Correlating Structured Inputs and Outputs in Functional Specifications

Oana Fabiana 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. We present a static correlation analysis that computes a safe approximation of what part of an input state of a function is copied to the output state. This information is to be used by an interactive theorem prover to automate the discharging of proof obligations concerning unmodified parts of the state. The analysis is defined for a strongly-typed, functional language that handles structures, variants and arrays. It uses partial equivalence relations as approximations of fine-grained correlations between inputs and outputs. The analysis is interprocedural and summarizes not only what is modified but also how and to what extent. We have applied it to a functional specification of a microkernel, and obtained results that demonstrate both its precision and its scalability.

1 Introduction

Any complete formal software verification endeavour focuses on two fundamental, mutually dependent questions: what are the effects of program operations on their environment, i.e. what do program operations do, and what do they leave unmodified, i.e. what are they *not* doing. The latter concern inevitably leads to some manifestation of the *frame problem* [8], imposing superfluous manual verification effort and having notoriously tedious consequences. These are particularly visible in the context of complex transitions systems, which consist of complex states and transitions between them, i.e. state changes. States are defined using associative arrays and algebraic data types (structures and variants). Transitions map an input state to an output state. In reality, the transitions' effects are often restricted to a small subset of the state, thus impacting only a limited number of invariants simultaneously. However, a considerable amount of time is spent on proof obligations concerning unmodified parts. Though intuitively easy, these are in practice a lengthy and repetitive task. Specifying and proving the preservation of logical properties for the unmodified part thus becomes a natural target for automation [9]. We propose to tackle the inference of preserved invariants for the unmodified parts by answering the following two questions, by means of *static analysis*:

R. De Nicola and E. Kühn (Eds.): SEFM 2016, LNCS 9763, pp. 85–103, 2016.
DOI: 10.1007/978-3-319-41591-8_7

(1) What is the input subset on which a logical property depends?
(2) How does the output relate to the input of an operation?

In [1], we have presented a static dependency analysis that addresses the first question and automatically determines the input subsets on which a property depends. This paper deals with the second question. More specifically, given an operation that manipulates a structured input, we strive to determine the subset that remains unchanged and is propagated into the output. Our goal is thus to summarize the behaviour of an operation by computing relations between parts of the input and parts of the output. To this end, we present a *correlation analysis*, meant to be used in an interactive verification context, that tracks the origin of subparts of the output and relates it to subparts of the input. The analysis produces expressive results without sacrificing scalability. By unifying these correlation and dependency results and thus by knowing the effects of an operation, after having detected that a property only depends on unmodified parts, the preservation of some invariants can be inferred.

1.1 Motivating Example

The motivation and ideas behind the correlation analysis presented in this paper stemmed from the formal verification of *ProvenCore* [7], a full-featured industrial isolation micro-kernel. To exemplify the addressed problem and the fine-grained correlation results that we are targeting, we consider an abstract process manager and the data structures for its fundamental components: process and thread, shown in Fig. 1-a. 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 t) | None`.

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

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

a) Data Structures

```
predicate stop(proc in, int i) -> [true: proc o | inval]
```

b) Signature for the Example Function

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

The signature of a function `stop`, written in a modeling language that we present in Sect. 2, is shown in Fig. 1-b. It has two possible execution scenarios: `true`, when the given index `i` corresponds to an active thread, and `inval` otherwise. In the former case, `stop` copies the `i`-th element of the `threads` array to a local variable `th`, sets its state to `BLOCKED` and leaves everything else unmodified. The new state `o` of the process is then returned, with its `i`-th element set to `th` and everything else copied from `in`. The body of `stop` is detailed in Fig. 2.

Our analysis should infer that between the input process `in` and the output `o`, the values of the fields `pid`, `current_thread` and `address_space` are equal. Furthermore, it should detect that all elements of the array `threads` are equal, except the value of the `i`-th element, for which only the `current_state` differs.

By tracking only equalities between pairs of variables of the same type, we can detect the equality of the values of the `pid`, `current_thread` and `address_space` fields between the input and the output. However, if we ignore the flow of an input's subelement value to a variable (or conversely, the flow of a variable's value to an output's subelement) valuable information is lost. We are not only losing information between inputs and outputs of different types, but by accumulating imprecisions, we also lose information concerning inputs and outputs of the same type. This is exactly what can happen in our example. The equality between the values extracted from the input `in` and copied into `th` as well as the relation between the value of `th` and `o.threads[i]` are ignored because `th` is not of the same type as `in` and `o`. As a consequence, we lose the information concerning the relation between `in`'s and `o`'s `threads` value altogether. It is therefore imperative to track (cor)relations between variables of different types as well.

The contributions of this paper include an interprocedural domain and a static analysis that allow us to compute expressive correlations between parts of the inputs and parts of the outputs in a flexible manner. An in-depth presentation of these is given in Sect. 3. Results obtained on a functional specification of an operating system are discussed in Sect. 4.

2 Language

We briefly present the unified programming and specification language targeted by our analysis. This is an idealized version of a language developed at *Prove & Run*[1], designed with a focus on subsequent proof facilitation. It is a first-order, purely functional and strongly-typed language with algebraic data types and arrays. The basic building blocks of programs written in our language are *predicates*, the equivalent of functions in common programming languages.

2.1 Types and Statements

We let $\mathbb{T}$ be the universe of type identifiers and $T_0 \subset \mathbb{T}$ the set of base type identifiers. The sets of structure field identifiers and variant constructors are denoted by $\mathscr{F}$ and $\mathscr{C}$, respectively.

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

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

$$
\begin{array}{rlll}
\tau \in \mathbb{T}, \tau := & \mid \tau_0 \in T_0 & & \text{base types} \\
& \mid \boldsymbol{struct}\{f_1 : \tau, \ldots, f_n : \tau\} & f_i \in \mathscr{F}, 0 \leq n & \text{structures} \\
& \mid \boldsymbol{variant}[C_1 : \tau \mid \ldots \mid C_m : \tau] & C_i \in \mathscr{C}, 1 \leq m & \text{variants} \\
& \mid \boldsymbol{arr}^{\tau}\langle\tau\rangle & & \text{arrays}
\end{array}
$$

Variants and structures can be used together to model traditional algebraic variants with zero or several parameters. For instance, the `option_thread` type given in Sect. 1.1 is actually modeled as:

$$\boldsymbol{variant}[\texttt{Some} : \boldsymbol{struct}\{\texttt{t} : \texttt{thread}\} \mid \texttt{None} : \boldsymbol{struct}\{\}].$$

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. The first statement represents a generic predicate call and is described later. All other statements can be seen as special cases of it, representing calls to built-in predicates. They all have a functional nature and handle immutable data. Thus, setting the value of a structure's field, shown in (4), returns a *new* structure where all fields have the same value as in r, except f_i which is set to e. Similarly, updating the i-th cell of an array, shown in (8), 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

In addition to input and output parameters, the declaration of a predicate also includes a non-empty set of *exit labels*, which behave like *exit codes*. 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 1. Subset of 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
	$\mid o := e$	(2) assignment
	$\mid o := r.f_i$	(3) access field f_i
	$\mid r' := \{r\ \boldsymbol{with}\ f_i = e\}$	(4) update field f_i
	$\mid v := C_p[e]$	(5) create v with constructor C_p
	$\mid \boldsymbol{switch}(v)\ \boldsymbol{as}\ [o_1 \mid \ldots \mid o_n]$	(6) variant matching
	$\mid o := a[i]$	(7) array access at index i
	$\mid a' := [a\ \boldsymbol{with}\ i = e]$	(8) array update at index i

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)
$o := e$	$[\mathsf{true} \mapsto o]$	(2)
$o := r.f_i$	$[\mathsf{true} \mapsto o]$	(3)
$r' := \{r \textbf{ with } f_i = e\}$	$[\mathsf{true} \mapsto r']$	(4)

Statement	Exit Labels	
$v := C_p[e]$	$[\mathsf{true} \mapsto v]$	(5)
$\textbf{switch}(v)$ **as** $[o_1 \mid \ldots \mid o_n]$	$[\ldots \lambda_{C_i} \mapsto o_i \ldots]$ where $C_1, \ldots, C_n$ are the constructors of the type of variant v	(6)
$o := a[i]$	$[\mathsf{true} \mapsto o, \mathsf{false} \mapsto \emptyset]$	(7)
$a' := [a \textbf{ with } i = e]$	$[\mathsf{true} \mapsto a', \mathsf{false} \mapsto \emptyset]$	(8)

Exit labels play an important role for control flow management, which 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 $\lambda_1, \ldots, \lambda_m$, each having its own set of associated output variables $\bar{o}_1, \ldots, \bar{o}_m$, respectively.

