Dependency Analysis of Functional Specifications with Algebraic Data Structures

Oana F. Andreescu[1,2(✉)], Thomas Jensen[1,2], and Stéphane Lescuyer[1]

[1] Prove & Run, 75017 Paris, France
{oana.andreescu,thomas.jensen,stephane.lescuyer}@provenrun.com
[2] INRIA Rennes - Bretagne Atlantique, Rennes, France

Abstract. In the context of interactive formal verification of complex systems, much effort is spent on proving the preservation of the systems invariants. However, most operations have a localized effect on the system, which only really impacts few invariants at the same time. Identifying those invariants that are unaffected by an operation can substantially ease the proof burden for the programmer. We present a dependency analysis for a strongly-typed, functional language, which computes a conservative approximation of the input fragments on which the operations depend. It is a flow-sensitive interprocedural analysis that handles arrays, structures and variant data types. For the latter, it simultaneously computes a subset of possible constructors. We have validated the scalability of the analysis to complex transition systems by applying it to a functional specification of the MINIX operating system.

1 Introduction

Algebraic data types (structures and variants) and associative arrays are fundamental building blocks when representing, grouping and handling complex data efficiently. However, operations manipulating them are rarely concerned with the entire compound input data structure. Most frequently, they depend only on a limited subset of their input. A complete specification of such an operation will not only stipulate that the output possesses a certain property but will also include its *framing requirements*, i.e. the part of the input that it operates on. Specifying and proving the preservation of logical properties for the unmodified part is a particular manifestation of the more general *frame problem* [8] – a notoriously cumbersome task in formal software verification, imposing unnecessary manual effort [9].

The verification of a given property can be simplified if we can determine the input fragments on which the property depends. This is the purpose of the dependency analysis presented in this paper. Our analysis targets a functional language that handles immutable algebraic data types and arrays. Furthermore, it is designed to be used on programs as well as on specifications. In contrast to the vast majority of static analyses that are mostly used only on actual code and in an essentially purely automatic setting, our analysis is thought of as a companion tool to be exploited in the middle of *interactive* program verification.

M. Butler et al. (Eds.): ICFEM 2015, LNCS 9407, pp. 116–133, 2015.
DOI: 10.1007/978-3-319-25423-4_8

1.1 A Motivating Example

The work reported in this paper is motivated by the formal verification of operating systems. To illustrate the problem that we are addressing, consider an abstract process manager and the data structures for its fundamental components: `process` and `thread`, shown in Fig. 1a. A *process* is an executing instance of an application that can consist of multiple *threads* that share the same address space. A *thread* is a path of execution within a process and it is modeled as a structure having fields such as the thread's identifier and the memory region for its stack. The current state of a thread is defined as a variant having three alternatives: `READY`, `BLOCKED`, `RUNNING`. Similarly, a process is a structure including an identifier for the currently running thread and an array of possibly inactive threads associated with it. Whether a thread in the `thread` array is active or has terminated is indicated by a variant of type `option_thread = | Some(thread th) | None`.

```
type process = {
  array<option_thread> threads;
  int pid;
  int current_thread;
  address_space address_space;
}
```

```
type thread = {
  int identifier;
  state current_state;
  mem_region stack;
}
```

a) Data Structures

```
predicate run_thread(process pr, int new_id)
-> [true: process new_pr | invalid_id]
predicate disjoint_stacks(process pr) -> [true | false]
```

b) Signatures for the Example Functions

Fig. 1. Example - data structures and functions of an abstract process manager

The signature of a Boolean function `disjoint_stacks`, written in a modeling language that we will present in Sect. 2, is shown in Fig. 1b. It verifies a fundamental property of a valid process state, namely that the stack regions of all active threads associated with the input `pr` are disjoint. Its result depends only on the array `threads` of the input `pr` and for each *active thread element* only on the field `stack`. All other input sub-elements are irrelevant to the result.

Another function `run_thread` has two possible execution scenarios: `true` and `invalid_id`. It stops the currently running thread and starts the one having the identifier given as an input. If it is valid, then a new process `new_pr` is returned for which `current_thread` is set to `new_id`. In the array `threads`, the state of the thread identified by `new_id` is set to `RUNNING`. The function's precondition stipulates that the `disjoint_stack` property holds for the input `pr` and that the input thread is `READY`. Proving the property's preservation is intuitively easy once the function's effects and the input subset on which `disjoint_stack` depends

are known. Automatically proving the preservation of invariants concerning only fields or elements that have not been altered by a transition in the system would considerably diminish the number of proof obligations.

This is precisely the issue that we are addressing: the delimitation of the input subset on which the output depends, given an operation with a compound input. We define *dependency* as the observed part of a structured domain and strive to obtain type-sensitive results, distinguishing between the sub-elements of arrays and algebraic data types and capturing the dependency specific to each. The targeted results mirror – in terms of dependency – the layered structure of compound data types.

Generally, our dependency analysis targets complex transition systems. These are characterized by states defined by compound data structures and transitions, i.e. state changes, that map an input state to an output state. In particular, we are applying it to an abstract model of an operating system, stemming from *ProvenCore* [6], an ongoing project revolving around a fully secure micro-kernel.

ProvenCore, inspired by MINIX 3.1, is a general-purpose micro-kernel that ensures *isolation*. Its proof is based on multiple refinements between successive models, from the most abstract, on which the *isolation* property is defined and proved, to the most concrete, i.e. the actual model used for code generation.

The *global states* of the abstract layers are complex structures with multiple compound fields. Commands such as *fork*, *exec*, *exit* can be executed. Each of these receives as an input the global state before executing the command and returns the state of the system after execution. Most supported commands affect only a handful of invariants, leading to a much more complex, but fundamentally similar version of the situation depicted by our introductory example.

Outline. The rest of this paper is structured as follows: in Sect. 2 we underline the specificities of our modeling language. The defined abstract domain of dependencies is described in Sect. 3. It is followed in Sect. 4 by an in-depth presentation of our analysis at an intraprocedural level and in Sect. 5 by a summary of it at an interprocedural level. In Sect. 6 we discuss the results obtained on two abstract layers of ProvenCore. Finally, in Sect. 7 we review related work.

2 The Modeling Language

In this section we present the unified programming and specification language that we will be analyzing. It is an idealized version of a language developed at *Prove & Run*[1] and designed to facilitate proofs and to allow users to write both the implementation and the specification of programs. It is purely functional, side-effect free and strongly-typed. The basic building blocks of programs written in our language are *predicates*, the equivalent of functions in common programming languages. In addition to the common built-in primitive types traditionally available, structures and variants are also provided. The language is designed to write code that will subsequently be proven, so it allows the definition

[1] http://www.provenrun.com/.

of various types of logical specifications, ranging from pre- and postconditions, local assertions and loop invariants to inductive predicates.

