Inferring Frame Conditions with Static Correlation Analysis

OANA F. ANDREESCU*, Internet of Trust, France
THOMAS JENSEN*, Inria, France and Prove & Run, France
STÉPHANE LESCUYER, Prove & Run, France
BENOÎT MONTAGU, Prove & Run, France

We introduce the abstract domain of *correlations* to denote equality relations between parts of inputs and outputs of programs. We formalise the theory of correlations, and mechanically verify their semantic properties. We design a static inter-procedural dataflow analysis for automatically inferring correlations for programs written in a first-order language equipped with algebraic data-types and arrays. The analysis, its precision and execution cost, have been evaluated on the code and functional specification of an industrial-size micro-kernel. We exploit the inferred correlations to automatically discharge two thirds of the proof obligations related to the preservation of invariants for this micro-kernel.

CCS Concepts: • **Theory of computation** → **Program analysis**; *Program verification*; • **Software and its engineering** → **Formal software verification**;

Additional Key Words and Phrases: Static analysis, Equality analysis, Function summaries, Frame conditions, Correlations, Invariant preservation

ACM Reference Format:
Oana F. Andreescu, Thomas Jensen, Stéphane Lescuyer, and Benoît Montagu. 2019. Inferring Frame Conditions with Static Correlation Analysis. *Proc. ACM Program. Lang.* 3, POPL, Article 47 (January 2019), 29 pages.
https://doi.org/10.1145/3290360

1 INTRODUCTION

In the context of the specification and verification of programs, a *frame condition* defines an upper bound of the effect performed by a program. In other words, it specifies which parts of the state a program *does not* modify. Frame conditions often occur in verification tools in the form of *modify clauses* [Filliâtre and Paskevich 2013; Leavens et al. 2006]. Stating and maintaining those conditions is referred to as the *frame problem* [Borgida et al. 1995; McCarthy and Hayes 1981; Meyer 2015].

While verifying a program, it is of prime importance to know when parts of an input state are copied to the output, because properties of that part of the input can be *directly* transferred to properties about the output. Stating and proving such equality relations between inputs and outputs, however, can be tedious and time-consuming. This is particularly pronounced in the context of the interactive verification of operating systems (OS). The state of an OS is indeed a large and complex data-structure, equipped with dozens of invariants, but most operations of the OS only affect a

*This work was developed while at Prove & Run.

Authors' addresses: Oana F. Andreescu, Internet of Trust, Paris, France, oana.andreescu@internetoftrust.com; Thomas Jensen, Inria, Rennes, France , Prove & Run, Paris, France, thomas.jensen@inria.fr; Stéphane Lescuyer, Prove & Run, Paris, France, stephane.lescuyer@provenrun.com; Benoît Montagu, Prove & Run, Paris, France, benoit.montagu@provenrun.com.

2475-1421/2019/1-ART47
https://doi.org/10.1145/3290360

small part of this global state. The goal of this paper is to statically *infer* such equalities, in order to *assist* the effort of interactive mechanised verification of large pieces of software.

We present a general-purpose static dataflow analysis [Nielson et al. 2010] for identifying equalities between parts of the input and output states, for programs written in a language used in the ProvenCore [Lescuyer 2015] development. ProvenCore is a project whose goal is the formal verification of a micro-kernel, starting from a high-level functional specification and refining it to an efficient implementation, that is eventually compiled to C. Starting from a set of atomic equality relations between basic values, we construct the abstract domain of *correlations* that describes relations between complex data-structures, including records, variants and arrays. For each construct in the language, we define a transfer function over correlations. This provides a compositional inter-procedural analysis for inferring input-output correlations in a modular fashion. We have used ProvenCore as an industrial-size testbed for our analysis.

Our correlation analysis is designed with the aim of finding a compromise between precision, scalability and trustworthiness. Our goal is to significantly *alleviate* the burden of interactive proof, by the use of a *provably correct* analyser. Therefore, our correlations only track *equalities* to keep the analyser simple. We leave more complex features such as arithmetic reasoning to the interaction with the proof engineer.

Correlations were previously introduced in [Andreescu et al. 2016], also for the purpose of over-approximating input-output relations. That previous work, however, included no formalisation, and treated variants and arrays in an unsound way. The present work shares the same goal, but features a domain of correlations that is original, radically different and *proved sound*. The present work also distinguishes itself by its implementation, that has been thoroughly tested and experimentally evaluated. Section 9 provides a detailed comparison with related approaches.

The contributions of this paper are the following:

- We introduce the general purpose abstract domain of *correlations* (§ 2) as abstractions of binary relations over values of algebraic data-types. To our knowledge, the domain of correlations is the first relational domain that supports variants. We formally define the semantics and algebraic operations (§ 3) on correlations, and prove their soundness. Additionally, we mechanised the soundness proofs in Coq [Inria 2017].
- We exploit correlations to define a static analysis that safely approximates the frames of functions written in an intermediate representation language (§ 4 and § 5) used in the code and specification of ProvenCore. A salient fact of the analysis is that it computes—by its very nature—summary relations of functions (§ 5) and hence is easily extended to a *modular* inter-procedural analysis [Cousot and Cousot 2002]. We have proved the soundness of the transfer functions that define the analysis. A worked example is given in § 6.
- We have added support for *functional arrays* (§ 7), and proved this extension correct on paper.
- We have implemented the analysis and executed it on the ProvenCore development (§ 8), a sizeable example of approximately 58000 lines of code. We have exploited our correlation analysis to automatically prove the preservation of invariants by the system calls of ProvenCore. The experiment confirmed that the analysis has the desired precision, as we were able to automatically prove 68% of all the preservation statements.
- Finally, we have thoroughly tested the implementation by randomly generating correlations and values, so as to test the algebraic properties and the soundness of the operations on correlations, and also to test the soundness of the transfer functions.

```
type state = // States
{ procs: procs;  // the table of processes
  sched: sched } // the scheduler state
type procs = // Process table
  array<int, option<proc>>
type proc = // Process descriptor
{ nr: int;          // process index
  regs: regs;       // registers
  exe_name: string; // name of executable
  ipc_status: ipc_status } // IPC status
```

```
type regs = // Registers
  { r0: int; r1: int; r2: int; r3: int }
type ipc_status = // Process IPC statuses
| Ready                // ready to run
| Sleeping             // sleeping process
| Sending (int dst)    // blocked sending
| Receiving (int src) // blocked receiving
type option<A> = | Some (A x) | None
type bool = | T | F
type sched // Scheduler state (implicit)
```

Fig. 1. Type definitions for the state of a minimalist OS.