As shown in Table 2, statement (6) has a label corresponding to each constructor of the input variant. Statements (7) and (8) are bi-labeled, using false as an "out of bounds" exception and generating an output only for the label true.

Figure 2 details the body of our example predicate from Sect. 1.1, where arrows show the control-flow between the various statements of the predicate.

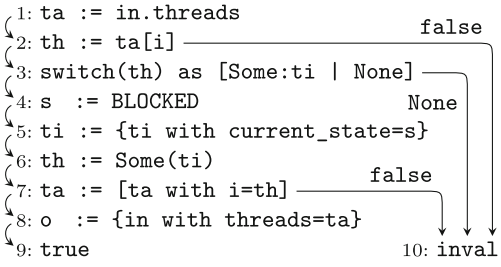


Fig. 2. Body of the `stop` predicate

3 Correlation Analysis

We present a flow-sensitive, conservative static analysis inferring what is modified by an operation and to *what* extent. It approximates the flow of input values into output values, by uncovering *equalities* and computing *correlations* as pairs between input parts and the output parts into which these are injected.

Outputs are often complex compounds of different subparts of different input variables: a subset of the input is modified, while the rest is injected as is. We track the origin of subparts of the output and relate it to subparts of the input. As previously explained in Sect. 1.1, we prevent avoidable over-approximations by considering pairs of different types and granularities. As a consequence, in order

to avoid dealing with data in a monolithic manner, we are forced to introduce an extra level of granularity below variables. At the intraprocedural level, illustrated in Fig. 3(a), we define the *correlation* domain as mappings between pairs of inputs and outputs to which we associate mappings between pairs of valid inner *paths* and the relations binding them. Correlations for arrays and variants are shown in Fig. 3(b, c).

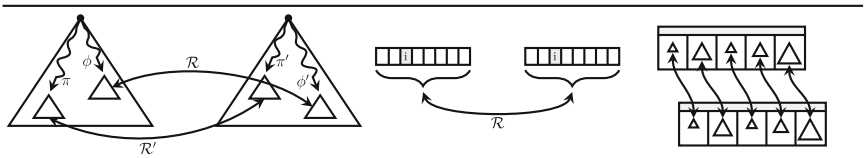


Fig. 3. Intraprocedural domain - general representation and examples

3.1 Partial Equivalence Relations

The first step towards automatically reasoning about the propagation of input subelements into output subelements is the definition of an abstract *partial equivalence type* $\mathscr{R}$ that mimics the structure of algebraic data types and arrays. A partial equivalence $\mathcal{R} \in \mathscr{R}$ is defined inductively from the two atomic elements, Equal and Any, and mirrors the structure of the concrete types:

$$
\begin{array}{rll}
\mathcal{R} := & \mid \mathsf{Equal} \mid \mathsf{Any} & \text{atomic cases} \\
& \mid \{f_1 \mapsto \mathcal{R}_1; \ldots; f_n \mapsto \mathcal{R}_n\} & f_1, \ldots, f_n \text{ fields} \quad \text{(i)} \\
& \mid [C_1 \mapsto \mathcal{R}_1; \ldots; C_n \mapsto \mathcal{R}_n] & C_1, \ldots, C_n \text{ constructors} \quad \text{(ii)} \\
& \mid \langle \mathcal{R}_{def} \rangle & \text{array} \quad \text{(iii)} \\
& \mid \langle \mathcal{R}_{def} \triangleright i \ :\ \mathcal{R}_{exc} \rangle & i \text{ array index} \quad \text{(iv)}
\end{array}
$$

Such relations represent fine-grained partial equivalences between pairs of values of the same type. Equal and Any represent respectively equal and unrelated values. Partial equivalence relations for structures (given by (i)) and for variants (given by (ii)), are expressed in terms of the partial equivalences of their subparts, by mapping each field or constructor to the corresponding relations. For arrays, we distinguish between two cases, namely arrays with a general relation applying to all of the cells (as given by (iii)) or to all but one exceptional cell (as given by (iv)), for which a specific relation is known.

Even if the syntactic partial equivalences are untyped, their interpretation is made in the context of a type $\tau \in \mathbb{T}$. The semantics of a partial equivalence $\mathcal{R}$ for a type τ is a partial equivalence relation over values of type τ. Cases other than Equal and Any only have non-empty interpretations for types τ which are compatible with their shape. For instance, the structured relation $\{f \mapsto \mathcal{R}\}$ only really makes sense for structured types with a single field f, whose type itself is

compatible with $\mathcal{R}$, and shall not be used in connection with variant or array types for example.

To describe the semantics of elements in $\mathscr{R}$, we define for each type τ the set $\mathbb{D}_\tau$ of semantic values of that type. For each primitive type $t \in T_0$, we suppose a given $\mathbb{D}_t$. Other semantic values are defined inductively as follows:

$$\begin{aligned}\mathbb{D}_{\textbf{\textit{struct}}\{f_1:\tau_1,\ldots,f_n:\tau_n\}} &= \{\{f_1 = v_1, \ldots f_n = v_n\} \mid \forall i, v_i \in \mathbb{D}_{\tau_i}\}\\ \mathbb{D}_{\textbf{\textit{variant}}[C_1:\tau_1 \mid \ldots \mid C_n:\tau_n]} &= \biguplus_{1\le i\le n}\{C_i[v_i] \mid v_i \in \mathbb{D}_{\tau_i}\}\\ \mathbb{D}_{\textbf{\textit{arr}}^{\tau_i}\langle\tau\rangle} &= \{(\mathcal{P}, (v_k)_{k\in\mathcal{P}}) \mid \mathcal{P} \subseteq \mathbb{D}_{\tau_i}, \forall k, v_k \in \mathbb{D}_\tau\}.\end{aligned}$$

Given a *valuation* E from variables to semantic values, the interpretation of a relation $\mathcal{R} \in \mathscr{R}$ with respect to some type τ is a binary relation over $\mathbb{D}_\tau$ defined as shown in Table 3.