2.1 Types and Statements

For defining the language we are working on, we let $\mathbb{T}$ be the universe of type identifiers and $T_0 \subset \mathbb{T}$ the set of base types identifiers. Furthermore, let $\mathscr{F}$ be the set of structure field identifiers and $\mathscr{C}$ the set of variant constructors.

$$
\begin{array}{rll}
t := & \mid \tau \in T_0 & \text{base types} \\
& \mid \boldsymbol{structure}\{f_1 : t, \ldots, f_n : t\} \quad f_i \in \mathscr{F}, 1 \leq i \leq n & \text{structures} \\
& \mid \boldsymbol{variant}[C_1(t\ e_1) \mid \ldots \mid C_m(t\ e_m)] \; C_i \in \mathscr{C}, 1 \leq i \leq m & \text{variants} \\
& \mid \boldsymbol{array}^t\langle t \rangle & \text{arrays}
\end{array}
$$

A *structure* is a data type grouping elements of different types called *fields* and represents the *Cartesian product* of its fields' types. A *variant* is the *disjoint union* of different types. It represents data that may take on multiple forms, where each form is marked with a specific tag called the *constructor*. *Arrays* group a collection of data of the same type (given in angle brackets) into a single entity; each element is selected by an index whose type is included (as denoted by the superscript) in the array's definition.

A program in our language is a collection of predicates. A predicate has input and output parameters and a body of statements of the form shown in Table 1.

Table 1. Supported statements

statement :=	$\mid p(e_1, \ldots, e_n)\ [\lambda_1 : \bar{o}_1 \mid \ldots \mid \lambda_m : \bar{o}_m]$	(1)	predicate call
	$\mid e_1 = e_2$	(2)	equality test
	$\mid o := e$	(3)	assignment
	$\mid s := \{e_1, \ldots, e_n\}$	(4)	create structure s
	$\mid \{o_1, \ldots, o_n\} := s$	(5)	structure destructuring
	$\mid o := s.f_i$	(6)	access field f_i
	$\mid s' := \{s\ \boldsymbol{with}\ f_i = e_i\}$	(7)	update field f_i
	$\mid s' = \langle f_1, \ldots, f_k \rangle s''$	(8)	test equality on fields $f_1, ..., f_k$
	$\mid v := C_p[e_p]$	(9)	create v with constructor C_p
	$\mid \boldsymbol{switch}(v)\ \boldsymbol{as}\ [o_1 \mid \ldots \mid o_n]$	(10)	variant destructuring
	$\mid v \in \{C_1, \ldots, C_k\}$	(11)	variant possible
	$\mid o := a[i]$	(12)	array access at index i
	$\mid a' := [a\ \boldsymbol{with}\ i = e]$	(13)	array update at index i

The first statement represents a generic predicate call and is described later in Sect. 2.2. All other statements could be seen as special cases of it, representing calls to built-in predicates. Statement (2) is a call to the "=" predicate, that checks whether its two inputs are equal. Similarly, (3) is a call to the assignment predicate ":=". Both are generic and can be applied on any supported type of the language.

The statements (4)–(8) are structure-related. (4) creates a new structure s with $e_1; \ldots; e_n$ as field values. (5) returns the values of all the fields of s in the output parameters $o_1; \ldots; o_n$. Statement (6) returns the value of the f_i field. As previously mentioned, we are focusing on a purely functional language and consider immutable algebraic data structures and arrays. Therefore, setting the value of a structure's field, shown in (7), returns a *new* structure where all fields have the same value as in s, except f_i which is set to e_i. Statement (8) verifies if the values of the given subset of fields of two structures s' and s'' are equal.

Statements (9) – (11) are variant-related. (9) creates a new variant v using the constructor C_p with e_p as an argument. Statement (10) is used for matching on the different constructors of an input variant v. Statement (11) verifies if the input variant v was created with one of the constructors in $\{C_1, \ldots, C_k\}$. This could be obtained with a *variant switch*, but for practical considerations it has been provided as a built-in predicate.

The last two statements are array-related. (12) returns the value of the i-th cell of the input array a. Similarly to (7), updating the i-th cell of an array – shown in (13) – has a functional nature. It returns a *new* array where all cells have the same value as in a, except the i-th cell which is set to e.

2.2 Exit Labels

Besides input and output parameters, the declaration of a predicate also includes a non-empty set of *exit labels*. When called, a predicate exits with one of the specified exit labels, thus summarizing and returning to its callers further information regarding its execution.

Table 2. Statements and their exit labels

Statement	Exit Labels	
$p(e_1, \ldots, e_n)$ $[\lambda_1 : \bar{o}_1 \mid \ldots \mid \lambda_m : \bar{o}_m]$	$\begin{bmatrix} \lambda_1 & \mapsto & \bar{o}_1 \\ \vdots & \ddots & \vdots \\ \lambda_m & \mapsto & \bar{o}_m \end{bmatrix}$	(1)
$e_1 = e_2$	$[\mathsf{true} \mapsto \emptyset, \mathsf{false} \mapsto \emptyset]$	(2)
$o := e$	$[\mathsf{true} \mapsto o]$	(3)
$s := \{e_1, \ldots, e_n\}$	$[\mathsf{true} \mapsto s]$	(4)
$\{o_1, \ldots, o_n\} := s$	$[\mathsf{true} \mapsto o_1 \ldots o_n]$	(5)
$o := s.f_i$	$[\mathsf{true} \mapsto o]$	(6)
$s' := \{s \textbf{ with } f_i = e_i\}$	$[\mathsf{true} \mapsto s']$	(7)
$s' = \langle f_1, \ldots, f_k \rangle s''$	$[\mathsf{true} \mapsto \emptyset, \mathsf{false} \mapsto \emptyset]$	(8)
$v := C_p[e_p]$	$[\mathsf{true} \mapsto v]$	(9)
switch(v) ***as*** $[o_1 \mid \ldots \mid o_n]$	$\begin{bmatrix} \lambda_{C_1} & \mapsto & o_1 \\ \vdots & \ddots & \vdots \\ \lambda_{C_n} & \mapsto & o_n \end{bmatrix}$ where $C_1, \ldots, C_n$ are the constructors of the type of variant v	(10)
$v \in \{C_1, \ldots, C_k\}$	$[\mathsf{true} \mapsto \emptyset, \mathsf{false} \mapsto \emptyset]$	(11)
$o := a[i]$	$[\mathsf{true} \mapsto o, \mathsf{false} \mapsto \emptyset]$	(12)
$a' := [a \textbf{ with } i = e]$	$[\mathsf{true} \mapsto a', \mathsf{false} \mapsto \emptyset]$	(13)

Exit labels constitute the main specificity of the language. They can denote different exceptional execution scenarios and act as exit codes, similarly to exceptions and exit status return values in other programming languages. For example, the predicate `run_thread(process pr, int new_id)` introduced in Sect. 1 has

two exit labels: true, corresponding to a successful execution and `invalid_id`, indicating that the given identifier is invalid. Labels also offer a convenient way to model a Boolean result. Frequently, a Boolean output value can be replaced by declaring two possible exit labels: true for a successful execution of the predicate and false for its opposite. This is illustrated by the previously defined property `disjoint_stacks(process pr)` (in Fig. 1b).

Exit labels play an important role with respect to control flow management. Complex control flow is expressed and directed by catching and transforming labels. Furthermore, they condition the existence of output parameters, as these are associated to the exit labels of a predicate. Whenever a predicate exits with an exit label λ, all the outputs associated to it are effectively produced, whereas all other outputs are discarded. If no output is associated to an exit label, it means that no output is generated when the predicate exits with this particular label. We can now explain the generic predicate call statement (1) from Table 1: the predicate p is called with inputs $e_1, \ldots, e_n$ and yields one of the declared exit labels $\lambda_1, \ldots, \lambda_n$, each having its own set of associated output variables $\bar{o}$.

