extract their dependences. This approach is well suited to the treatment of imperative languages. Formal definitions and a classification of different notions of slicing are provided in [26]. The main distinctions are backward vs forward analysers, executable vs non executable slices, and dynamic vs static analysers. Their definitions are based on denotational semantics and they focus on the specifications of the analyses. In [8] a description of a family of slicing algorithms generalising the notions of dynamic and static slice to that of a constrained slice is presented. Genericity with respect to the programming language is achieved through a translation into an intermediate representation called PIM. Programs are represented as directed acyclic graphs whose semantics is defined in terms of rewriting rules. Slicing is carried out using term graph rewriting with a technique for tracing dynamic dependence relations. It should be noted that a richer notion of slicing has been proposed for logic programming languages, which returns not only the set of program points that must be kept in the slice, but also the necessary variables at each program point [22]. This increased precision can also be expressed in our framework, but we preferred to present the simpler version here for the sake of size and readability. By collecting the following information

$$(\bigcup_i S_i + \{(\pi, D \cup N)\}, N)$$

we can modify straightforwardly each rule in order to get the same precision as [22].

6 Conclusion

We have presented a method to derive dynamic analysers by program transformation (folding/unfolding). A dynamic analyser is expressed as composition of a semantics and a property functions. The analyser is called *a posteriori*, it is a function computing first a complete program execution trace (derivation tree) and then extracting the property of interest. An recursive definition of an analyser can be obtained by program transformation. This function is a dynamic *on the fly* analyser that computes the property during program execution.

We have focussed on dynamic analysis in the body of paper. Our generic dynamic analyser is defined in a strongly typed functional language[9]. As a consequence, we can rely on previous results on logical relations and abstract interpretation [1, 4] in order to systematically construct static analysers from the dynamic analysers. The first task is to provide abstract domains for the static slicing analyser and the corresponding abstraction functions. We recall that the type of the dynamic analyser is $\mathtt{C} \times \mathtt{T} \times (\mathtt{PP} \rightarrow \mathcal{P}(\mathtt{Pvar})) \times \mathcal{P}(\mathtt{Pvar}) \ \rightarrow \ \mathcal{P}(\mathtt{PP}) \times \mathcal{P}(\mathtt{Pvar})$. Since $\mathtt{PP} \rightarrow \mathcal{P}(\mathtt{Pvar})$, $\mathcal{P}(\mathtt{Pvar})$ and $\mathcal{P}(\mathtt{PP})$ are already abstract domains associated with the dynamic analysis, only $\mathtt{C}$ needs to be abstracted[10]. The next stage to derive a correct static analyser is to find appropriate abstractions for the constants and operators occurring in the definition of the analyser. It is shown in [1] that the correctness of the abstract interpretation of the constants and operators of the language entails the correctness of the abstract interpretation of the whole language. The correctness of the abstract interpretation means that the results of the dynamic analysis and the static analysis are related if their arguments are. In fact, it is possible to define the most precise abstraction for each constant and operator of the language [1]. The basic idea to find the best abstraction $op^a(v_1^a, \ldots, v_n^a)$ of an operator *op* is to define it as the least upper bound of the abstractions of all the results of *op* applied to arguments v_i belonging to the concretisation sets of the arguments of the v_i^a. The technique sketched here provides a systematic way to construct a correct abstract interpretation, and thus to derive a static analyser from a dynamic analyser [10, 11]. By deriving static analysers as abstractions of dynamic analysers, we can see the dynamic analyser either as an intermediate stage in the derivation of a static analyser (playing a role similar to a collecting semantics) or as the final product of the derivation.

The theory of abstract interpretation [4] provides a strong formal basis for static program analysis. The work described here does not provide an alternative formal underpinning for program analysis. Its goal is rather to put forward a derivation approach for the design of analysers from high level specifications.

Our framework is applicable to a wide variety of languages, properties and *type of service* (dynamic or static). We have proposed in the body of the paper a formal definition of a dynamic slicing analyser for a logic programming language. To our knowledge, this definition is the first one to be formal, so the benefit of our approach is striking in this case. In [10], we present the derivation of dynamic and static analysers for a strictness analysis of a higher-order functional language and a live variable analysis for an imperative language. We have also applied this work for a globalisation analysis of a higher-order functional language and a generic sharing analysis. Pushing our approach ever further we arrive at a natural semantics format and a format for slicing, as presented in [11]. We have shown the correctness of the slicing property format. These formats can be instantiated for several programming languages (imperative language, logic programming language and functional language). The slicing property for the logic programming that we have presented here is an instantiation of the slicing format.

As mentioned in the introduction, we wanted to establish the connection between the result of the analysis and its intended use. Analyses are generally performed to check assumptions about the be-

[9]Note that the typing mentioned here has nothing to do with the language in which the analysed programs are written, this language itself can perfectly well be untyped.