Table 3. Partial equivalence relations – semantics

$$[\![\mathsf{Equal}]\!]^\tau = \{(x,x) \mid x \in \mathbb{D}_\tau\} \qquad [\![\mathsf{Any}]\!]^\tau = \mathbb{D}_\tau \times \mathbb{D}_\tau$$

$$[\![\{f_1 \mapsto \mathcal{R}_1; \ldots; f_n \mapsto \mathcal{R}_n\}]\!]^{\textbf{\textit{struct}}\{f_1:\tau_1,\ldots,f_n:\tau_n\}} = \{(\{f_1 = v_1; \ldots; f_n = v_n\}, \{f_1 = w_1; \ldots; f_n = w_n\}) \mid \forall i, 1 \le i \le n,\ (v_i, w_i) \in [\![\mathcal{R}_i]\!]^{\tau_i}\}$$

$$[\![[C_1 \mapsto \mathcal{R}_1; \ldots; C_n \mapsto \mathcal{R}_n]]\!]^{\textbf{\textit{variant}}[C_1:\tau_1 \mid \ldots \mid C_n:\tau_n]} = \{(C_i[v_i], C_i[w_i]) \mid \forall i, 1 \le i \le n,\ (v_i, w_i) \in [\![\mathcal{R}_i]\!]^{\tau_i}\}$$

$$[\![\langle \mathcal{R}_{def}\rangle]\!]^{\textbf{\textit{arr}}^{\tau_i}\langle\tau\rangle} = \{((\mathcal{P},(v)_k),(\mathcal{P},(w)_k)) \mid \forall k, (v_k, w_k) \in [\![\mathcal{R}_{def}]\!]^\tau\}$$

$$[\![\langle \mathcal{R}_{def} \triangleright i \ :\ \mathcal{R}_{exc}\rangle]\!]^{\textbf{\textit{arr}}^{\tau_i}\langle\tau\rangle} = \{((\mathcal{P},(v)_k),(\mathcal{P},(w)_k)) \mid E(i) \in \mathcal{P} \Longrightarrow (v_{E(i)}, w_{E(i)}) \in [\![\mathcal{R}_{exn}]\!]^\tau,\ \forall k \ne E(i), (v_k, w_k) \in [\![\mathcal{R}_{def}]\!]^\tau\}$$

The preorder relation of the partial equivalence lattice is denoted by $\sqsubseteq_{\mathscr{R}}$. It is defined in Table 4.

$$\sqsubseteq_{\mathscr{R}} \subseteq \mathscr{R} \times \mathscr{R} \qquad \vee_{\mathscr{R}}\colon \mathscr{R} \times \mathscr{R} \to \mathscr{R} \qquad \wedge_{\mathscr{R}}\colon \mathscr{R} \times \mathscr{R} \to \mathscr{R}.$$

The defined *join* and *meet* operations, denoted by $\vee_{\mathscr{R}}$ and $\wedge_{\mathscr{R}}$, are *commutative* operations, applied pointwise on each subelement. *Join* has Equal as its identity element and Any as its absorbing element. *Meet* has Equal as its absorbing element and Any as its identity element.

Additionally, the following extraction functions are defined:

$$\begin{aligned}\textit{extr}_f &: \mathscr{R} \nrightarrow \mathscr{R} \text{ extraction of a field's relation}\\ \textit{extr}_C &: \mathscr{R} \nrightarrow \mathscr{R} \text{ extraction of a constructor's relation}\\ \textit{extr}_{\langle i\rangle} &: \mathscr{R} \nrightarrow \mathscr{R} \text{ extraction of a cell's relation.}\end{aligned}$$

These are partial functions and can only be applied on relations of the corresponding types. For example, the field extraction $\textit{extr}_f$ only makes sense for

Table 4. $\sqsubseteq_{\mathscr{R}}$ – Comparison of two domains

$$\frac{}{\mathcal{R} \sqsubseteq_{\mathscr{R}} \mathsf{Any}}\ \textsc{Top} \qquad \frac{}{\mathsf{Equal} \sqsubseteq_{\mathscr{R}} \mathcal{R}}\ \textsc{Bot}$$

$$\frac{\mathcal{R}_1 \sqsubseteq_{\mathscr{R}} \mathcal{R}'_1 \quad \dots \quad \mathcal{R}_n \sqsubseteq_{\mathscr{R}} \mathcal{R}'_n}{\{f_1 \mapsto \mathcal{R}_1; \dots; f_n \mapsto \mathcal{R}_n\} \sqsubseteq_{\mathscr{R}} \{f_1 \mapsto \mathcal{R}'_1; \dots; f_n \mapsto \mathcal{R}'_n\}}\ \textsc{Str}$$

$$\frac{\mathcal{R}_1 \sqsubseteq_{\mathscr{R}} \mathcal{R}'_1 \quad \dots \quad \mathcal{R}_n \sqsubseteq_{\mathscr{R}} \mathcal{R}'_n}{[C_1 \mapsto \mathcal{R}_1; \dots; C_n \mapsto \mathcal{R}_n] \sqsubseteq_{\mathscr{R}} [C_1 \mapsto \mathcal{R}'_1; \dots; C_n \mapsto \mathcal{R}'_n]}\ \textsc{Var}$$

$$\frac{\mathcal{R} \sqsubseteq_{\mathscr{R}} \mathcal{R}'}{\langle \mathcal{R} \rangle \sqsubseteq_{\mathscr{R}} \langle \mathcal{R}' \rangle}\ \textsc{Adef} \qquad \frac{\mathcal{R}_{def} \sqsubseteq_{\mathscr{R}} \mathcal{R}'_{def} \qquad \mathcal{R}_{exc} \sqsubseteq_{\mathscr{R}} \mathcal{R}'_{exc}}{\langle \mathcal{R}_{def} \triangleright i \ : \ \mathcal{R}_{exc} \rangle \sqsubseteq_{\mathscr{R}} \left\langle \mathcal{R}'_{def} \triangleright i : \ \mathcal{R}'_{exc} \right\rangle}\ \text{AI}$$

$$\frac{\mathcal{R}_{def} \sqsubseteq_{\mathscr{R}} \mathcal{R}' \qquad \mathcal{R}_{exc} \sqsubseteq_{\mathscr{R}} \mathcal{R}'}{\langle \mathcal{R}_{def} \triangleright i \ : \ \mathcal{R}_{exc} \rangle \sqsubseteq_{\mathscr{R}} \langle \mathcal{R}' \rangle}\ \text{AIA} \qquad \frac{\mathcal{R} \sqsubseteq_{\mathscr{R}} \mathcal{R}'_{def} \qquad \mathcal{R} \sqsubseteq_{\mathscr{R}} \mathcal{R}'_{exc}}{\langle \mathcal{R} \rangle \sqsubseteq_{\mathscr{R}} \left\langle \mathcal{R}'_{def} \triangleright i : \ \mathcal{R}'_{exc} \right\rangle}\ \text{AAI}$$

$$\frac{\mathcal{R}_{def} \sqsubseteq_{\mathscr{R}} \mathcal{R}'_{def} \qquad \mathcal{R}_{def} \sqsubseteq_{\mathscr{R}} \mathcal{R}'_{exc} \qquad \mathcal{R}_{exc} \sqsubseteq_{\mathscr{R}} \mathcal{R}'_{def} \qquad \mathcal{R}_{exc} \sqsubseteq_{\mathscr{R}} \mathcal{R}'_{exc}}{\langle \mathcal{R}_{def} \triangleright i \ : \ \mathcal{R}_{exc} \rangle \sqsubseteq_{\mathscr{R}} \left\langle \mathcal{R}'_{def} \triangleright j : \ \mathcal{R}'_{exc} \right\rangle}\ i \neq j\ \text{AIJ}$$

atomic or structured relations having a field named f, which should be the case if the relation connects two values of a structured type with a field f. For any of the two atomic relations $\mathcal{R}_a$, applying any of these extractions yields $\mathcal{R}_a$.

3.2 Paths and Correlations

Partial equivalence relations are enough to represent fine-grained information for values of the same structured type. For the example introduced in Sect. 1.1 and detailed in Fig. 2 these would suffice to express the equality of the `pid`, `current_thread` and `address_space` fields between the input process `in` and the output process `o`, by simply mapping this pair to {`threads` $\mapsto$ Any; `pid` $\mapsto$ Equal; `current_thread` $\mapsto$ Equal; `address_space` $\mapsto$ Equal}. However, the partial equivalence relations cannot, for example, be used to convey the equality at line 1 in Fig. 2 between the value of the `threads` field of `in` and the local `ta` variable. In order to express this information, we first need to be able to refer to the substructure `in.threads` and relate its value to the one of `ta`.