2 CORRELATIONS

2.1 Examples of Correlations

Consider as an example the type definitions of Fig. 1, that describe the state of some minimalist OS. It comprises a table of processes—an array of optional process descriptors—where unused slots are set to None. Process descriptors are composed of an index, a bank of registers that are saved and restored at context switches, an executable name, and an inter-process communication (IPC) status, that describes whether the process is ready to run, or is sleeping, or is blocked sending a message to another process, or is blocked waiting to receive a message from another process.

System calls can be modelled as functions that transform states into new ones. Sleep is one system call. It makes the calling process sleep for a given amount of time and may only modify the IPC status of the caller process and the scheduling state, so its modification of the state is very local. Another more involved system call is Kill, that kills some process at index i. It removes the process descriptor with index i from the process table—by setting its cell to None—and removes i from the scheduler state—because it will not be running anymore. It then clears any reference to i from the whole state. In our simplified example, this amounts to replacing any IPC status of processes of the form Receiving(i) or Sending(i) with Ready and then writing some error number in their r0 register. Since this operation might make some blocked processes ready to run, they will need to be registered as runnable processes in the scheduler state. Overall, the Kill system call performs a fairly complex change of the state, but it keeps whole *slices* of the state *unmodified*. For example, process indices (*i.e.*, values of the nr field) are never modified by Kill, and registers other than r0 are never modified either. Additionally, Kill never turns a non-sleeping process into a sleeping one, it does not create new processes in the process table, and it only removes *one* process from the table, namely the process at index i. All these properties might prove tedious to formally verify manually. We will show that our correlation analysis indeed succeeds in inferring them automatically, along with other properties (§ 6 and § 7).

Our correlation analysis focuses on finding equality relations between input and output of a function. Additionally, it is able to determine how the cases of variants may change, such as the transition from Some to None for the process descriptor removed by the Kill example. To define abstract domains of correlations we start with three basic relations: the equality relation Eq_τ that relates equal values of a given type τ, $\top$ that relates any value to any other one, and the empty relation $\bot$, that does not relate any values at all. We shall then extend correlations to inductive data-types—structures and variants—and to arrays.

Consider as an example a function that takes as input a value r of type regs and returns an integer i, where $r.\mathsf{r1} = i$. Formally, the semantics of the function is described by the relation

$$C_{\text{regs}} = \left\{\begin{array}{lllll} \mathsf{r0} \rightarrow \{\mathsf{r0} \rightarrow \mathsf{Eq}_{\mathtt{int}}; & \mathsf{r1} \rightarrow \top; & \mathsf{r2} \rightarrow \top; & \mathsf{r3} \rightarrow \mathsf{Eq}_{\mathtt{int}} & \}^{\mathsf{R}} \\ \mathsf{r1} \rightarrow \{\mathsf{r0} \rightarrow \top; & \mathsf{r1} \rightarrow \mathsf{Eq}_{\mathtt{int}}; & \mathsf{r2} \rightarrow \top; & \mathsf{r3} \rightarrow \top & \}^{\mathsf{R}} \\ \mathsf{r2} \rightarrow \{\mathsf{r0} \rightarrow \top; & \mathsf{r1} \rightarrow \top; & \mathsf{r2} \rightarrow \mathsf{Eq}_{\mathtt{int}}; & \mathsf{r3} \rightarrow \top & \}^{\mathsf{R}} \\ \mathsf{r3} \rightarrow \{\mathsf{r0} \rightarrow \top; & \mathsf{r1} \rightarrow \top; & \mathsf{r2} \rightarrow \top; & \mathsf{r3} \rightarrow \mathsf{Eq}_{\mathtt{int}} & \}^{\mathsf{R}} \end{array}\right\}^{\mathsf{L}}$$

$$C_{\text{option}} = \left[\begin{array}{ll} \mathsf{None} \rightarrow [\mathsf{None} \rightarrow \top \mid \mathsf{Some} \rightarrow \bot &]^{\mathsf{R}} \\ \mathsf{Some} \rightarrow [\mathsf{None} \rightarrow \top \mid \mathsf{Some} \rightarrow \mathsf{Eq}_{\{\mathtt{x:int}\}} &]^{\mathsf{R}} \end{array}\right]^{\mathsf{L}}$$

Fig. 2. Some examples of correlations.

$\{(\{\mathsf{r0} = v_0; \mathsf{r1} = v_1; \mathsf{r2} = v_2; \mathsf{r3} = v_3\}, v') \mid v_1 = v'\}$. As a convention, we always consider relations where inputs are on the left, and outputs on the right. This relation is denoted by the *record correlation* $\{\mathsf{r0} \rightarrow \top; \mathsf{r1} \rightarrow \mathsf{Eq}_{\mathtt{int}}; \mathsf{r2} \rightarrow \top; \mathsf{r3} \rightarrow \top\}^{\mathsf{L}}$. This notation reads as follows. The superscript L—the *side* of the correlation—specifies that the record value is the left component in the relation. The equality $r.\mathsf{r1} = i$ is specified by the presence of $\mathsf{Eq}_{\mathtt{int}}$ that is associated to the r1 record field. The other fields are unconstrained, as indicated by the presence of ⊤.

Similarly, the record correlation $\{\mathsf{r0} \rightarrow \top; \mathsf{r1} \rightarrow \top; \mathsf{r2} \rightarrow \mathsf{Eq}_{\mathtt{int}}; \mathsf{r3} \rightarrow \mathsf{Eq}_{\mathtt{int}}\}^{\mathsf{R}}$ relates any integer i with any value r of type `regs` such that $i = r.\mathsf{r2}$ and $i = r.\mathsf{r3}$, as required by $\mathsf{Eq}_{\mathtt{int}}$ that is bound to the fields r2 and r3. The side R is used this time, to express that the record value is the right component of the relation, *i.e.*, the output.

To express relations between sub-structures, we can mix sides as in the correlation C_{regs} (Fig. 2). This correlation relates two values r and r' of types `regs` such that $r.\mathsf{r}_i = r'.\mathsf{r}_i$ for every $0 \leq i \leq 3$—thanks to the presence of $\mathsf{Eq}_{\mathtt{int}}$ on the diagonal of the matrix—and such that $r.\mathsf{r}_0 = r'.\mathsf{r}_3$—as required by $\mathsf{Eq}_{\mathtt{int}}$ in the top right corner. The typesetting as a matrix of this correlation is made on purpose. We shall later see that there is indeed an analogy between correlations and matrices.