As shown in Table 2, statement (10) will have an exit label corresponding to each constructor of the given input variant. Statements (2), (8) and (11) are bi-labeled, using true and false as logical values. Statements (12) and (13) are bi-labeled as well. However, unlike the previously mentioned statements, they use the label false as an "out of bounds" exception and generate an output only for the label true.

2.3 The Control Flow Graph

In the following we will work with a control flow graph representation of the predicates' bodies. The nodes represent program states, and the edges are defined by statements with a particular exit label λ.

The control flow graph $G_p = (N, E)$ of a predicate p has a node $n_i \in N$ for each program point. For each statement s at program point n_i that can execute and reach program point n_j with exit label λ_k, an edge (n_i, n_j) is added to G_p and labeled with s and λ_k. G_p has a single *entry node* $n_{in} \in N$ corresponding to the program point associated to the first statement of p. The set of *exit nodes* $n_{out} \subset N$ consists of the nodes associated to each possible exit label λ_k of the predicate.

In practice, all the outgoing edges of a node $n_i \in N$ bear the different cases of the same statement s found at program point n_i. Thus, the edges are labeled with the same statement s and there is an edge labeled s, λ_k for each possible exit label λ_k of s. However, the analysis does not depend on this special case.

The subfigures in Fig. 2 show the control flow graph of the following predicate: **predicate** `thread(process p, int i) -> [true: thread ti | None | oob]` which receives a process `p` and an index `i` as inputs and returns the `i`-th active thread of the input process (the `process` and `thread` types are defined in Fig. 1a). If the `i`-th thread is inactive, it exits with the exit label `None`. In the case of an "out of bounds" exception, the exit label `oob` is returned. For better readability, Fig. 2b gives the control flow of the same predicate where we

have labeled the nodes with statements of the predicate and the edges with their exit labels.

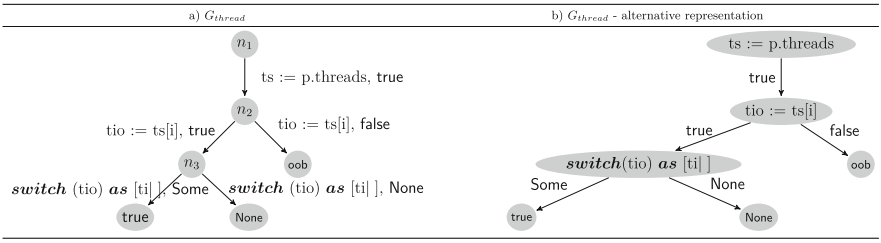


Fig. 2. Example – control flow graph of predicate `thread`

3 Abstract Domain of Dependencies

The goal of our analysis is to detect the input subset on which the outputs of a predicate *may* depend. More precisely, the analysis makes a conservative approximation and must guarantee that what is marked as not needed is definitely not needed.

The first step towards such results is the definition of an abstract domain of dependencies $\mathbb{D}$, shown below. The domain $\delta \in \mathbb{D}$ is defined inductively from the three atomic cases $\top$, $\oslash$ and $\bot$, mimicking the structure of the concrete types:

$$
\begin{array}{lll}
\delta := \mid \top \mid \oslash \mid \bot & \text{atomic cases} & \\
\quad \mid \{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\} & f_1, \ldots, f_n \text{ fields} & \text{(i)} \\
\quad \mid [C_1 \mapsto \delta_1; \ldots; C_m \mapsto \delta_m] & C_1, \ldots, C_m \text{ constructors} & \text{(ii)} \\
\quad \mid \langle \delta \rangle & & \text{(iii)} \\
\quad \mid \langle \delta_{default} \triangleright i \ : \ \delta_{exc} \rangle & i \text{ array index} & \text{(iv)}
\end{array}
$$

For atomic types the dependency is expressed in terms of the domain's atomic cases: $\top$ (least precise), denoting that *everything* is needed and $\oslash$, denoting that *nothing* is needed. The third atomic case $\bot$, denoting *impossible*, is explained below. The dependency of a structure (i) describes the dependency on each of its fields. For arrays we distinguish between two cases, namely arrays with a general dependency applying to all of the cells (iii) and arrays with a general dependency applying to all but one exceptional cell, for which a specific dependency is known (iv). For variants (ii), the dependency is expressed in terms of the dependencies of its constructors, expressed in terms of their arguments' dependencies. Thus, a constructor having a dependency mapped to $\oslash$ is one for which nothing but the tag has been read, i.e. its arguments, if any, are irrelevant for the execution. For variants, we also include the information that certain constructors cannot occur. The third atomic case – $\bot$ – is introduced for this purpose. We perform a "possible-constructors" analysis simultaneously, which computes for each execution scenario, the subset of possible constructors for a given value, at a given program point. All constructors that cannot occur are marked as being $\bot$. This

Table 3. $\sqsubseteq$ – comparison of two domains

$$\frac{}{\delta \sqsubseteq \top}\ \textsc{Top} \qquad \frac{\delta_1 \sqsubseteq \delta'_1 \quad \dots \quad \delta_n \sqsubseteq \delta'_n}{\{f_1 \mapsto \delta_1; \dots; f_n \mapsto \delta_n\} \sqsubseteq \{f_1 \mapsto \delta'_1; \dots; f_n \mapsto \delta'_n\}}\ \textsc{Str}$$

$$\frac{}{\bot \sqsubseteq \delta}\ \textsc{Bot} \qquad \frac{\delta_1 \sqsubseteq \delta'_1 \quad \dots \quad \delta_n \sqsubseteq \delta'_n}{[C_1 \mapsto \delta_1; \dots; C_n \mapsto \delta_n] \sqsubseteq [C_1 \mapsto \delta'_1; \dots; C_n \mapsto \delta'_n]}\ \textsc{Var}$$

$$\frac{\oslash \sqsubseteq \delta_1 \quad \dots \quad \oslash \sqsubseteq \delta_n}{\oslash \sqsubseteq \{f_1 \mapsto \delta_1; \dots; f_n \mapsto \delta_n\}}\ \oslash\textsc{Str} \qquad \frac{\oslash \sqsubseteq \delta_1 \quad \dots \quad \oslash \sqsubseteq \delta_n}{\oslash \sqsubseteq [C_1 \mapsto \delta_1; \dots; C_n \mapsto \delta_n]}\ \oslash\textsc{Var}$$

$$\frac{\delta_{def} \sqsubseteq \delta'_{def}}{\langle \delta_{def} \rangle \sqsubseteq \langle \delta'_{def} \rangle}\ \textsc{ADef} \qquad \frac{\delta_{def} \sqsubseteq \delta'_{def} \quad \delta_{exc} \sqsubseteq \delta'_{def}}{\langle \delta_{def} \triangleright i \ :\ \delta_{exc} \rangle \sqsubseteq \langle \delta'_{def} \rangle}\ \textsc{AIA} \qquad \frac{\delta_{def} \sqsubseteq \delta'_{def} \quad \delta_{def} \sqsubseteq \delta'_{exc}}{\langle \delta_{def} \rangle \sqsubseteq \langle \delta'_{def} \triangleright i :\ \delta'_{exc} \rangle}\ \textsc{AAI}$$