Rather than handling only partial equivalences between pairs of variables of the same type and approximating the rest to Any – the element that conveys no information – we introduce an intermediate level, allowing us to store relations between subparts of values. To this end, we begin by introducing *paths*.

A *path* is rooted at one of the program's variables and represents a unique sequence of internal accesses inside some value's structure, i.e. it is a traversal from one value to one of its subparts. Each path is a unique chain of accesses leading to a nested element. We define a recursive type Π encompassing this:

$$\pi \in \Pi,\ \pi := \begin{array}{|ll} \varepsilon & \text{empty} - \text{root} \\ .f\pi & f \in \mathscr{F} \\ @C\pi & C \in \mathscr{C} \\ \langle i \rangle \pi & i \text{ index, program variable.} \end{array}$$

The *empty* path, denoted by ε, is the special case denoting an access to an entire element, i.e. the root. The action of appending a *non-empty* path π' to another path π is denoted by $\pi :: \pi'$.

Meaningful information is conveyed by associating paths and partial equivalence relations. For example, the equality between `in.threads` and `ta` at line 1 in Fig. 2 can be expressed by associating Equal to the pair of subelements identified by the `.threads` path in `in` and by ε in `ta`. Thus, we introduce *correlation maps* $\hat{c} \in \hat{\mathcal{C}}$, which are finite mappings from pairs of paths to relations $\mathcal{R} \in \mathscr{R}$:

$$\hat{\mathcal{C}} : \Pi \times \Pi \ \rightarrow \mathscr{R}$$

Generally, for two given variables e and o, a correlation $(\pi, \rho) \mapsto \mathcal{R}$ specifies that e and o have nested subelements, respectively identified by the inner paths π and ρ, whose values are related by the relation $\mathcal{R}$.

There is no clear canonical form for correlations. For instance, it is equivalent to write $(\varepsilon, \varepsilon) \mapsto \{f \mapsto \mathcal{R}\}$ and $(.f, .f) \mapsto \mathcal{R}$. Operations can create and manipulate them in different manners, that are hard to predict. New correlations can also be introduced while considering *def-use* chains in the transfer function presented later in Sect. 3.3. This trait renders the definition of a partial order between correlation maps difficult. In order to compare two correlation maps $\hat{c}_1$ and $\hat{c}_2$, we cannot simply verify if the path pairs are identical and compare their associated relations. A correlation of the second map could be *linked*, in different manners, to multiple mappings of the first. For example, between a process `p` of the type defined in Sect. 1.1 and an array `ta` of the same type as the field `threads` of the process, we might have the following correlation maps:

$$\hat{c}_1 : \begin{array}{r} (\texttt{.threads}, \varepsilon) \mapsto \langle \mathsf{Equal} \triangleright \texttt{i} : \ \mathsf{Any} \rangle \\ (\texttt{.threads}\langle\texttt{i}\rangle\texttt{@Some.t}, \langle\texttt{i}\rangle\texttt{@Some.t}) \mapsto \left\{ \begin{array}{r} \texttt{identifier} \mapsto \mathsf{Equal} \\ \texttt{current\_state} \mapsto \mathsf{Any} \\ \texttt{stack} \mapsto \mathsf{Equal} \end{array} \right\} \end{array}$$

$$\hat{c}_2 : \ (\texttt{.threads}, \varepsilon) \mapsto \left\langle \left[\begin{array}{l} \texttt{None} \mapsto \mathsf{Any} \\ \texttt{Some} \mapsto \left\{ \texttt{t} \mapsto \left\{ \begin{array}{r} \texttt{identifier} \mapsto \mathsf{Equal} \\ \texttt{current\_state} \mapsto \mathsf{Any} \\ \texttt{stack} \mapsto \mathsf{Equal} \end{array} \right\} \right\} \end{array} \right] \right\rangle .$$

To compare two correlation maps $\hat{c}_1$ and $\hat{c}_2$, we need to collect for each pair (π, ρ) mapped to $\mathcal{R}$ in $\hat{c}_2$ all the information contained by $\hat{c}_1$ that refers to the elements identified by (π, ρ) and verify if this covers at least the same information as the relation $\mathcal{R}$. This information could be scattered across multiple mappings of the correlation map $\hat{c}_1$. For example, in the given map $\hat{c}_1$, in addition to the relation associated to $(\texttt{.threads}, \varepsilon)$, the relation associated

to $(.\texttt{threads}\langle \texttt{i} \rangle @\texttt{Some.t}, \langle \texttt{i} \rangle @\texttt{Some.t})$ expresses information about the values of the process' `threads` field and `ta` as well. These are nested in the `i`-th element of each, as identified by $\langle \texttt{i} \rangle @\texttt{Some.t}$. To compare these two correlation maps, we have to first determine the relationships between the pair of paths $(.\texttt{threads}, \varepsilon)$ from $\hat{c}_2$ and each pair of paths of $\hat{c}_1$. The first pair of paths in $\hat{c}_1$ is identical, whereas the second pair refers to elements that are further away from the root. Based on these relationships, we have to extract all the information relevant to $(.\texttt{threads}, \varepsilon)$ from $\hat{c}_1$. This amounts to:

$$(.\texttt{threads}, \varepsilon) \mapsto \left\langle \mathsf{Equal} \triangleright \texttt{i} : \left[\begin{array}{l} \texttt{None} \mapsto \mathsf{Any} \\ \texttt{Some} \mapsto \left\{ \texttt{t} \mapsto \left\{ \begin{array}{r} \texttt{identifier} \mapsto \mathsf{Equal} \\ \texttt{current\_state} \mapsto \mathsf{Any} \\ \texttt{stack} \mapsto \mathsf{Equal} \end{array} \right\} \right\} \end{array} \right] \right\rangle .$$

which is more precise than the relation associated to $(.\texttt{threads}, \varepsilon)$ in $\hat{c}_2$. We call this process *alignment*. It is necessary in the absence of a canonical form, a trait of our approach that is both a weakness and a strength: it leads to complex computations but gives considerable flexibility.

For aligning, we first determine the relationships between paths by determining the relationship between the sequences of internal accesses that they represent. These can be identical, representing the same traversal to the same subelement of a value or they can be completely unrelated, such as $.f$ and $.g$ for example, representing accesses to two different fields of a structure. They can also represent sequences of accesses of different depths, one being the prefix of the other, i.e. being closer to the root. For example, the path $.f$ is a prefix of the path $.f\langle i \rangle$; the first represents the access to the field f, whereas the second one represents an access to the i-th element of the array nested in the field f.

To distinguish between these cases, we have defined a *link* type, $\mu \in \mathscr{M}$:

$$\mu := \mid \mathsf{Identical} \mid \mathsf{Left}\ \pi \mid \mathsf{Right}\ \pi \mid \mathsf{Incompatible}$$

and a *matching* operator $\curlywedge$:

$$\curlywedge\ :\ \Pi \times \Pi\ \to \mathscr{M} \qquad \curlywedge\,(\pi, \rho) = \begin{cases} \mathsf{Identical}, & \pi = \rho \\ \mathsf{Left}\ \pi', & \pi :: \pi' = \rho \\ \mathsf{Right}\ \rho', & \rho :: \rho' = \pi \\ \mathsf{Incompatible}, & \text{otherwise} \end{cases}$$

that retrieves the link between two paths. *Aligning* a correlation $(\pi, \rho) \mapsto \mathcal{R}$ to another pair of paths (π', ρ'), is denoted by $\|$.

$$\| : \hat{\mathcal{C}} \times (\Pi \times \Pi) \to \mathscr{R} \quad [(\pi, \rho) \mapsto \mathcal{R}] \parallel (\pi', \rho') = \mathcal{R}^{(\pi,\rho)}_{\|(\pi',\rho')}.$$

From $\mathcal{R}$ we obtain the information referring to the elements identified by (π', ρ') and denote it by $\mathcal{R}^{(\pi,\rho)}_{\|(\pi',\rho')}$. This is done by *matching* on π and π' on the one hand and on ρ and ρ' on the other and by distinguishing between the different cases. When the paths are identical, we can simply return the relation $\mathcal{R}$.