Finally, consider a correlation between two values of types `option⟨int⟩`. The *variant correlation* C_{option} (Fig. 2) relates None with None, and Some(x) with None, and Some(y) with Some(y). The pair (None, Some(z)) is outside the correlation; this is stated by ⊥ in the upper right corner. Moreover, when (Some(y_1), Some(y_2)) is in the correlation, it must necessarily be the case that $y_1 = y_2$, as indicated by $\mathsf{Eq}_{\{\mathtt{x:int}\}}$ in the lower right corner. To summarise, the above correlation can be interpreted as a *transition*, where options are either left unchanged, or are replaced with None.

In general, correlations may relate values of *different* types, and might be neither reflexive, symmetric, nor transitive. In the rest of this section we define the types and values of our language (§ 2.2). We then give the syntax and semantics of (well-typed) correlations (§ 2.3), and define a pre-order on correlations that respects the standard order on relations based on set inclusion (§ 2.4).

2.2 Types and Values

In the rest of the document, we assume the existence of countable sets of field names, constructors, and value variables. Field names (f, g, etc.) are names by which record fields are accessed. Constructor names (A, B, etc.) are names of head constructors of variant values. Value variables (`x`, `y`, `z`, `i`, `j`, etc.) are names used in programs to refer to values. We also assume the existence of a special variable ⋆, named *ghost variable*, that is supposed to never occur in programs. The ghost variable will be used at the intra-procedural level of the analysis (§ 5) in order to deal with the evolution of variant constructors. We write I to denote a finite set of integers.

Types and values are defined in Fig. 3. We assume pre-existing sets of basic types and basic values, on which we build compound types and compound values. Record types are maps from field names to types, whereas variant types are maps from constructor names to types.

$$\begin{array}{lcll} \tau & ::= & \cdots \mid \{\overline{\mathsf{f}_i : \tau_i}^{i \in I}\} \mid [\overline{\mathsf{A}_i : \tau_i}^{i \in I}] & \text{(Types)} \\ v & ::= & \cdots \mid \{\overline{\mathsf{f}_i = v_i}^{i \in I}\} \mid \mathsf{A}(v) & \text{(Values)} \end{array}$$

$$\textsc{Record}\quad \frac{\forall i \in I,\quad \vdash v_i : \tau_i \qquad \forall i, j \in I, i \neq j \Rightarrow \mathsf{f}_i \neq \mathsf{f}_j}{\vdash \{\overline{\mathsf{f}_i = v_i}^{i \in I}\} : \{\overline{\mathsf{f}_i : \tau_i}^{i \in I}\}}$$

$$\textsc{Variant}\quad \frac{k \in I \qquad \vdash v : \tau_k \qquad \forall i, j \in I, i \neq j \Rightarrow \mathsf{A}_i \neq \mathsf{A}_j}{\vdash \mathsf{A}_k(v) : [\overline{\mathsf{A}_i : \tau_i}^{i \in I}]}$$

Fig. 3. Types, values and well-typed values. Typing rules for basic values are omitted.

In the example of Fig. 1, the `option` type does not directly fit in the grammar definition of types, since the grammar expects that variant constructors have exactly one argument. In the concrete syntax, constructors with no arguments are considered as expecting an empty record as argument, and constructors with at least one argument are considered as expecting a record of the arguments. Therefore, the abstract syntax for the concrete definition of the `option` type is $\mathtt{option}\langle\alpha\rangle = [\mathsf{None} : \{\} \mid \mathsf{Some} : \{\mathsf{x} : \alpha\}]$.

Values and their typing judgments are defined in Fig. 3. Basic values are extended with record values and variant values. Record values are maps from field names to values, whereas variant values are pairs of a constructor name and a value. Similarly, we assume typing rules for basic values, which we extend with standard structural typing rules for records and variants.

2.3 Well-Typed Correlations and Their Matrix Representation

The purpose of our correlation analysis is to identify parts of input to a computation that appear unaltered in the output. The analysis works by computing a relation between input and output values, specifying which parts are equal. To this end, we shall define *correlations* to be a collection of syntactic objects that denote relations on values.

Definition 2.1 (Correlations). Correlations are inductively defined as follows:

$$C \quad ::= \quad \top \mid \mathsf{Eq}_\tau \mid \bot \mid \{\overline{\mathsf{f}_i \to C_i}^{i \in I}\}^{\mathcal{S}} \mid [\overline{\mathsf{A}_i \to C_i}^{i \in I}]^{\mathcal{S}}$$

There are three basic correlations: the full correlation $\top$ that relates any value to any other one, the equality correlation Eq_τ that relates equal values of a given type τ, and the empty relation $\bot$, that does not relate any value at all. On top of the basic correlations, we then build up compound correlations over algebraic data-types.

We first explain the correlations for records. The notation $\{\overline{\mathsf{f}_i \to C_i}^{i \in I}\}^{\mathcal{S}}$ is used to denote a relation between values $(v_\mathsf{L}, v_\mathsf{R})$ where one of the values must be a record and where the other value is related to the values of the *fields* of the record, as described by each C_i. It is the *side* superscript (L or R) on the correlation that determines whether it is the first (v_L) or the second (v_R) of the related values that is constrained to be a record. The sides are necessary in order to be able to define correlations between structures of different types.

The correlations for variants can be understood similarly: we use the notation $[\overline{\mathsf{A}_i \to C_i}^{i \in I}]^{\mathcal{S}}$ to denote a relation between a variant and another value, satisfying the condition that if the variant value has constructor tag $\mathsf{A_i}$ then the tagged value is related to the other value as described by the correlation C_i. Again, the side indicates which of the two values has to be a variant.