$$\frac{\delta_{def} \sqsubseteq \delta'_{def} \quad \delta_{exc} \sqsubseteq \delta'_{exc} \quad \delta_{def} \sqsubseteq \delta'_{exc} \quad \delta_{exc} \sqsubseteq \delta'_{def} \quad i \neq j}{\langle \delta_{def} \triangleright i \ :\ \delta_{exc} \rangle \sqsubseteq \langle \delta'_{def} \triangleright j :\ \delta'_{exc} \rangle}\ \textsc{AIJ}$$

$$\frac{\oslash \sqsubseteq \delta_{def}}{\oslash \sqsubseteq \langle \delta_{def} \rangle}\ \oslash\textsc{A} \qquad \frac{\delta_{def} \sqsubseteq \delta'_{def} \quad \delta_{exc} \sqsubseteq \delta'_{exc}}{\langle \delta_{def} \triangleright i \ :\ \delta_{exc} \rangle \sqsubseteq \langle \delta'_{def} \triangleright i :\ \delta'_{exc} \rangle}\ \textsc{AI} \qquad \frac{\oslash \sqsubseteq \delta_{def} \quad \oslash \sqsubseteq \delta_{exc}}{\oslash \sqsubseteq \langle \delta_{def} \triangleright i \ :\ \delta_{exc} \rangle}\ \oslash\textsc{AI}$$

atomic value is the lower bound of our domain and hence, the most precise value. The final abstract domain is a closure of all these combined recursively.

The partial order relation $\sqsubseteq \subseteq \mathbb{D} \times \mathbb{D}$ used to compare dependency domains is detailed in Table 3. The greatest element is $\top$ (Top) and $\bot$ is the least (Bot). Instances of identical structure and variant types are compared pointwise (Str, Var). For arrays without known exceptional dependencies we compare the default dependencies applying to all array cells (ADef). If exceptional dependencies are known for the same cell, these are additionally compared (AI). For arrays with known exceptional dependencies for different cells, we compare each dependency on the left-hand side with each one on the right-hand side (AIJ). The comparison of $\oslash$ with structures ($\oslash$Str), variants ($\oslash$Var) and arrays ($\oslash$A, $\oslash$AI) is a pointwise comparison between $\oslash$ and the dependency of each sub-element.

Table 4. Join operation

δ'	δ''	$\delta' \vee \delta''$
$\top \vee \delta$		$= \top$
$\bot \vee \delta$		$= \delta$
$\{f_1 \mapsto \delta_1; \dots; f_n \mapsto \delta_n\} \vee$	$\{f_1 \mapsto \delta'_1; \dots; f_n \mapsto \delta'_n\}$	$= \{f_1 \mapsto \delta_1 \vee \delta'_1; \dots; f_n \mapsto \delta_n \vee \delta'_n\}$
$[C_1 \mapsto \delta_1; \dots; C_n \mapsto \delta_n] \vee$	$[C_1 \mapsto \delta'_1; \dots; C_n \mapsto \delta'_n]$	$= [C_1 \mapsto \delta_1 \vee \delta'_1; \dots; C_n \mapsto \delta_n \vee \delta'_n]$
$\langle \delta_{def} \rangle \vee$	$\langle \delta'_{def} \rangle$	$= \langle \delta_{def} \vee \delta'_{def} \rangle$
$\langle \delta_{def} \rangle \vee$	$\langle \delta'_{def} \triangleright i :\ \delta'_{exc} \rangle$	$= \langle \delta_{def} \vee \delta'_{def} \triangleright i :\ \delta_{def} \vee \delta'_{exc} \rangle$
$\langle \delta_{def} \triangleright i \ :\ \delta_{exc} \rangle \vee$	$\langle \delta'_{def} \triangleright j :\ \delta'_{exc} \rangle$	$\begin{cases} i = j & = \langle \delta_{def} \vee \delta'_{def} \triangleright i :\ \delta_{exc} \vee \delta'_{exc} \rangle \\ i \neq j & = \langle \delta_{def} \vee \delta_{exc} \vee \delta'_{def} \vee \delta'_{exc} \rangle \end{cases}$
$\oslash \vee \oslash$		$= \oslash$

The defined *join* operation $\vee : \mathbb{D} \times \mathbb{D} \rightarrow \mathbb{D}$ is detailed in Table 4. It is a *commutative* operation for which the undisplayed cases are defined with respect to their symmetrical counterparts. The operation is total: joining incompatible domains such as a structure and a variant or two structures having different field identifiers, results in $\top$. Join is applied *pointwise* on each sub-element; $\bot$ is its *identity* element and $\top$ is its *absorbing* element. Joining $\oslash$ and the dependency of a structure, variant or array is applied pointwise. The value obtained by joining δ and δ' is an upper bound of the two:

$$\delta \sqsubseteq \delta \vee \delta' \quad \wedge \quad \delta' \sqsubseteq \delta \vee \delta', \quad \forall\, \delta, \delta' \in \mathbb{D}.$$

It is not a *least upper bound* as a consequence of the *non-monotonic* approximations made for arrays (rule AIJ).

Besides *join*, a reduction operator $\oplus : \mathbb{D} \times \mathbb{D} \rightarrow \mathbb{D}$ has been defined as well. This is a recursive, commutative, pointwise operation. The need for such an operator is a consequence of the possible-constructors analysis that we perform simultaneously. Following the same execution path, the same constructors must be possible. Thus, the reduction operator is used in order to combine dependencies on the same execution path and consists in performing the intersection of constructors in the case of variants and the union of dependencies for all other types. Its *identity* element is $\oslash$ and its *absorbing* element is $\bot$. The reduce operator between $\top$, and the dependency of a structure, variant or array is applied pointwise. Two instances of identical variant types are pointwise reduced.

Finally, the projections summarized in Table 5, have been defined on a dependency domain δ and are used to express the data-flow equations of Sect. 4:

$.\mathrm{f} \quad : \mathbb{D} \rightarrow \mathbb{D}$ projection of a field's dependency
$@\mathrm{C} \quad : \mathbb{D} \rightarrow \mathbb{D}$ projection of a constructor's dependency
$\langle i \rangle \quad : \mathbb{D} \rightarrow \mathbb{D}$ projection of a cell's dependency
$\langle * \setminus i \rangle : \mathbb{D} \rightarrow \mathbb{D}$ projection of an array's dependency outside cell i
$\langle * \rangle \quad : \mathbb{D} \rightarrow \mathbb{D}$ projection of an array's general dependency

Table 5. Dependency projections

$\delta.f, f \in \mathscr{F}$	$\delta@C, C \in \mathscr{C}$
$\{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\}.f = \delta_i$ if $f = f_i$	$[C_1 \mapsto \delta_1; \ldots; C_m \mapsto \delta_m]@C = \delta_j$ if $C = C_j$