When the links between the paths differ or when the paths are incompatible, we have to approximate to the least precise relation, thus returning Any. When π and ρ are more shallow paths, i.e. closer to the root, we need to make a *projection*, denoted by $\rightsquigarrow$. For example, aligning $(.f, \varepsilon) \mapsto \{a \mapsto \mathcal{R}_a; b \mapsto \mathcal{R}_b; c \mapsto \mathcal{R}_c\}$ to $(.f.b, .b)$ consists in projecting $.b$ on the relation $\{a \mapsto \mathcal{R}_a; b \mapsto \mathcal{R}_b; c \mapsto \mathcal{R}_c\}$ and thus obtaining $\mathcal{R}_b$. On the contrary, if π' and ρ' are closer to the root, we need to perform an *injection*, denoted by $\curvearrowleft$. For example, aligning $(.f.b, .b) \mapsto \mathcal{R}_b$ to $(.f, \varepsilon)$ consists in creating a relation $\{a \mapsto \mathsf{Any}; b \mapsto \mathcal{R}_b; c \mapsto \mathsf{Any}\}$.

$$\mathcal{R}^{(\pi,\rho)}_{\parallel(\pi',\rho')} = \begin{cases} \mathcal{R} & \text{when } \curlywedge\,(\pi,\pi') = \curlywedge(\rho,\rho') = \mathsf{Identical} \\ \rightsquigarrow(\sigma,\mathcal{R}) & \text{when } \curlywedge\,(\pi,\pi') = \curlywedge(\rho,\rho') = \mathsf{Left}\ \sigma \\ \curvearrowleft(\mathcal{R},\sigma) & \text{when } \curlywedge\,(\pi,\pi') = \curlywedge(\rho,\rho') = \mathsf{Right}\ \sigma \\ \mathsf{Any} & \text{otherwise} \end{cases}$$

$$\text{where} : \rightsquigarrow : \Pi \times \mathscr{R} \nrightarrow \mathscr{R} \quad \curvearrowleft : \mathscr{R} \times \Pi \nrightarrow \mathscr{R}$$

$$\rightsquigarrow(\pi,\mathcal{R}) = \begin{cases} \mathcal{R} & \text{when } \pi = \varepsilon \\ \rightsquigarrow(\pi', \mathit{extr}_f(\mathcal{R})), & \text{when } \pi = .f\pi' \\ \rightsquigarrow(\pi', \mathit{extr}_C(\mathcal{R})), & \text{when } \pi = @C\pi' \\ \rightsquigarrow(\pi', \mathit{extr}_{\langle i\rangle}(\mathcal{R})), & \text{when } \pi = \langle i\rangle\pi' \end{cases}$$

$$\curvearrowleft(\mathcal{R},\pi) = \begin{cases} \mathcal{R} & \text{when} \pi = \varepsilon \\ \{f_1 \mapsto \mathsf{Any}; \ldots; f_i \mapsto \curvearrowleft(\mathcal{R},\pi'); \ldots; f_n \mapsto \mathsf{Any}\}, & \text{when} \pi = .f\pi', f = f_i \\ [C_1 \mapsto \mathsf{Any}; \ldots; C_i \mapsto \curvearrowleft(\mathcal{R},\pi'); \ldots; C_n \mapsto \mathsf{Any}], & \text{when} \pi = @C\pi', C = C_i \\ \langle \mathsf{Any} \triangleright i : \ \curvearrowleft(\mathcal{R},\pi')\rangle, & \text{when} \pi = \langle i\rangle\pi' \end{cases}$$

Aligning a correlation map $\hat{c} \in \hat{\mathcal{C}}$ to (π', ρ'), amounts to performing this operation for each element $(\pi, \rho) \mapsto \mathcal{R}$ of $\hat{c}$ and intersecting the results with the $\wedge_{\mathscr{R}}$ operator:

$$\hat{c} \parallel (\pi', \rho') = \bigwedge_{(\pi,\rho)\mapsto\mathcal{R}\in\hat{c}}{}_{\mathscr{R}}\ \mathcal{R}^{(\pi,\rho)}_{\parallel(\pi',\rho')}.$$

Finally, we can define the preorder for correlation maps:

$$\hat{c}_1 \mathbin{\hat{\sqsubseteq}} \hat{c}_2 \iff \forall\,[(\pi,\rho) \mapsto \mathcal{R}] \in \hat{c}_2,\ \hat{c}_1 \parallel (\pi,\rho) \sqsubseteq_{\mathscr{R}} \mathcal{R}.$$

Any correlation map $\hat{c} \in \hat{\mathcal{C}}$ is smaller than $\emptyset$, the empty correlation map.
The defined join operation between two correlation maps is denoted by $\hat{\bigvee}$:

$$\hat{c}_1 \mathbin{\hat{\bigvee}} \hat{c}_2 = \hat{c}_3 \iff \forall\,[(\pi,\rho) \mapsto \mathcal{R}] \in \hat{c}_1, \hat{c}_3(\pi,\rho) = \mathcal{R} \vee_{\mathscr{R}} (\hat{c}_2 \parallel (\pi,\rho))\,.$$

The *meet* operation between two correlation maps is denoted by $\hat{\bigwedge}$:

$$\hat{c}_1 \mathbin{\hat{\bigwedge}} \hat{c}_2 = \hat{c}_3 \iff \hat{c}_3(\pi,\rho) = \hat{c}_1(\pi,\rho) \wedge_{\mathscr{R}} \hat{c}_2(\pi,\rho),\ \forall(\pi,\rho).$$

3.3 Intraprocedural Analysis and Correlation Summaries

We work with a control flow graph (CFG) representation of the predicates' bodies. Nodes represent program states and edges are defined by statements with a particular exit label λ. In our case, all the outgoing edges of a node n

bear the different cases of the same statement s found at the program point n. For each statement s there is an edge labeled s, λ_k for each of its possible exit labels λ_k. However, the analysis does not depend on this specificity.

Correlation information has to be kept at each point of the CFG, for each input and output pair of the node. An *intraprocedural* correlation summary:

$$\Delta \in \mathscr{D}, \quad \Delta : \mathcal{V} \times \mathcal{V} \to \hat{\mathcal{C}}.$$

is thus a mapping from pairs of variables $v \in \mathcal{V}$ to correlation maps.

For each node of a given control flow graph, $\Delta(e, o)$ retrieves the correlation map between the local variable e and the output variable o. If a mapping for e and o does not currently exist, $\Delta(e, o)$ retrieves the correlation $(\varepsilon, \varepsilon) \mapsto \mathsf{Equal}$ when $e = o$ or the empty correlation map $\emptyset$, otherwise. Establishing the partial order $\sqsubseteq$ and the join operation $\bigvee : \mathscr{D} \times \mathscr{D} \to \mathscr{D}$ is straightforward: $\hat{\sqsubseteq}$ and $\hat{\bigvee}$ are extended pointwise to an intraprocedural summary, for each ordered input-output pair and its associated correlation map.