$$\textsc{Any}\ \frac{}{\vdash \top : \tau \times \tau'} \qquad \textsc{Eq}\ \frac{}{\vdash \mathsf{Eq}_\tau : \tau \times \tau} \qquad \textsc{Impossible}\ \frac{}{\vdash \bot : \tau \times \tau'}$$

$$\textsc{RecordL}\ \frac{\forall i \in I, \vdash C_i : \tau_i \times \tau'}{\vdash \{\overline{\mathsf{f}_i \to C_i}^{i\in I}\}^{\mathsf{L}} : \{\overline{\mathsf{f}_i : \tau_i}^{i\in I}\} \times \tau'} \qquad \textsc{RecordR}\ \frac{\forall i \in I, \vdash C_i : \tau \times \tau'_i}{\vdash \{\overline{\mathsf{f}_i \to C_i}^{i\in I}\}^{\mathsf{R}} : \tau \times \{\overline{\mathsf{f}_i : \tau'_i}^{i\in I}\}}$$

$$\textsc{VariantL}\ \frac{\forall i \in I, \vdash C_i : \tau_i \times \tau'}{\vdash [\overline{\mathsf{A}_i \to C_i}^{i\in I}]^{\mathsf{L}} : [\overline{\mathsf{A}_i : \tau_i}^{i\in I}] \times \tau'} \qquad \textsc{VariantR}\ \frac{\forall i \in I, \vdash C_i : \tau \times \tau'_i}{\vdash [\overline{\mathsf{A}_i \to C_i}^{i\in I}]^{\mathsf{R}} : \tau \times [\overline{\mathsf{A}_i : \tau'_i}^{i\in I}]}$$

Fig. 4. Well-typed correlations.

$$\begin{array}{rcl}
\llbracket \top \rrbracket^{\tau_1\times\tau_2} & = & \{(v_1, v_2) \mid \vdash v_1 : \tau_1 \text{ and } \vdash v_2 : \tau_2\} \\
\llbracket \mathsf{Eq}_\tau \rrbracket^{\tau\times\tau} & = & \{(v, v) \mid \vdash v : \tau\} \\
\llbracket \bot \rrbracket^{\tau_1\times\tau_2} & = & \emptyset \\
\left\llbracket \{\overline{\mathsf{f}_i \to C_i}^{i\in I}\}^{\mathsf{L}} \right\rrbracket^{\{\overline{\mathsf{f}_i:\tau'_i}^{i\in I}\}\times\tau} & = & \left\{(\{\overline{\mathsf{f}_i = v'_i}^{i\in I}\}, v) \mid \forall i \in I, (v'_i, v) \in \llbracket C_i \rrbracket^{\tau'_i\times\tau} \wedge \vdash v : \tau\right\} \\
\left\llbracket \{\overline{\mathsf{f}_i \to C_i}^{i\in I}\}^{\mathsf{R}} \right\rrbracket^{\tau\times\{\overline{\mathsf{f}_i:\tau'_i}^{i\in I}\}} & = & \left\{(v, \{\overline{\mathsf{f}_i = v'_i}^{i\in I}\}) \mid \vdash v : \tau \wedge \forall i \in I, (v, v'_i) \in \llbracket C_i \rrbracket^{\tau\times\tau'_i}\right\} \\
\left\llbracket [\overline{\mathsf{A}_i \to C_i}^{i\in I}]^{\mathsf{L}} \right\rrbracket^{[\overline{\mathsf{A}_i:\tau'_i}^{i\in I}]\times\tau} & = & \bigcup_{i\in I} \left\{(\mathsf{A}_i(v'_i), v) \mid (v'_i, v) \in \llbracket C_i \rrbracket^{\tau'_i\times\tau}\right\} \\
\left\llbracket [\overline{\mathsf{A}_i \to C_i}^{i\in I}]^{\mathsf{R}} \right\rrbracket^{\tau\times[\overline{\mathsf{A}_i:\tau'_i}^{i\in I}]} & = & \bigcup_{i\in I} \left\{(v, \mathsf{A}_i(v'_i)) \mid (v, v'_i) \in \llbracket C_i \rrbracket^{\tau\times\tau'_i}\right\}
\end{array}$$

Fig. 5. Semantics of correlations.

A correlation can be given a type $\tau_1 \times \tau_2$ that describes the types of the values that it relates (Fig. 4). The correlations $\top$ and $\bot$ relate values of arbitrary types, whereas the equality correlation Eq only relates values of the same types. The extension to records and variants is straightforward, and uses the side exponent to determine which of the involved values should have a type that matches the type imposed by the correlation.

Formally, the *semantics* of correlations is then defined as a type-indexed translation $\llbracket\ \ \rrbracket^{\tau_1\times\tau_2}$ from correlations to binary relations between values of types τ_1 and τ_2 (Fig. 5). The semantics is defined so that the value inhabitants have the types that are specified by the indices given to the translation function.

Lemma 2.2. *If* $(v_1, v_2) \in \llbracket C \rrbracket^{\tau_1\times\tau_2}$*, then* $\vdash v_1 : \tau_1$ *and* $\vdash v_2 : \tau_2$.

The semantic translation is a partial function. It is well-defined for well-typed correlations only.

Lemma 2.3. $\llbracket C \rrbracket^{\tau_1\times\tau_2}$ *is well-defined iff* $\vdash C : \tau_1 \times \tau_2$.

The semantics collapses correlations of the empty record type to a two-point lattice, because of the semantic equality $\llbracket \mathsf{Eq}_{\{\}} \rrbracket^{\{\}\times\{\}} = \llbracket \top \rrbracket^{\{\}\times\{\}} = \{(\{\}, \{\})\}$. Similarly, the semantics for the empty variant type collapses to a single point, because $\llbracket \mathsf{Eq}_{[]} \rrbracket^{[]\times[]} = \llbracket \top \rrbracket^{[]\times[]} = \llbracket \bot \rrbracket^{[]\times[]} = \emptyset$.

It is instructive to write correlations using a matrix notation. In this matrix view of correlations, the indices of the matrix are the *paths* for accessing the values being related and the entries are

$$p \quad ::= \quad \epsilon \mid .\mathsf{f}p \mid @\mathsf{A}p \quad \text{(Paths)}$$

$$\begin{array}{rcll}
v \Downarrow \epsilon & = & v & \\
\{\mathsf{f}_1 = v_1; \ldots; \mathsf{f}_n = v_n\} \Downarrow .\mathsf{f}_k p & = & v_k \Downarrow p & \text{when } 1 \leq k \leq n \\
\mathsf{A}(v) \Downarrow @\mathsf{B}p & = & v \Downarrow p & \text{when } \mathsf{A} = \mathsf{B}
\end{array}$$

$$\begin{array}{rcl}
\tau \Downarrow \epsilon & = & \tau \\
\{\mathsf{f}_1 : \tau_1; \ldots; \mathsf{f}_n : \tau_n\} \Downarrow .\mathsf{f}_k p & = & \tau_k \Downarrow p \text{ when } 1 \leq k \leq n \\
[\mathsf{A}_1 : \tau_1 \mid \ldots \mid \mathsf{A}_n : \tau_n] \Downarrow @\mathsf{A}_k p & = & \tau_k \Downarrow p \text{ when } 1 \leq k \leq n
\end{array}$$