$\delta\langle * \setminus i \rangle$	$\delta\langle i \rangle$	$\delta\langle * \rangle$
$\langle \delta_{def} \rangle \langle * \setminus i \rangle = \delta_{def}$ $\langle \delta_{def} \triangleright k : \ \delta_{exc} \rangle \langle * \setminus i \rangle =$ $\begin{cases} \delta_{def} & \text{when } i = k \\ \delta_{def} \vee \delta_{exc} & \text{otherwise} \end{cases}$	$\langle \delta_{def} \rangle \langle i \rangle = \delta_{def}$ $\langle \delta_{def} \triangleright k : \ \delta_{exc} \rangle \langle i \rangle =$ $\begin{cases} \delta_{exc} & \text{when } i = k \\ \delta_{def} \vee \delta_{exc} & \text{otherwise} \end{cases}$	$\langle \delta_{def} \rangle \langle * \rangle = \delta_{def}$ $\langle \delta_{def} \triangleright k : \ \delta_{exc} \rangle \langle * \rangle = \delta_{def} \vee \delta_{exc}$

They are partial functions, and can only be applied on domains of the corresponding kind. For instance, the field projection $.f$ only makes sense for atomic

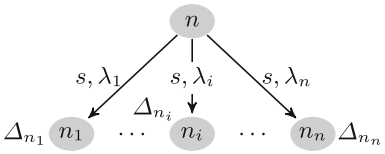
domains or structured domains with a field named f, which should be the case if the domain represents a variable of a structured type with some field f. For any of the atomic domains δ_a, applying any of the defined projections yields δ_a.

4 Intraprocedural Analysis and Data-Flow Equations

Intraprocedural Domains. Dependency information has to be kept at each point of the control flow graph, for each variable of the environment Γ, that maps input, output and local variables to their types. An *intraprocedural* domain $\Delta : \mathcal{V} \to \delta$ is thus a mapping from variables to dependencies, and is associated to every node of the control flow graph, representing the dependencies at the node's entry point. A special case is the mapping which maps all variables to $\bot$, which we call *Unreachable*. In particular it is associated to nodes that cannot be reached during the analysis. Also, if any of the variables of Δ is marked as $\bot$, the entire node collapses, becoming *Unreachable*.

For any node of the control flow graph associated to an intraprocedural domain Δ, $\Delta(x)$ retrieves the dependency associated to the variable x. If a mapping for x does not currently exist, $\Delta(x)$ retrieves $\oslash$. Forgetting a variable x from a reachable intraprocedural domain, $\Delta \setminus x$, removes its mapping. The $\vee$, $\sqsubseteq$ and $\oplus$ operations are extended pointwise to an intraprocedural domain, for each variable and its associated dependency domain δ_v. In particular, *Unreachable* is the bottom of this intraprocedural lattice.

Table 6. Statements – representations and data-flow equations

Representation	Equation
	$\Delta_n = \bigvee_{n \xrightarrow{s,\lambda_i} n_i} [\![s]\!]\lambda_i(\Delta_{n_i})$

Data-Flow Equations. Our dependency analysis is a *backward* data-flow analysis. For each exit label, it traverses the control flow graph starting with its corresponding exit node and marking all other exit points as *Unreachable*. The intraprocedural domain for the currently analyzed label is initialized with its associated output variables mapped to $\top$. The analysis traverses the control flow graph and gradually refines the dependencies until a fixed point is reached. Table 6 summarizes the representation and general equation of the statements. For each statement, the presented data-flow equation operates on the intraprocedural domains of the statement's *successor* nodes. The intraprocedural domain at the *entry point* of the node is obtained by *joining* the contributions of each

outgoing edge. The contribution of an edge (n_i, n_j) labeled with s and λ is given by $[\![s]\!]_\lambda(\Delta_{n_j})$ where $[\![s]\!]_\lambda(.)$ is the *transfer function* of the edge labeled s, λ.

Tables 7, 8, 9, 10 define the transfer functions for each built-in statement of our language, whereas the general case of a predicate call and its corresponding equation will be detailed in the following section.

Table 7 presents the transfer functions for statements which are not type-specific. For equality tests (1) both of the inputs e_1, e_2 are completely read, whether the test returns true or false. The transfer functions therefore, reduce the domain of the corresponding successor node with a domain consisting of e_1 and e_2 both mapped to $\top$. In the case of assignment (2), the dependency of the written output variable o is forgotten from the successor's intraprocedural domain, thus being mapped to $\oslash$ and forwarded to the input variable e.

Table 7. Generic statements – data-flow equations

Statement	$[\![s]\!]_{\lambda_i}(\Delta)$	
Equality test (1)	$[\![e_1 = e_2]\!]_{\mathsf{true}}(\Delta) = \Delta \oplus dep$ $[\![e_1 = e_2]\!]_{\mathsf{false}}(\Delta) = \Delta \oplus dep$	where $dep = \left\{ \begin{matrix} e_1 \mapsto \top \\ e_2 \mapsto \top \end{matrix} \right\}$
Assignment (2)	$[\![o := e]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus o) \oplus \{e \mapsto \Delta(o)\}$	

The data-flow equations given in Table 8 correspond to structure-related statements. For the Eqs. (3), (4), (5) and (6) we assume that the variable s is of type $\boldsymbol{structure}\{f_1 : t, \ldots, f_n : t\}$ for some fields f_i, $1 \leq i \leq n$. The equation (3) refers to the creation of a structure: each input e_i is read as much as the corresponding field f_i of the structure is read. The destructuring of a structure is handled in (4): each field f_i is needed as much as the corresponding variable o_i is. When accessing the i-th field of a structure s (5), only the field f_i is read, and only as much as the access' result o itself. The equation (6) treats field updates: the variable e_i is read as much as the field f_i is. The structure s is read as much as all the fields other than f_i are read in s'. Finally, the equations given in (7) handle partial structure equality tests, and the transfer functions are the same for the labels true or false: for both compared structures s' and s'', all the fields in the given set $f_1, \ldots, f_k$ are completely read, and only those.

The data-flow equations given in Table 9 correspond to variant-related statements. They follow the same principles as those used for structure-related statements above. Note that the transfer functions for the switch (9) and possible constructor test (10) introduce $\bot$ dependencies for constructors which are known to be impossible on the considered edge. In particular, since $\bot$ is an absorbing element for $\oplus$, these transfer functions erase, for every constructor which is known to be locally impossible, all the dependency information possibly attached to said constructor in the successor nodes. This is the actual *raison d'être* for the reduction operator, since using $\vee$ to combine a successor domain and a local contribution would lose this information.