[10]Of course, as usual in abstract interpretation, $\mathtt{PP} \rightarrow \mathcal{P}(\mathtt{Pvar})$, $\mathcal{P}(\mathtt{Pvar})$ and $\mathcal{P}(\mathtt{PP})$ can also be abstracted if further approximations are needed, but we do not consider this issue here.

haviour of the program at specific points of its execution or to enable program optimisations. In both cases the intention of the analysis can be expressed in terms of a transformation and a relation as presented in [10, 11]. The transformation depends on the result of the analysis and the relation establishes a correspondence between the semantics of the original program and the transformed program. For example, in the case of a program analysis for compiler optimisation the transformation expresses the optimisation that is allowed by the information provided by the analysis and the relation is the equality between the final results (or outputs) of the original and the transformed program. It is not always the case that the relation is the equality: a counter-example is slicing analysis described in this paper (because the new program is required to behave like the original one only with respect to specific program points and variables). We have formally defined and proved in [11] a property for the intention of a slicing analysis but space considerations prevent us from presenting the intentional property for slicing.

There is a main aspect in which the work described here may seem limited: we have used only natural semantics and terminating programs. Structural Operational Semantics (SOS) are more precise than natural semantics and they are required for a proper treatment of non-determinism, non-termination and parallelism [16]. In fact, the natural semantics introduced in section 2 can be replaced by SOS without difficulty[11] and the dynamic analyses can be defined in the very same way. The extra difficulty introduced by SOS is the fact that they create new program fragments which makes it necessary to abstract over the syntax of the language to derive a static analyser. This problem is discussed in [21]. We can also adapt our natural semantics to SOS by using the technique presented in [12]. To achieve this goal, the classical inductive interpretation of natural semantics has to be extended with coinduction mechanisms and rules must be defined to express divergence.

References

[1] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1:5–40, 1990.

[2] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24:44–67, 1977.

[3] W.N. Chin. *Automatic methods for program transformation.* PhD thesis, Imperial College, 1990.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth annual ACM Symposium on Principles of Programming Languages, POPL'77*, pages 238–252, Los Angeles, California, January 1977.

[5] J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.

[6] J. Darlington and R. M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

[7] T. Despeyroux. Typol: a formalism to implement natural semantics. Technical Report 94, INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le chesnay Cedex FRANCE, 1988.

[8] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *22nd annual ACM Symposium on Principles of Programming Languages, POPL'95*, pages 379–392, San Francisco, California, January 1995.

[11] In order to deal with SOS, we basically need to change the type `NF` in `STT` and to introduce a global loop in the semantics since a SOS rule represents a single evaluation step.

[9] A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pages 223–232, Copenhagen, 1993.

[10] V. Gouranton. *Dérivation d'analyseurs dynamiques et statiques à partir de spécifications opérationnelles.* PhD thesis, Université de Rennes, France, 1997.

[11] V. Gouranton and D. Le Métayer. Dynamic slicing: a generic analysis based on a natural semantics format. Technical Report 3375, INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le chesnay Cedex FRANCE, March 1998.

[12] H. Ibraheem and D. A. Schmidt. Adapting big-step semantics to small-step style: coinductive interpretations and "higher-order" derivations. In *Second Workshop on Higher-Order Techniques in Operational Semantics (HOOTS2)*, December 1997.

[13] G. Kahn. Natural semantics. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science*, number 247 in Lecture Notes in Computer Science, pages 22–39, Passau, Germany, 1987. Springer-Verlag.

[14] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A generic interpretation algorithm and its complexity analysis. In koichi Furukawa, editor, *Proceedings of the 8th International Conference on Logic Programming*, pages 64–78, Paris, France, 1991.

[15] D. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *Third International Logic Programming Conference*, volume 225 of *Lecture Notes in Computer Science*, pages 448–462, Imperial College, London, United Kingdom, July 1986.

[16] H. Riis Nielson and F. Nielson. *Semantics With Applications.* John Wiley & Sons, 1992.

[17] T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *International Seminar on Partial Evaluation*, Dagstuhl Castle, Germany, February 1996.

[18] T. Reps and W. Yang. The semantics of program slicing. Technical Report 777, University of Wisconsin, Madison, 1988.

[19] D. Sands. Proving the correctness of recursion-based automatic program transformations. In *Theory and Practice of Software Development, TAPSOFT'95.* Springer-Verlag, 1995.

[20] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18:175–234, March 1996.

[21] D.A. Schmidt. Abstract interpretation of small-step semantics. In *5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, Stockhohn, Sweden, June 1996. Springer Lecture Notes in Computer Science.

[22] S. Schoenig and M. Ducassé. A backward slicing algorithm for prolog. In *Third International Static Analysis Symposium, SAS'96*, number 1145 in Lecture Notes in Computer Science, pages 317–331, Aachen, Germany, September 1996. Springer-Verlag.

[23] B. Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Science*, 21(2):115–139, October 1993.

[24] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[25] G.A. Venkatesh. A framework for construction and evaluation of high-level specifications for program analysis techniques. In *SIGPLAN Conference on Programming Language Design and Implementation, PLDI'89*, volume 24, pages 1–12, Portland, Oregon, July 1989.

[26] G.A. Venkatesh. The semantics approach to program slicing. In *the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, PLDI'91*, pages 107–119, Toronto, Ontario, Canada, June 1991.

[27] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[28] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 4:352–357, July 1984.

[29] K. Yi and W. L. Harrison III. Automatic generation and management of interprocedural program analyses. In *Twentieth Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'93*, pages 246–259, Charleston, South Carolina, January 1993.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399