Fig. 6. Paths (p), projections of values ($v \Downarrow p$) and projections of types ($\tau \Downarrow p$).

$$\begin{array}{rcl}
C \Downarrow^{\mathcal{S}} \epsilon & = & C \\
\top \Downarrow^{\mathcal{S}} p & = & \top \\
\bot \Downarrow^{\mathcal{S}} p & = & \bot \\
\{\mathsf{f}_1 \rightarrow C_1; \ldots; \mathsf{f}_n \rightarrow C_n\}^{\mathcal{S}} \Downarrow^{\mathcal{S}} .\mathsf{f}_k p & = & C_k \Downarrow^{\mathcal{S}} p \text{ if } 1 \leq k \leq n \\
\{\mathsf{f}_1 \rightarrow C_1; \ldots; \mathsf{f}_n \rightarrow C_n\}^{\mathcal{S}'} \Downarrow^{\mathcal{S}} p & = & \{\mathsf{f}_1 \rightarrow C_1 \Downarrow^{\mathcal{S}} p; \ldots; \mathsf{f}_n \rightarrow C_n \Downarrow^{\mathcal{S}} p\}^{\mathcal{S}'} \text{ if } \mathcal{S} \neq \mathcal{S}' \\
[\mathsf{A}_1 \rightarrow C_1 \mid \ldots \mid \mathsf{A}_n \rightarrow C_n]^{\mathcal{S}} \Downarrow^{\mathcal{S}} @\mathsf{A}_k p & = & C_k \Downarrow^{\mathcal{S}} p \text{ if } 1 \leq k \leq n \\
[\mathsf{A}_1 \rightarrow C_1 \mid \ldots \mid \mathsf{A}_n \rightarrow C_n]^{\mathcal{S}'} \Downarrow^{\mathcal{S}} p & = & [\mathsf{A}_1 \rightarrow C_1 \Downarrow^{\mathcal{S}} p \mid \ldots \mid \mathsf{A}_n \rightarrow C_n \Downarrow^{\mathcal{S}} p]^{\mathcal{S}'} \text{ if } \mathcal{S} \neq \mathcal{S}' \\
\mathsf{Eq}_{\{\mathsf{f}_1:\tau_1;\ldots;\mathsf{f}_n:\tau_n\}} \Downarrow^{\mathcal{S}} .\mathsf{f}_k p & = & \{\mathsf{f}_1 \rightarrow \top; \ldots; \mathsf{f}_k \rightarrow \mathsf{Eq}_{\tau_k} \Downarrow^{\mathcal{S}} p; \ldots; \mathsf{f}_n \rightarrow \top\}^{\mathcal{S}^{-1}} \text{ if } 1 \leq k \leq n \\
\mathsf{Eq}_{[\mathsf{A}_1:\tau_1 \mid \ldots \mid \mathsf{A}_n:\tau_n]} \Downarrow^{\mathcal{S}} @\mathsf{A}_k p & = & [\mathsf{A}_1 \rightarrow \bot \mid \ldots \mid \mathsf{A}_k \rightarrow \mathsf{Eq}_{\tau_k} \Downarrow^{\mathcal{S}} p \mid \ldots \mid \mathsf{A}_n \rightarrow \bot]^{\mathcal{S}^{-1}} \text{ if } 1 \leq k \leq n
\end{array}$$

Fig. 7. Projection of correlations ($C \Downarrow^{\mathcal{S}} p$).

basic correlations Eq, ⊤ or ⊥. For example, consider the following correlations, that relate values of the same types $\{\mathsf{f} : \tau; \mathsf{g} : \tau\}$:

$$C = \left\{ \begin{array}{l} \mathsf{f} \rightarrow \{\mathsf{f} \rightarrow \top; \quad \mathsf{g} \rightarrow \mathsf{Eq}_\tau\}^{\mathsf{R}} \\ \mathsf{g} \rightarrow \{\mathsf{f} \rightarrow \mathsf{Eq}_\tau; \ \mathsf{g} \rightarrow \top \quad\}^{\mathsf{R}} \end{array} \right\}^{\mathsf{L}} \qquad C' = \left\{ \begin{array}{l} \mathsf{f} \rightarrow \{\mathsf{f} \rightarrow \top; \quad \mathsf{g} \rightarrow \mathsf{Eq}_\tau\}^{\mathsf{L}} \\ \mathsf{g} \rightarrow \{\mathsf{f} \rightarrow \mathsf{Eq}_\tau; \ \mathsf{g} \rightarrow \top \quad\}^{\mathsf{L}} \end{array} \right\}^{\mathsf{R}}$$

C relates values v_1 and v_2 such that $v_1.\mathsf{f} = v_2.\mathsf{g}$ and $v_1.\mathsf{g} = v_2.\mathsf{f}$. The order in which the L and R sides are introduced does not matter, since the correlation C' is semantically equivalent to C. One can view those correlations as a matrix whose column indices are the paths of v_1, and whose line indices are the paths of v_2, and whose cells contain ⊤ or Eq_τ. We will see in § 3 that the matrix analogy is strong. Comparison, join and meet can indeed be seen as pointwise operations on the matrix, whereas composition is akin to matrix multiplication—with some interesting differences.

2.4 A Pre-Order on Correlations

The definition of a static correlation analysis requires a pre-order on correlations which expresses when one correlation is included in another (*i.e.*, is more *precise* in identifying identical parts of values). Our definition of a pre-order on correlations is based on the notion of *projection* which allows us to restrict a correlation to a sub-part of a data-structure. This sub-part is identified by an *access path*. Paths point deep inside values, types, or correlations. They are inductively defined (Fig. 6) as the empty path ϵ, or a projection on a field followed by a path $.\mathsf{f}p$, or a projection on a variant constructor followed by a path $@\mathsf{A}p$. The definitions of projections of values and types on a path are standard (Fig. 6).