Table 8. Structure-related statements – data-flow equations

Statement		$[\![s]\!]_{\lambda_i}(\Delta)$
Create	(3)	$[\![s := \{e_1, \ldots, e_n\}]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus s) \oplus \bigoplus_{1 \le i \le n} \{e_i \mapsto \Delta(s).f_i\}$
Destruct	(4)	$[\![\{o_1, \ldots, o_n\} := s]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus \{o_i \mid o_i \in \bar{o}\}) \oplus \{s \mapsto \{f_i \mapsto \Delta(o_i) \mid 1 \le i \le n\}\}$
Access field	(5)	$[\![o := s.f_i]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus o) \oplus \{s \mapsto \{f_1 \mapsto \oslash; \ldots; f_i \mapsto \Delta(o); \ldots; f_n \mapsto \oslash\}\}$
Update field	(6)	$[\![s' := \{s \textbf{ with } f_i = e_i\}]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus s') \oplus \left\{ \begin{array}{l} e_i \mapsto \Delta(s').f_i \\ s \mapsto \{f_j \mapsto \delta_j \mid 1 \le j \le n\} \end{array} \right\}$ where $\delta_j = \left\{ \begin{array}{ll} \Delta(s').f_j & \text{if } j \neq i \\ \oslash & \text{otherwise} \end{array} \right.$
Equality	(7)	$[\![s' = \langle f_1, \ldots, f_k \rangle s'']\!]_{\mathsf{true}}(\Delta) = \Delta \oplus d$ where $d = \left\{ \begin{array}{l} s' \mapsto \{f_i \mapsto \delta_i \mid 1 \le i \le n\} \\ s'' \mapsto \{f_i \mapsto \delta_i \mid 1 \le i \le n\} \end{array} \right\}$
		$[\![s' = \langle f_1, \ldots, f_k \rangle s'']\!]_{\mathsf{false}}(\Delta) = \Delta \oplus d$ $\delta_i = \left\{ \begin{array}{l} \top \text{ if } f_i \in \{f_1, \ldots, f_k\} \\ \oslash \text{ otherwise} \end{array} \right.$

Table 9. Variant-related statements – data-flow equations

Statement		$[\![s]\!]_{\lambda_i}(\Delta)$
Create variant	(8)	$[\![v := C_p[e_p]]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus v) \oplus \{e_p \mapsto \Delta(v)@C_p\}$
Variant Switch	(9)	$[\![\textbf{switch}(v) \textbf{ as } [o_1 \mid \ldots \mid o_n]]\!]_{\lambda_i}(\Delta) = (\Delta \setminus o_i) \oplus \{v \mapsto dep_i\}$ where $dep_i = [C_1 \mapsto \bot; \ldots; C_i \mapsto \Delta(o_i); \ldots; C_n \mapsto \bot]$
Possible variant	(10)	$[\![v \in \{C_1, \ldots, C_k\}]\!]_{\mathsf{true}}(\Delta) = \Delta \oplus \{v \mapsto [C_i \mapsto \delta_i \mid 1 \le i \le n\,]\}$ where $\delta_i = \left\{ \begin{array}{ll} \Delta(v)@C_i & \text{if } C_i \in \{C_1, \ldots, C_k\} \\ \bot & \text{otherwise} \end{array} \right.$
		$[\![v \in \{C_1, \ldots, C_k\}]\!]_{\mathsf{false}}(\Delta) = \Delta \oplus \left\{v \mapsto \left[\bar{C}_i \mapsto \bar{\delta}_i \mid 1 \le i \le n\,\right]\right\}$ where $\bar{\delta}_i = \left\{ \begin{array}{ll} \Delta(v)@\bar{C}_i & \text{if } \bar{C}_i \notin \{C_1, \ldots, C_k\} \\ \bot & \text{otherwise} \end{array} \right.$

Finally, the equations for array-related statements are given in Table 10. We assume for both that the context is fixed and that $\mathcal{I}$ is the distinguished set of input variables for the analyzed predicate. This set is used to make sure that exceptions in array dependencies are only registered to variables in $\mathcal{I}$ and not local or output variables. The reason for such a constraint is a pragmatic one: input variables are not assignable in our language, and therefore they always represent the same value intraprocedurally. Otherwise, each time a variable is written by a statement, we would need to traverse all the dependencies in the domain to erase or reinterpret the occurrences where this variable appears as an exception. Only recording exceptions for input variables makes this kind of costly traversal useless, and since only exceptions about input variables make sense at the interprocedural level (see Sect. 5), we do not lose much precision by doing so. The transfer functions for (11) and (12) thus take care of making adequate approximations when exceptions cannot be introduced. As for the cases when the array access exits with the false label, note that the contribution to the array a is $\langle \oslash \rangle$, which is strictly less precise than $\oslash$. The operation makes implicit bounds checking and this can thus be seen as accounting for the fact that no

cell in a has been read, but the "length" or "support" of a has been read, hence it would not be true to claim that the result of the statement did not depend on a at all. Similarly, a variant dependency $[C_1 \mapsto \oslash, \ldots, C_n \mapsto \oslash]$ mapping all constructors to nothing has not read any value in any of the constructors, but may still depend on the variant's constructor itself.

Table 10. Array-related statements – data-flow equations

Statement		$[\![s]\!]_{\lambda_i}(\Delta)$
Array access	(11)	$[\![o := a[i]]\!]_{\mathsf{true}}(\Delta) = \begin{cases} (\Delta \setminus o) \oplus \left\{ \begin{array}{l} i \mapsto \top \\ a \mapsto \langle \oslash \triangleright i : \Delta(o) \rangle \end{array} \right\} & \text{when } i \in \mathcal{I} \\ (\Delta \setminus o) \oplus \left\{ \begin{array}{l} i \mapsto \top \\ a \mapsto \langle \Delta(o) \vee \oslash \rangle \end{array} \right\} & \text{when } i \notin \mathcal{I} \end{cases}$
		$[\![o := a[i]]\!]_{\mathsf{false}}(\Delta) = \Delta \oplus \left\{ \begin{array}{l} i \mapsto \top \\ a \mapsto \langle \oslash \rangle \end{array} \right\}$
Array update	(12)	$[\![a' := [a \; \textbf{\textit{with}} \; i = e]]\!]_{\mathsf{true}}(\Delta) = \begin{cases} (\Delta \setminus a') \oplus \left\{ \begin{array}{l} i \mapsto \top \\ e \mapsto \Delta(a')\langle i \rangle \\ a \mapsto \langle \Delta(a')\langle * \setminus i \rangle \triangleright i : \oslash \rangle \end{array} \right\} & \text{when } i \in \mathcal{I} \\ (\Delta \setminus a') \oplus \left\{ \begin{array}{l} i \mapsto \top \\ e \mapsto \Delta(a')\langle * \rangle \\ a \mapsto \langle \Delta(a')\langle * \rangle \vee \oslash \rangle \end{array} \right\} & \text{when } i \notin \mathcal{I} \end{cases}$
		$[\![a' := [a \; \textbf{\textit{with}} \; i = e]]\!]_{\mathsf{false}}(\Delta) = \Delta \oplus \left\{ \begin{array}{l} i \mapsto \top \\ a \mapsto \langle \varnothing \rangle \end{array} \right\}$

5 Interprocedural Dependencies

Exit labels, presented in Sect. 2.2, constitute an increased source of expressivity, as they indicate the scenario that was observed while executing a predicate. We incorporate this expressivity in our dependency results, by computing specific dependencies for each possible execution scenario. Therefore, our analysis is performed label by label and interprocedural dependency domains associate an intraprocedural domain to each exit label of the analyzed predicate. The variable key-set of each associated intraprocedural domain comprises the inputs of the analyzed predicate. A label that cannot be returned is mapped to an *Unreachable* intraprocedural domain. This is a form of *path-sensitivity* [10]. However, we favor the term *label-sensitivity* for this characteristic, as it seems to be a more natural choice applied to our case and the language we are working on.