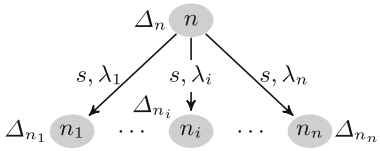
$$\sqsubseteq \subseteq \mathscr{D} \times \mathscr{D} \qquad \Delta_1 \sqsubseteq \Delta_2 \iff \forall e, o \in \mathcal{V},\ \Delta_1(e, o)\ \hat{\sqsubseteq}\ \Delta_2(e, o)$$

$$\Delta_1 \bigvee \Delta_2 = \Delta_3 \iff \forall (e, o),\ \Delta_3(e, o) = \Delta_1(e, o)\ \hat{\bigvee}\ \Delta_2(e, o)$$

Our correlation analysis is a *backward* data-flow analysis, computing an intraprocedural summary at each point of the control flow graph. This represents the correlations at the node's *entry point*. For each exit label, it traverses the control flow graph starting with its corresponding exit node. The intraprocedural summary for the currently analyzed label is initialized with pairs between the local value of each associated output variable of the label and the final value of the same output variable, mapped to $(\varepsilon, \varepsilon) \mapsto \mathsf{Equal}$. The analysis traverses the control flow graph and gradually refines the correlations, using Kildall's worklist algorithm [5], until a fixed point is reached. Table 5 summarizes the representation and general equation of the statements. For each statement, the presented data-flow equation operates on the intraprocedural summaries of the statement's *successor* nodes. The intraprocedural summary at the *entry point* of the node is obtained by *joining* the contributions of each *outgoing* edge. The contribution of an edge (n, n_i) labeled with s and λ_i is given by $\mathbb{C}^s_{\lambda_i}(\Delta_{n_i}) \in \mathbb{C}$ where $\mathbb{C}^s_{\lambda_i}(.)$ is the *transfer function* of the edge labeled s, λ_i.

The transfer function $\mathbb{C}^s_\lambda(.)$ formalizes the correlations created by the statement s on the label λ between its local input variables and its local output variables, denoted by δ^s_λ, as well as the set $kill_\lambda$ of variables whose values have been redefined by the statement s on the label λ. These are shown in Table 5. There is one crucial difference between transfer functions $\mathbb{C}^s_\lambda(.)$ and intraprocedural summaries Δ. An intraprocedural summary Δ implicitly maps any pair (v, v) for $v \in \mathcal{V}$ to $(\varepsilon, \varepsilon) \mapsto \mathsf{Equal}$. On the contrary, in δ^s_λ, when the variable v is used as both input and output by the statement s, the pair (v, v) is mapped to the correlation known between the input's v old value and the output's v fresh value. Otherwise, when v is an output, i.e. $v \in kill_\lambda$, but not an input of s, (v, v) is mapped to $\emptyset$.

Table 5. Statements – representations and data-flow equations

Representation	Equation
	$\Delta_n = \bigvee_{n \xrightarrow{s,\lambda_i} n_i} \mathbb{C}^s_{\lambda_i}(\Delta_{n_i})$

Statement	$\mathbb{C}^s_\lambda(.)$: δ^s_λ	$kill_\lambda$
Assignment $o := e$	$\{(e, o) \mapsto [(\varepsilon, \varepsilon) \mapsto \mathsf{Equal}]\}$	$\{o\}_{true}$
Get Field $o := r.f_i$	$\{(r, o) \mapsto [(.f_i, \varepsilon) \mapsto \mathsf{Equal}]\}$	$\{o\}_{true}$
Set Field $r' := \{r\ \textbf{\textit{with}}\ f_i = e\}$	$\{(r, r') \mapsto [(\varepsilon, \varepsilon) \mapsto \{f_1 \mapsto \mathsf{Equal}; \ldots; f_i \mapsto \mathsf{Any}; \ldots; f_n \mapsto \mathsf{Equal}\}]$ $(e, r') \mapsto [(\varepsilon, .f_i) \mapsto \mathsf{Equal}]\}$	$\{r'\}_{true}$
Create Var. $v := C_p[e]$	$\{(e, v) \mapsto [(\varepsilon, @C_p.e) \mapsto \mathsf{Equal}]\}$	$\{v\}_{true}$
Var. Switch $\textbf{\textit{switch}}(v)\ \textbf{\textit{as}}\ [o_1 \mid \ldots \mid o_n]$	$\{(v, o_i) \mapsto [(@C_i.e, \varepsilon) \mapsto \mathsf{Equal}]\}$	$\{o_i\}_{\lambda_{C_i}}$
Array Get $o := a[i]$	$\{(a, o) \mapsto [(\langle i \rangle, \varepsilon) \mapsto \mathsf{Equal}]\}$	$\{o\}_{true}$
Array Set $a' := [a\ \textbf{\textit{with}}\ i = e]$	$\{(a, a') \mapsto [(\varepsilon, \varepsilon) \mapsto \langle \mathsf{Equal} \triangleright i\ :\ \mathsf{Any} \rangle]$ $(e, a') \mapsto [(\varepsilon, \langle i \rangle) \mapsto \mathsf{Equal}]\}$	$\{a'\}_{true}$

In order to obtain the contribution $\mathbb{C}^s_{\lambda_i}(\Delta_{n_i})$ of an edge labeled with s and λ_i, we need to connect the information given by the $\delta^s_{\lambda_i}$ to the information contained in the intraprocedural summary Δ_{n_i}. For example, at the entry of node 3 in Fig. 2, when considering the scenario in which the predicate exits with `true`, the intraprocedural summary contains the mapping:

$$(\texttt{th}, \texttt{o}) \mapsto \left[(@\texttt{Some.t}, \texttt{.threads}\langle \texttt{i} \rangle @\texttt{Some.t}) \mapsto \left\{ \begin{array}{r} \texttt{identifier} \mapsto \mathsf{Equal} \\ \texttt{current\_state} \mapsto \mathsf{Any} \\ \texttt{stack} \mapsto \mathsf{Equal} \end{array} \right\} \right].$$

On the `true` edge statement 2 creates the mapping: $(\texttt{ta}, \texttt{th}) \mapsto [(\langle \texttt{i} \rangle, \varepsilon) \mapsto \mathsf{Equal}]$. Intuitively, since we are traversing the graph backwards and mapping ordered (local) input-output pairs, (`ta`, `th`) and (`th`, `o`) can be seen as a *def-use* pair: the correlation associated to (`ta`, `th`) expresses the relation between the *defined* value of `th` and the input `ta` used for creating it, while the correlation associated to (`th`, `o`) shows a subsequent *use* of that value of `th` for creating `o`. The contribution of statement 2 on the `true` edge should capture this flow of `ta`'s value to `o`'s value, through the variable `th`. Thus, it should contain a mapping for the pair (`ta`, `o`). In the general case we need to detect any variable r such that $[(p, r) \mapsto \hat{c}] \in \delta^s_{\lambda_i}$, $[(r, q) \mapsto \hat{c}'] \in \Delta_{n_i}$ and compute the mapping for (p, q) in $\mathbb{C}^s_{\lambda_i}(\Delta_{n_i})$.