We now define what it means to project a correlation on a path, on a given side (Fig. 7). It basically amounts to selecting the correct line (or column) in the matrix interpretation of the correlation. To make the definition more compact, we use the *flip* operation on sides, written $\mathcal{S}^{-1}$, that is defined such that $\mathsf{L}^{-1} = \mathsf{R}$ and $\mathsf{R}^{-1} = \mathsf{L}$. Projecting a correlation on the empty path is the identity, and projecting from either $\top$ or $\bot$ is also the identity. Projecting a record or variant correlation on its side selects the corresponding correlation and projects it further on the rest of the path. To project a record or variant correlation on its opposite side, it suffices to recursively project its sub-parts on the same path. Projecting an equality correlation amounts to η-expanding the equality correlation into a square matrix and then projecting using the other rules. The η-expansion of a record equality is a matrix with equalities on the diagonal and $\top$ everywhere else. For a variant equality, the η-expansion is a matrix with equalities on the diagonal and $\bot$ everywhere else, since the cases are pairwise incompatible.

Projection is a well-typed operation, which is formally stated by the next lemma.

Lemma 2.4. *Assume that* $\vdash C : \tau_1 \times \tau_2$.
(1) *If* $\tau_1 \Downarrow p = \tau_1'$, *then* $C \Downarrow^{\mathsf{L}} p = C'$ *is well-defined, and* $\vdash C' : \tau_1' \times \tau_2$.
(2) *If* $\tau_2 \Downarrow p = \tau_2'$, *then* $C \Downarrow^{\mathsf{R}} p = C'$ *is well-defined, and* $\vdash C' : \tau_1 \times \tau_2'$.

The next lemma relates the projection of a correlation to the projections of its inhabiting values.

Lemma 2.5. *If* $(v_1, v_2) \in [\![C]\!]^{\tau_1\times\tau_2}$, *then* $(v_1 \Downarrow p_1, v_2) \in [\![C \Downarrow^{\mathsf{L}} p_1]\!]^{(\tau_1\Downarrow p_1)\times\tau_2}$ *and* $(v_1, v_2 \Downarrow p_2) \in [\![C \Downarrow^{\mathsf{R}} p_2]\!]^{\tau_1\times(\tau_2\Downarrow p_2)}$ *when the projections are well-defined.*

Using projections of correlations, we can define what we mean by comparing correlations.

Definition 2.6 (Pre-order on correlations).

$$\textsc{Bot}\ \frac{}{\bot \sqsubseteq C} \qquad \textsc{Top}\ \frac{}{C \sqsubseteq \top} \qquad \textsc{Eq}\ \frac{}{\mathsf{Eq}_\tau \sqsubseteq \mathsf{Eq}_\tau} \qquad \textsc{Record}\ \frac{\forall i \in I,\ (C' \Downarrow^{\mathcal{S}} .\mathsf{f}_i) \sqsubseteq C_i}{C' \sqsubseteq \{\overline{\mathsf{f}_i \to C_i}^{i\in I}\}^{\mathcal{S}}} \qquad \textsc{Variant}\ \frac{\forall i \in I,\ (C' \Downarrow^{\mathcal{S}} @\mathsf{A}_i) \sqsubseteq C_i}{C' \sqsubseteq [\overline{\mathsf{A}_i \to C_i}^{i\in I}]^{\mathcal{S}}}$$

The pre-order $\sqsubseteq$ on correlations is defined so that $\bot$ is the smallest element in the pre-order, and $\top$ is a maximal element. Compound correlations are compared pointwise, using projections. So, for example, if we want to compare a correlation C' with a record-type correlation we project C' onto each field of the record, and compare that projection with the correlation specified for that field. Formally, this gives the inference rule Record. For variants, we reason as follows. A correlation C' is smaller than (*i.e.*, included in) a variant correlation, if when projecting C' onto any of the possible constructors for that variant, a correlation is obtained that is smaller than the one specified for that constructor. This gives the rule Variant.

The comparison relation is *not* anti-symmetric. For example, several correlations are equivalent to $\top$: record and variant correlation whose components are all set to $\top$ are equivalent to $\top$.

The relation $\sqsubseteq$ is a pre-order. While reflexivity is easily obtained, the proof of transitivity is more involved, because of the use of projections in the definition of $\sqsubseteq$.

Lemma 2.7 (Reflexivity). *For every correlation* C, *we have* $C \sqsubseteq C$.

Lemma 2.8 (Transitivity). *Assume that* $\vdash C_1 : \tau \times \tau'$ *and* $\vdash C_2 : \tau \times \tau'$ *and* $\vdash C_3 : \tau \times \tau'$. *If* $C_1 \sqsubseteq C_2$ *and* $C_2 \sqsubseteq C_3$, *then* $C_1 \sqsubseteq C_3$.

The proof of transitivity involves the following key lemma, stating that projections are monotonic.

Lemma 2.9 (Monotony of projection). *Assume that* $\vdash C_1 : \tau \times \tau'$ *and* $\vdash C_2 : \tau \times \tau'$. *Assume that* $C_1 \Downarrow^{\mathcal{S}} p = C_1'$ *and* $C_2 \Downarrow^{\mathcal{S}} p = C_2'$. *Assume that* $\tau \Downarrow p$ *is defined when* $\mathcal{S} = \mathsf{L}$, *and that* $\tau' \Downarrow p$ *is defined when* $\mathcal{S} = \mathsf{R}$. *If* $C_1 \sqsubseteq C_2$, *then* $C_1' \sqsubseteq C_2'$.

The pre-order is expected to respect the inclusion order on relations, *i.e.*, if one correlation is $\sqsubseteq$-smaller than another, then the semantics of the first is a subset of the latter. The following theorem states that this is indeed the case.

Lemma 2.10 (Semantic correctness of pre-order). *If $C_1 \sqsubseteq C_2$, then $[\![C_1]\!]^{\tau\times\tau'} \subseteq [\![C_2]\!]^{\tau\times\tau'}$.*

Notice that the converse implication does *not* hold. For example, the correlation $\{\mathsf{f} \to \top; \mathsf{g} \to \bot\}^{\mathsf{L}}$ is *semantically* equal to $\bot$, but is nonetheless strictly larger than $\bot$ for $\sqsubseteq$. The fact that the fields of record correlations are examined independently of each other is indeed a source of incompleteness.

On a different note, the pre-order does not include *extensional reasoning* on records or variants. For example, the equality correlation on a record type is *strictly* smaller than its η-expansion—*i.e.*, its square matrix representation. We found that supporting extensionality in the pre-order was the source of technical issues on the formal side and induced extra execution costs on the implementation side. We leave the support of extensionality to future work.

3 OPERATIONS ON CORRELATIONS

3.1 Unions and Intersections of Correlations

In this section, we define binary operators for approximating the union and the intersection of two correlations. They shall be used in the correlation analysis. We first define the union of two correlations which—with slight abuse of common conventions—we call the *join* of two correlations. The join, written $\sqcup$, will be an upper bound with respect to $\sqsubseteq$. In the presence of arrays or recursive types, the *least* upper bound may not exist. The join of correlations is specified in Fig. 8.