An interprocedural domain of a predicate p is thus defined as follows:

$$\mathscr{D}_p : \Lambda_p \to \Delta, \quad \text{where } \Lambda_p \text{ the set of output labels of predicate } p$$

For each analyzed label of a predicate, the analysis starts by initializing the intraprocedural domain mapped to it, with the output variables associated to the exit label. To avoid making any false supposition, these are initially mapped to the most general dependency, namely $\top$. Subsequently, as described in Sect. 4, the dependency information is gradually refined until a fixed point is reached. The execution scenarios denoted by the exit labels of a predicate are mutually

exclusive. Therefore, during the analysis of a particular exit label, all other exit labels of the predicate are mapped to *Unreachable*. After reaching a fixed point, the intraprocedural domain is *filtered* so that only input variables appear in the variable set. As explained in Sect. 4, the intraprocedural domains are built such that only input variables may appear as exception indices in dependencies computed for arrays. This invariant is preserved throughout the analysis.

A substitution must be performed on interprocedural domains. This consists in substituting all occurrences of formal input parameters of a predicate by the corresponding effective input parameters. The substitution operation is denoted by $\blacktriangleleft\,(\sigma)$ where σ is a substitution from formal to effective parameters.

We proceed by detailing the equation corresponding to a call to a predicate:

$$p(e_1, \ldots, e_n)[\lambda_1 : \bar{o}_1 \mid \ \ldots \ \mid \lambda_m : \bar{o}_m]$$

having the following signature:

$$p(\epsilon_1, \ldots, \epsilon_n)[\lambda_1 : \bar{\omega}_1 \mid \ \ldots \ \mid \lambda_m : \bar{\omega}_m]$$

The general equation form applies:

$$\Delta_n = \bigvee_{n \xrightarrow{s,\lambda_i} n_i} [\![p(e_1, \ldots, e_n)\ [\lambda_1 : \bar{o}_1 \mid \ldots \mid \lambda_m : \bar{o}_m]]\!]_{\lambda_i}(\Delta_{n_i})$$

The transfer functions for the predicate call statement are deduced from the predicate's interprocedural domain in the following fashion:

$$[\![p(e_1, \ldots, e_n)\ [\lambda_1 : \bar{o}_1 \mid \ldots \mid \lambda_m : \bar{o}_m]]\!]_{\lambda_i}(\Delta) = (\Delta \setminus \bar{o}_i) \bigoplus_{j \in \{1,\ldots,n\}} e_j \mapsto dep_j^i$$

where $\quad dep_j^i = \mathscr{D}_p(\lambda_i)(\epsilon_j) \blacktriangleleft (\bar{\epsilon} \mapsto \bar{e})$

Namely, the mappings for the outputs $\bar{o}$ associated to a label λ_i are removed, and the contribution of a call to each input e_j stems from the contribution of the interprocedural domain for label λ_i and formal input ϵ_j. In these, all the formal input parameters $\bar{\epsilon}$ in array dependency domains are substituted by the corresponding effective input parameters from $\bar{e}$.

Semantics. We conclude this section by briefly presenting the two possible interpretations of the results of our analysis. Considering an intraprocedural result Δ_p^λ for a predicate p and label λ, a first interpretation of our dependency analysis is an equivalence relation on tuples of values $\approx_{\Delta_p^\lambda}$ which relates values that only differ in places on which p, λ does not depend. It can be used for applying *congruence modulo* reasoning to predicate calls. Namely, if we write $p(\bar{v}) \xrightarrow{\lambda:\bar{w}}$ to denote that applying p to the values $\bar{v}$ yields the exit label λ with outputs $\bar{w}$, then if p is applied in turn to two input data structures $\bar{u}$ and $\bar{v}$ that are congruent w.r.t. $\approx_{\Delta_p^\lambda}$, the predicate will exercise the same execution scenario:

$$\bar{u} \approx_{\Delta_p^\lambda} \bar{v} \implies p(\bar{u}) \xrightarrow{\lambda:\bar{w}} \implies p(\bar{v}) \not\xrightarrow{\mu:\bar{z}}$$

Furthermore, identical outputs will be obtained:

$$\bar{u} \approx_{\Delta_p^\lambda} \bar{v} \implies p(\bar{u}) \xrightarrow{\lambda:\bar{w}} \implies p(\bar{v}) \xrightarrow{\lambda:\bar{z}} \implies \bar{w} = \bar{z}$$

whereas this first interpretation focuses on the dependency part of the analysis, it is also possible to focus on the possible constructors part of the analysis. This additional interpretation is a characteristic function $\mathbb{1}_{\Delta_p^\lambda}$ on input values which constrains the space of inputs that can make p exit with label λ. It denotes the necessary conditions on inputs according to the observed execution scenario and can be used as an *inversion lemma* when reasoning on calls to a predicate:

$$p(\bar{u}) \xrightarrow{\lambda:\bar{w}} \implies \mathbb{1}_{\Delta_p^\lambda}(\bar{u})$$

A detailed presentation of these semantics is out of the scope of this paper but in order to give a good intuition of the adequation between the interpretation and the lattice operations described in Sect. 3, we can give some fundamental properties relating the domain operations and these interpretations:

$$\bar{v} \approx_\top \bar{w} \iff \bar{v} = \bar{w} \qquad \bar{v} \approx_\varnothing \bar{w} \wedge \bar{v} \approx_\bot \bar{w} \quad \forall \bar{v}, \bar{w} \qquad \mathbb{1}_\top = \mathbb{1}_\varnothing = \_ \mapsto 1$$

$$\mathbb{1}_\bot = \_ \mapsto 0 \qquad \Delta \sqsubseteq \Delta' \implies \approx_\Delta \supseteq \approx_{\Delta'} \qquad \Delta \sqsubseteq \Delta' \implies \mathbb{1}_\Delta \subseteq \mathbb{1}_{\Delta'}$$

$$\approx_{\Delta \oplus \Delta'} \subseteq \approx_\Delta \cap \approx'_\Delta \qquad \mathbb{1}_\Delta \wedge \mathbb{1}'_\Delta \implies \mathbb{1}_{\Delta \oplus \Delta'}$$

The soundness of the second interpretation as well as the *well-formedness* of our dependency domains have been proven in Coq[2].

6 Preliminary Results and Experiments

Our analysis has currently been applied on two abstract layers of *ProvenCore*, described in Sect. 1. These are the *Refined Security Model* (RSM), an abstract layer situated just underneath the top-most layer of the refinement chain and the *Functional Specifications* (FSP) layer, a model closely resembling the most concrete layer (*Target of Evaluation Design* – TDS) but using data structures and algorithms that facilitate reasoning. Each layer is characterized by a global state with numerous fields and different transitions, i.e. supported commands. Various invariants and properties characterize their states. For example, the FSP's state contains 14 fields; it is characterized by approximately 50 invariants. In the TDS, these figures are doubled. Each invariant is concerned with a different subset of the global state's fields. Some of the invariants concern all the processes held in the process store. However, most transitions affect only a few of these fields. We have applied our analysis at a medium-scale on the RSM and FSP layers. The results for over 660 predicates having approximately 11000 lines of code have been computed in 1.5 s.