In order to compute the correlation map associated to (`ta`, `o`), we take into account the fact that both the right path ε of $\delta^s_\lambda(\texttt{ta}, \texttt{th})$ and the left path `@Some.t` of $\Delta_{n_3}(\texttt{th}, \texttt{o})$ refer to the `th` variable. However, they do not represent traversals of the same depth: ε refers to the entire value of `th`, while `@Some.t` refers to the value below the constructor `Some`. Between `ta` and `o` we can conclude that the values nested under the `Some` constructor of the `i`-th elements are related:

$$(\texttt{ta},\texttt{o}) \mapsto \left[\langle\texttt{i}\rangle\texttt{@Some.t}, \texttt{.threads}\langle\texttt{i}\rangle\texttt{@Some.t}) \mapsto \left\{\begin{array}{r}\texttt{identifier} \mapsto \mathsf{Equal}\\ \texttt{current\_state} \mapsto \mathsf{Any}\\ \texttt{stack} \mapsto \mathsf{Equal}\end{array}\right\}\right].$$

We call the process of obtaining the correlation map associated to (ta, o) from the correlations associated to (ta, th) and (th, o) *composition* and denote it by $\odot$. In the general case, we obtain the link between ρ and π' by *matching* with $\curlywedge$. In the context of the example given above, ρ and π' are the paths referring to the th variable, i.e. ε and @Some.t, respectively. If these paths are compatible, we compose the correlation elements $(\pi,\rho) \mapsto \mathcal{R}$ and $(\pi',\rho') \mapsto \mathcal{R}'$, obtaining a new correlation element, $(\pi_\bullet,\rho_\bullet) \mapsto \mathcal{R}_{\bowtie}$, computed as follows:

$$(\pi_\bullet,\rho_\bullet) = (\pi,\rho) \bullet (\pi',\rho') \stackrel{def}{=} \left\{\begin{array}{ll}(\pi,\rho') & \text{when } \curlywedge\,(\rho,\pi') = \mathsf{Identical}\\ (\pi :: \sigma,\rho') & \text{when } \curlywedge\,(\rho,\pi') = \mathsf{Left}\ \sigma\\ (\pi,\rho' :: \sigma) & \text{when } \curlywedge\,(\rho,\pi') = \mathsf{Right}\ \sigma\end{array}\right.$$

$$\mathcal{R}_{\bowtie} = \mathcal{R} \bowtie \mathcal{R}' \stackrel{def}{=} \left\{\begin{array}{ll}\mathcal{R} \vee_{\mathscr{R}} \mathcal{R}' & \text{when } \curlywedge\,(\rho,\pi') = \mathsf{Identical}\\ \rightsquigarrow(\sigma,\mathcal{R}) \vee_{\mathscr{R}} \mathcal{R}' & \text{when } \curlywedge\,(\rho,\pi') = \mathsf{Left}\ \sigma\\ \mathcal{R} \vee_{\mathscr{R}} \rightsquigarrow(\sigma,\mathcal{R}') & \text{when } \curlywedge\,(\rho,\pi') = \mathsf{Right}\ \sigma\end{array}\right.$$

Note that given the special form of partial relations $\mathcal{R} \in \mathscr{R}$, the compose operation at this level is equivalent to $\vee_{\mathscr{R}}$. However, this would not be the case anymore for a more complex partial relation type.

The composition of correlation maps is denoted by $\bigcirc$. Computing $\hat{c}_1 \bigcirc \hat{c}_2$ amounts to intersecting the composition of all correlation elements from $\hat{c}_1$ and $\hat{c}_2$:

$$(\hat{c}_1 \bigcirc \hat{c}_2)(\pi_\bullet,\rho_\bullet) = \bigwedge_{\substack{\mathcal{R}\\ (\pi,\rho)\mapsto\mathcal{R}\in\hat{c}_1\\ (\pi',\rho')\mapsto\mathcal{R}'\in\hat{c}_2\\ (\pi_\bullet,\rho_\bullet)=(\pi,\rho)\bullet(\pi',\rho')}} \mathcal{R} \bowtie \mathcal{R}'.$$

Finally, the contribution $\mathbb{C}^s_{\lambda_i}(\Delta_{n_i})$ is obtained by:

$$\odot : \mathbb{C} \times \mathscr{D} \to \mathscr{D} \quad \delta^s_\lambda \odot \Delta = \Delta' \quad \text{where } \Delta'(p,q) = \hat{\bigwedge_r}(\delta^s_\lambda(p,r) \bigcirc \Delta(r,q)).$$

Interprocedural Level. Our analysis is performed label by label and interprocedural correlation domains associate an intraprocedural summary to each exit label of the analyzed predicate. Therefore, interprocedural domains encapsulate an intraprocedural summary for each possible execution scenario of a predicate.

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

$$\Xi_p : \Lambda_p \to \Delta \quad \text{where } \Lambda_p \text{ is the set of output labels of predicate } p.$$

The intraprocedural summary associated to each label is *filtered* so as to contain only ordered pairs of variables where the left member is an input of the analyzed predicate and the right member is an output associated to the analyzed label. The correlation maps associated to such pairs are built so as to contain correlations where only input variables may appear in array cell paths. Similarly, the

exception index in partial equivalence relations of arrays must be an input variable. Registering exceptions in array correlations only for input variables is not a consequence of a language restriction on array operations, but simply a consequence of the fact that at the interprocedural level, only correlation information between inputs and outputs makes sense.

The interprocedural domain of a predicate is used for deducing the transfer functions for a predicate call statement.

In the following we detail 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:

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

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

$$\mathbb{C}_{\lambda_i}^{s}(\Delta_{n_i}) = \delta_{\lambda_i}^{s} \odot \Delta_{n_i},\ kill_{\lambda_i} = \{\bar{o}_i\}$$
$$\delta_{\lambda_i}^{s}(e_j, o_i^k) = \hat{c}_i^{j,k}, \forall j \in \{1, \ldots, n\}, \forall k \in \{1, \ldots, h\}$$

where

$$\hat{c}_i^{j,k} = \Xi_p(\lambda_i)(\epsilon_j, \omega_i^k) \blacktriangleleft (\bar{\epsilon} \mapsto \bar{e})$$
$$s = p(e_1, \ldots, e_n)\ [\lambda_1 : \bar{o}_1 \mid \ldots \mid \lambda_m : \bar{o}_m]; \quad \bar{o}_i = \{o_i^1, \ldots, o_i^h\}.$$

Namely, the contribution of a predicate call to each (e_j, o_i^k) input-output pair stems from the contribution of the interprocedural domain for label λ_i and formal input-output pair (ϵ_j, ω_i^k). In these, all the formal input parameters $\bar{\epsilon}$ in array partial equivalences and in array cell paths are substituted by the corresponding effective input parameters from $\bar{e}$ or approximated away.

The substitution operation is denoted by $\blacktriangleleft\ (\sigma)$ where σ is a substitution from formal to effective parameters.

4 Preliminary Results and Experiments

Our analysis has currently been applied to a functional specification of *ProvenCore* [7], a general-purpose microkernel inspired by Minix 3.1 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.

Some of the abstract layers of *ProvenCore* are the *Refined Security Model* (RSM), the *Functional Specification* (FSP) and the *Target of Evaluation Design* (TDS). RSM is an abstract layer located just below the top-most layer of the refinement chain; the FSP is a model closely resembling the most concrete layer – 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 such as *fork, exec, exit*. 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 limited subset of the input state. For example, in FSP there are 25 possible transitions. Its state contains 15 fields; it is characterized by 70 invariants. In the TDS these figures are doubled. Each invariant is concerned with a different subset of the global's state fields. Some of these invariants concern all the processes held in the process store. Processes are complex structures in their own right, having more than 20 fields themselves. However, most transitions affect only a few of these fields.

We have applied our analysis on the RSM, FSP and TDS layers. These are medium-sized experiments. An overview of their characteristics and the time needed to obtain the correlation results are given in Table 6. The first column shows the total number of predicates of the analysed layers. In parentheses, we indicate the number of predicates that only read information, i.e. logical properties, as well as the number of opaque predicates for which a pessimistic assumption is made. The second column shows the total number of lines of code (LoC) for each. The next two columns indicate the number of LoC corresponding to type definitions and comments, respectively. The average time needed to compute the correlation and dependency results are shown in the last two columns. Unlike the correlation analysis that only computes information for predicates that actually modify data structures, the dependency analysis computes information for code as well as specifications, i.e. logical properties, in a unified manner. This explains the time difference between the two analyses.