Joining two basic correlations is straightforward. Joining any correlation C with $\bot$ yields C, and joining a correlation with $\top$ yields $\top$. Joining two correlations where one of them is a record or variant correlation proceeds *pointwise*, *i.e.*, by joining each sub-component of the structure with the projection of the other correlation onto that sub-component. For example, we have the rule

$$\left\{\overline{\mathsf{f}_i \to C_i}^{\,i\in I}\right\}^{\mathsf{L}} \sqcup C' = \left\{\overline{\mathsf{f}_i \to C_i \sqcup \left(C' \Downarrow^{\mathsf{L}} .\mathsf{f}_i\right)}^{\,i\in I}\right\}^{\mathsf{L}}$$

Here, the projection $C' \Downarrow^{\mathsf{L}} .\mathsf{f}_i$ "unfolds" the correlation C' to obtain the relation concerning the sub-structure in the field f_i.

Interestingly, the problematic case is the join with the equality correlation which will require the introduction of another, more approximate version of the projection operator. The problem lies with the *termination* of the join, because projecting the equality creates a larger correlation. To join with Eq_τ, we expand the other correlation and perform the join pointwise on the sub-components. The termination is guaranteed when the sizes of types strictly decrease, but this criterion does not work for recursive types. To ensure termination even in the presence of recursive types, we define a *non-expansive* version of path projection, written $\downarrow$, which is an over-approximation of the projections defined in Fig. 7.

Definition 3.1 (Non-expansive projection of correlations (excerpts)).

$$C \downarrow^{\mathcal{S}} \epsilon = C \qquad \top \downarrow^{\mathcal{S}} p = \top \qquad \bot \downarrow^{\mathcal{S}} p = \bot \qquad \mathsf{Eq}_\tau \downarrow^{\mathcal{S}} .\mathsf{f}p = \top \qquad \mathsf{Eq}_\tau \downarrow^{\mathcal{S}} @\mathsf{A}p = \top$$

The remaining rules for records and for variants are the same as the ones for regular projection.

The non-expansive projection works like the ordinary projection, except on equality correlations. Instead of unfolding the equality relation and projecting it to an equality relation on sub-structures, the non-expansive version of projection just returns the trivial relation $\top$. The non-expansive projection turns a correlation into one whose *height* is smaller or equal. The join is then guaranteed to terminate because the heights of its arguments strictly decrease.

$$
\begin{array}{rcl}
\top \sqcup C & = & \top \\
\bot \sqcup C & = & C \\
\{\overline{\mathsf{f}_i \to C_i}^{i\in I}\}^{\mathcal{S}} \sqcup C' & = & \{\overline{\mathsf{f}_i \to C_i \sqcup (C' \Downarrow^{\mathcal{S}} .\mathsf{f}_i)}^{i\in I}\}^{\mathcal{S}} \\
[\overline{\mathsf{A}_i \to C_i}^{i\in I}]^{\mathcal{S}} \sqcup C' & = & [\overline{\mathsf{A}_i \to C_i \sqcup (C' \Downarrow^{\mathcal{S}} @\mathsf{A}_i)}^{i\in I}]^{\mathcal{S}} \\
\mathsf{Eq}_\tau \sqcup \mathsf{Eq}_\tau & = & \mathsf{Eq}_\tau \\
\mathsf{Eq}_{\{\overline{\mathsf{f}_i:\tau_i}^{i\in I}\}} \sqcup \{\overline{\mathsf{f}_i \to C_i}^{i\in I}\}^{\mathcal{S}} & = & \left\{\overline{\mathsf{f}_i \to \{\overline{\mathsf{f}_j \to C'_{ij}}^{j\in I}\}^{\mathcal{S}^{-1}}}^{i\in I}\right\}^{\mathcal{S}} \\
 & & \text{where } C'_{ij} = \begin{cases} \mathsf{Eq}_{\tau_i} \sqcup (C_i \downarrow^{\mathcal{S}^{-1}} .\mathsf{f}_i) & \text{when } i = j \\ \top & \text{when } i \neq j \end{cases} \\
\mathsf{Eq}_{[\overline{\mathsf{A}_i:\tau_i}^{i\in I}]} \sqcup [\overline{\mathsf{A}_i \to C_i}^{i\in I}]^{\mathcal{S}} & = & \left[\overline{\mathsf{A}_i \to [\overline{\mathsf{A}_j \to C'_{ij}}^{j\in I}]^{\mathcal{S}^{-1}}}^{i\in I}\right]^{\mathcal{S}} \\
 & & \text{where } C'_{ij} = \begin{cases} \mathsf{Eq}_{\tau_i} \sqcup (C_i \downarrow^{\mathcal{S}^{-1}} @\mathsf{A}_i) & \text{when } i = j \\ C_i \Downarrow^{\mathcal{S}^{-1}} @\mathsf{A}_j & \text{when } i \neq j \end{cases}
\end{array}
$$

Fig. 8. Join of correlations. The rules for the remaining cases are obtained by symmetry.

The next lemma states that the non-expansive projection is an over-approximation of the projection. It is used in the proof of Lemma 3.3.

Lemma 3.2. *Let* $C \Downarrow^{\mathcal{S}} p = C_1$ *and* $C \downarrow^{\mathcal{S}} p = C_2$. *Then,* $C_1 \sqsubseteq C_2$.

Lemma 3.3 (Join is an upper bound). *Assume that* $\vdash C_1 : \tau \times \tau'$ *and* $\vdash C_2 : \tau \times \tau'$. *Let* $C_{12} = C_1 \sqcup C_2$. *Then,* $C_1 \sqsubseteq C_{12}$ *and* $C_2 \sqsubseteq C_{12}$.

Then, the fact that join is a safe approximation of the union of relations follows from Lemma 2.10.

Lemma 3.4 (Semantic correctness of join). $[\![C_1]\!]^{\tau\times\tau'} \cup [\![C_2]\!]^{\tau\times\tau'} \subseteq [\![C_1 \sqcup C_2]\!]^{\tau\times\tau'}$.