One of the analyzed predicates from the FSP layer is:
predicate `proc_mem_auth_ok(proc proc) -> [true | false]` verifying a fundamental property that has to hold for all processes in the process store of `proc`. It

[2] The corresponding files are provided: http://ajl2015.ddns.net/ajl2015/proveCoq.

refers to the *relevance of memory permissions* and states that every process has permissions covering a valid range of memory addresses inside its virtual space. The `process` type is a structure with 26 fields, of which 11 are compound data types themselves. Among these, 2 fields are arrays, 3 fields are variants and 6 others are structures with a number of fields ranging from 3 to 17 fields.

The dependency results computed by our analysis for this predicate are shown below. The analysis detects that for each of the possible execution scenarios, the outcome depends only on 2 out of the 26 fields, namely the stackframe and the memory permissions. The dependency on the `stackframe` is confined to only one of the 3 fields: the data and stack segment. The memory permissions are given by a variant with 3 constructors, denoting reading and writing permissions or the absence of any permission. Furthermore, besides pinning down the outcome's dependency on 2 out of the 26 fields of the `process` structure, the analysis also detects that the absence of any memory permission, indicated by the constructor `NONE` of the `mem_auth` variant, is $\bot$ for the false execution scenario. In other words, unused permissions cannot threaten the property `proc_mem_auth_ok`.

$$
\begin{array}{lllll}
\textbf{false} \to \{proc \to \{ & mem\_auth \to [& READ & \to \{ & base \to \top;\ len \to \top\} \\
 & & WRITE & \to \{ & base \to \top;\ len \to \top\} \\
 & & NONE & \to & \bot] \\
 & stackframe \to \{ & ds & \to & \top\}\}\} \\
\textbf{true} \to \{proc \to \{ & mem\_auth \to [& READ & \to \{ & base \to \top;\ len \to \top\} \\
 & & WRITE & \to \{ & base \to \top;\ len \to \top\} \\
 & & NONE & \to & \oslash] \\
 & stackframe \to \{ & ds & \to & \top\}\}\}
\end{array}
$$

The *relevant memory permissions* property is thus only threatened by transitions that add memory permissions or change a process' virtual space layout. Only 3 transitions out of the 25 belong to this category: `exec` which resets the process' segments, `do_auth_read` and `do_auth_write` which add permissions. In particular, transitions deleting memory permissions do not impact the property since the absence of permissions, as shown by the dependency of the constructor `NONE` for the false label, is an impossible case when the property does not hold. This is one of the practical advantages of tracking constructor possibilities simultaneously.

Space constraints prevent us from discussing other examples here. However, various other examples are provided on the webpage[3] dedicated to our analysis.

7 Related Work

The *frame problem* and its manifestations in the software verification process – detecting program properties that remain unchanged under a certain operation – are notorious. First described in 1969 [8], the frame problem is still a target for full automation in the software verification realm. A *complete specification* of a program will necessarily include *frame properties*. However, though necessary, frame properties are tedious and repetitive. Two prominent solutions to the

[3] http://ajl2015.ddns.net/ajl2015/.

frame problem come from *separation logic* [4] and *ownership types* [1]. However, it is argued that the problem itself should not impose such annotation-heavy solutions. Simpler, automatic solutions for their specification and verification would allow programmers to concentrate on the truly challenging part [9].

The dependency results computed by our analysis are similar to *primitive read* and *write effects* used in ownership type systems [1]. Write effects in our case are implicit and include strictly the output variables associated to an exit label. Read effects can only refer to input variables of a predicate. Also, read effects comprise the whole execution of a method even if they are irrelevant for the method's results. We however, ignore read effects on which the output does not depend, reflecting only those which contribute to the observed result. A technique for declaring and verifying read effects in an ownership type system is presented in [1]. We use static analysis to automatically detect them.

Our dependencies are similar to the *influence sets* presented by Leino and Müller [5]. Influence sets are represented as sets of heap locations and they are used to specify the parts of the program state that are allowed to impact the return values. Reasoning about heap locations is beyond the scope of our analysis. We treat mappings between variables and values, analyze their evolution in a side-effect free environment and express dependencies as input-output relations.

Static dependence or liveness analyses are typically used for code optimization, dead code elimination [7] and compile time garbage collection, but only seldom for program verification. One case we are aware of comes from Frama-C [2], where it is used in a purely *automatic* setting and unlike our analysis it does not handle unions and arrays. A plug-in based on the available value analysis [3] computes lists of input and output locations for each function, distinguishing between *operational*, *functional* and *imperative inputs*.

8 Conclusion

We have presented a *flow-sensitive*, *path-sensitive*, interprocedural dependency analysis that handles arrays, structures and variants. For the latter it simultaneously computes a subset of possible constructors. We have defined our own abstract dependency domain and obtain dependency information that mirrors the layered structure of compound data types.

The main original traits of our contribution stem from its design as an analysis meant to be used as a companion tool during interactive program verification, in an unified manner on programs as well as on specifications.

An obvious first challenge is to address the issue of *context-sensitivity*. We plan to introduce lazy components in our interprocedural dependency summaries and to inject in them the current intraprocedural context on an as-needed basis. Early experiments show much promise in terms of improved precision, with only a marginal decrease in performance.

Our long-term goal is to combine the dependency analysis with a correlation analysis, meant to detect relations between inputs and outputs. By uncovering relations (preorders and equivalences) between inputs and outputs, after having

detected that a property only depends on unmodified parts and unifying the results, the preservation of invariants for the unmodified parts could be inferred.

Acknowledgments. We would like to thank the anonymous referees for helpful comments and suggestions. For his excellent comments and sharp observations, we are particularly grateful to Olivier Delande. Our article also benefited from the remarks of P. Bolignano, G. Dupéron, L. Hubert and B. Montagu.

References

1. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 2002, pp. 292–310. ACM, New York, NY, USA (2002)
2. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012)
3. Cuoq, P., Prevosto, V., Yakobowski, B.: Frama-c value analysis manual. http://frama-c.com/download/value-analysis-Neon-20140301.pdf
4. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
5. Leino, K.R.M., Müller, P.: Verification of equivalent-results methods. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 307–321. Springer, Heidelberg (2008)
6. Lescuyer, S.: ProvenCore: Towards a verified isolation micro-kernel (2015)
7. Liu, Y., Stoller, S.: Eliminating dead code on recursive data. Sci. Comput. Program. **47**(2–3), 221–242 (2003). (special Issue on Static Analysis (SAS 1999))
8. Mccarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. In: Machine Intelligence, pp. 463–502. Edinburgh University Press (1969)
9. Meyer, B.: Framing the frame problem. In: Pretschner, A., Broy, M., Irlbeck, M. (eds.) Dependable Software Systems, Proc. of August 2014 Marktoberdorf Summer School. pp. 174–185. D: Information and Communication Security, Springer (2015)
10. Robert, V., Leroy, X.: A formally-verified alias analysis. In: Hawblitzel, C., Miller, D. (eds.) CPP 2012. LNCS, vol. 7679, pp. 11–26. Springer, Heidelberg (2012)