Table 6. Abstract layers - evaluation data and analysis timing

	Predicates	Total LoC	Types	Comments	Correlation	Dependency
RSM/FSP	633 (235/65)	9853	596	855	0.90 s	1.84 s
TDS	418 (58/105)	6804	460	623	0.62 s	1.09 s

One of the analyzed predicates is `do_auth`. It is a system call clearing or granting an authorization to some process to read from or write to some memory range of the current process. It receives a global state `in` and an index `i` as inputs and produces, on the `true` label, the new global state `out`, after modifying the permission for the `i`-th process in the process store. The code of `do_auth` performs various system-wide checks before registering the permission change, and is therefore not trivial, although its effect is quite limited. Indeed, the correlation

results computed by our analysis for the `true` label of this predicate are shown below. The analysis detects that out of the 15 fields of `out`, only the `i`-th element of the `procs` field is changed. Furthermore, it detects that if this element is an active process, only the `mem_auth` field is modified out of the total of 26 fields. Everything else is copied from the input state `in`.

```
true : (in, out) ↦ [

                    (ε, ε) ↦ { ...      ↦ Equal } 14 fields
                               procs    ↦ Any }
(.procs, .procs) ↦ ⟨ Equal ▷ i :  [None  ↦ Equal
                                   Some  ↦ {v ↦ { ...       ↦ Equal } 25 fields
                                                mem_auth ↦ Any}}]⟩]
```

Combined with dependency results for logical properties, these results would allow us to infer the preservation of all invariants that are not concerned with the memory permissions. All but one out of the 70 properties fall into this category. This is the *relevant memory permissions* property, which states that a process has permissions covering a valid range of memory addresses and referring only to existing processes. It has to hold for every process in the process store. After executing `do_auth`, this property is threatened and needs to be verified only for the `i`-th process of the store. It is preserved for all others.

Space constraints prevent us from discussing more examples here. However, various other examples are provided and explained on the web page[2] dedicated to our analysis. Users can devise and test their own examples as well.

5 Related Work

In [3], Chang and Leino present the congruence-closure abstract domain, designed for an object-oriented context and implemented in the Spec# program verifier. They infer and express relations between fields of variables, a goal similar to ours. The congruence-closure domain maintains equivalence graphs mapping field accesses to symbolic locations. On its own, this domain allows the inference and expression of relations for accessed fields. In order to take into account updates as well, this needs to use the heap succession domain as a base. Unlike us, they can express preorders between fields, depending on the base domains used. However, our domain handles both accesses and updates to structures, arrays and variants in a uniform manner, independent of additional information.

Rakamarić and Hu report in [12] a method to infer frame axioms of procedures and loops based on static analysis. As a starting point, they use the DSA shape analysis, presented by Lattner et al. [6]. DSA provides a summary of points-to relations as a graph, that is used to compute a set of memory locations that are modified by a procedure or its callers. By a pass through the graph,

[2] http://ajl-demo.fr/2016.

for each node reachable from the globals or procedure parameters, they generate expressions representing a path to that node. The generated frame axioms are used internally by an extended static checker of C programs, i.e. in a purely automatic setting. In contrast, our analysis is designed for an interactive verification context. Our technique focusing on a purely functional language is not concerned by aliasing and does not depend on an external points-to framework.

In [15], Taghdiri et al. present a technique for extracting procedure summaries for object-oriented procedures, used to prove verification conditions. Procedures are executed symbolically and the environment of the post-state is computed so as to express every variable and field in terms of the values of the variables and fields of the pre-state. Their goal is broader than ours. However, unlike their summaries, our correlation results encompass only information that is visible from the outside (to the callers).

The literature on shape analysis [2,4,11,13] and side effects analyses [10,14] is vast. The former is aimed at deep-heap mutations, while we are focusing on deep-state modifications, in the context of complex transition systems. The latter determine memory locations that may be modified by an operation. Reasoning about heap locations is beyond our scope. We treat mappings between variables and their values, analyze their evolution in a side-effect free environment and detect not only what is modified, but also how and to what extent.

6 Conclusion and Future Work

Identifying precise information concerning the effects of program operations is possible by means of static analysis without sacrificing scalability. We have presented a flow-sensitive, interprocedural correlation analysis that has been applied to a functional specification of an operating system. The analysis tracks the origin of subparts of the output and relates it to subparts of the inputs thus detecting not only what is modified, but also how and to what extent. It is designed as a companion tool to be used during interactive program verification.

We have plans for future work along two main directions. The first is to go beyond the detection of equivalences and to handle *preorders*. This would allow us to detect the evolution of constructors for variants. Tracking this would allow the inference of properties that are not affected by a transition from a stronger state to a weaker state. Also, experiments show that the simultaneous use of dependency and correlation information can lead to a substantial reduction of proof obligations. Our priority is to employ the two, to develop a proof tactic for the inference of preserved invariants and to integrate it in our prover.

Acknowledgments. We would like to thank the anonymous referees for helpful comments and suggestions. For their excellent comments and sharp observations, we are particularly grateful to Olivier Delande and Georges Dupéron. Our article also benefited from the remarks of B. Montagu and H. Chataing.

References

1. Andreescu, O.F., Jensen, T., Lescuyer, S.: Dependency analysis of functional specifications with algebraic data structures. In: Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Proceedings, pp. 116–133 (2015). doi:10.1007/978-3-319-25423-4_8
2. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of Bi-abduction. In: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 289–300 (2009). http://doi.acm.org/10.1145/1480881.1480917
3. Chang, B.E., Leino, K.R.M.: Abstract interpretation with alien expressions and heap structures. In: Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Proceedings, pp. 147–163 (2005). http://dx.doi.org/10.1007/978-3-540-30579-8_11
4. Jones, N.D., Muchnick, S.S.: Flow analysis and optimization of lisp-like structures. In: Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, 1979, pp. 244–256 (1979). http://doi.acm.org/10.1145/567752.567776
5. Kildall, G.A.: A unified approach to global program optimization. In: Conference Record of the ACM Symposium on Principles of Programming Languages, 1973, pp. 194–206 (1973). http://doi.acm.org/10.1145/512927.512945
6. Lattner, C., Lenharth, A., Adve, V.S.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, 2007, pp. 278–289 (2007). http://doi.acm.org/10.1145/1250734.1250766
7. Lescuyer, S.: ProvenCore: towards a verified isolation micro-kernel (2015). http://milsworkshop2015.euromils.eu/downloads/hipeac_literature/04-mils15_submission_6.pdf
8. Mccarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. In: Machine Intelligence. Edinburgh University Press (1969)
9. Meyer, B.: Framing the frame problem. In: Dependable Software Systems Engineering, pp. 193–203 (2015). http://dx.doi.org/10.3233/978-1-61499-495-4-193
10. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Trans. Softw. Eng. Methodol. **14**(1), 1–41. http://doi.acm.org/10.1145/1044834.1044835 (2005)
11. Montenegro, M., Peña, R., Segura, C.: Shape analysis in a functional language by using regular languages. Sci. Comput. Program. **111**, 51–78 (2015). http://dx.doi.org/10.1016/j.scico.2014.12.006
12. Rakamaric, Z., Hu, A.J.: Automatic inference of frame axioms using static analysis. In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), pp. 89–98 (2008). http://dx.doi.org/10.1109/ASE.2008.19
13. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL 1999, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1999, pp. 105–118 (1999). http://doi.acm.org/10.1145/292540.292552
14. Sălcianu, A., Rinard, M.: Purity and side effect analysis for Java programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 199–215. Springer, Heidelberg (2005)
15. Taghdiri, M., Seater, R., Jackson, D.: Lightweight extraction of syntactic specifications. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, pp. 276–286 (2006). http://doi.acm.org/10.1145/1181775.1181809