Similar to the join operator, it is possible to define a "meet" operator $\sqcap$ that takes two correlations and computes an over-approximation of the *intersection*. The meet is defined in a way similar to how the join is defined: $\top$ is a neutral element and $\bot$ is absorbant. To compute the meet with Eq_τ, we use the non-expansive projection in the recursive call, to ensure termination, once again. Computing the meet of compound correlations amounts to recursively computing the meet in a pointwise manner. We leave out the formal definition of meet and just state its soundness result.

Lemma 3.5 (Semantic correctness of meet). $[\![C_1]\!]^{\tau\times\tau'} \cap [\![C_2]\!]^{\tau\times\tau'} \subseteq [\![C_1 \sqcap C_2]\!]^{\tau\times\tau'}$.

3.2 Sequential Composition of Correlations

An essential operation in the correlation analysis is the (sequential) *composition* of two correlations, written $C_1 \mathbin{⨟} C_2$ (Fig. 9). The intended semantics is that of relational composition. In particular, the equality correlation Eq_τ is the identity for $⨟$: composing any relation C with Eq_τ either to the right or to the left yields C. The empty correlation $\bot$ is absorbant, since anything composed with the empty relation yields the empty relation. The remaining rules are of two kinds: *contextual* rules, and *combination* rules. Contextual rules are of the form $C_1 \mathbin{⨟} C_2$, where C_1 is left-sided, or C_2 is right-sided. In such cases, the composition is defined recursively, in a pointwise manner. Combination rules are however of the form $C_1 \mathbin{⨟} C_2$, where C_1 is either right-sided or $\top$, and C_2 is either left-sided or $\top$. The two key rules are the combination rules for records and for variant. The

$$
\begin{array}{rcl}
\bot \fatsemi C & = & \bot \\
C \fatsemi \bot & = & \bot \\
\mathsf{Eq}_\tau \fatsemi C & = & C \\
C \fatsemi \mathsf{Eq}_\tau & = & C \\
\top \fatsemi \top & = & \top \\
\{\mathsf{f}_1 \to C_1; \dots; \mathsf{f}_n \to C_n\}^{\mathsf{L}} \fatsemi C' & = & \{\mathsf{f}_1 \to C_1 \fatsemi C'; \dots; \mathsf{f}_n \to C_n \fatsemi C'\}^{\mathsf{L}} \\
C' \fatsemi \{\mathsf{f}_1 \to C_1; \dots; \mathsf{f}_n \to C_n\}^{\mathsf{R}} & = & \{\mathsf{f}_1 \to C' \fatsemi C_1; \dots; \mathsf{f}_n \to C' \fatsemi C_n\}^{\mathsf{R}} \\
\{\mathsf{f}_1 \to C_1; \dots; \mathsf{f}_n \to C_n\}^{\mathsf{R}} \fatsemi \{\mathsf{f}_1 \to C'_1; \dots; \mathsf{f}_n \to C'_n\}^{\mathsf{L}} & = & \bigsqcap_{1 \le i \le n} C_i \fatsemi C'_i \\
\{\mathsf{f}_1 \to C_1; \dots; \mathsf{f}_n \to C_n\}^{\mathsf{R}} \fatsemi \top & = & \bigsqcap_{1 \le i \le n} C_i \fatsemi \top \\
\top \fatsemi \{\mathsf{f}_1 \to C_1; \dots; \mathsf{f}_n \to C_n\}^{\mathsf{L}} & = & \bigsqcap_{1 \le i \le n} \top \fatsemi C_i \\
[\mathsf{A}_1 \to C_1 \,|\, \dots \,|\, \mathsf{A}_n \to C_n]^{\mathsf{L}} \fatsemi C' & = & [\mathsf{A}_1 \to C_1 \fatsemi C' \,|\, \dots \,|\, \mathsf{A}_n \to C_n \fatsemi C']^{\mathsf{L}} \\
C' \fatsemi [\mathsf{A}_1 \to C_1 \,|\, \dots \,|\, \mathsf{A}_n \to C_n]^{\mathsf{R}} & = & [\mathsf{A}_1 \to C' \fatsemi C_1 \,|\, \dots \,|\, \mathsf{A}_n \to C' \fatsemi C_n]^{\mathsf{R}} \\
[\mathsf{A}_1 \to C_1 \,|\, \dots \,|\, \mathsf{A}_n \to C_n]^{\mathsf{R}} \fatsemi [\mathsf{A}_1 \to C'_1 \,|\, \dots \,|\, \mathsf{A}_n \to C'_n]^{\mathsf{L}} & = & \bigsqcup_{1 \le i \le n} C_i \fatsemi C'_i \\
\top \fatsemi [\mathsf{A}_1 \to C_1 \,|\, \dots \,|\, \mathsf{A}_n \to C_n]^{\mathsf{L}} & = & \bigsqcup_{1 \le i \le n} \top \fatsemi C_i \\
[\mathsf{A}_1 \to C_1 \,|\, \dots \,|\, \mathsf{A}_n \to C_n]^{\mathsf{R}} \fatsemi \top & = & \bigsqcup_{1 \le i \le n} C_i \fatsemi \top
\end{array}
$$

Fig. 9. Composition of correlations.

combination of records returns the *meet* of the pointwise compositions, *i.e.*, $\bigsqcap_{1 \le i \le n} C_i \fatsemi C'_i$. Indeed, the composition of the two record relations relates two values v_1 and v_3 that are transitively related by *every* projection on $.\mathsf{f}_i$ of some value v_2 by C_i on the left-hand side and by C'_i on the right-hand side. Therefore, v_1 and v_3 are related by every $C_i \fatsemi C'_i$ for every $1 \le i \le n$, thus they are related by their meet (Lemma 3.5). The combination of variants returns the *join* of the pointwise compositions, *i.e.*, $\bigsqcup_{1 \le i \le n} C_i \fatsemi C'_i$. Indeed, the composition of the two variant relations relates two values v_1 and v_3 that are transitively related by *some* projection on $@\mathsf{A}_i$ of some value v_2 by C_i on the left-hand side and by C'_i on the right-hand side. Therefore, v_1 and v_3 are related by some $C_i \fatsemi C'_i$ for some $i \in I$, thus they are related by their join (Lemma 3.4).

Interestingly, composing with the trivial (total) relation $\top$ might not necessarily give back $\top$: for example composing $\bot$ with $\top$ gives $\bot$. More generally, composing with $\top$ only forgets *one side* of the correlation: the remaining pieces of information are the possible variant cases of the other side. For instance, $\top \fatsemi \left[\mathsf{A} \to \bot \mid \mathsf{B} \to [\mathsf{C} \to \top \mid \mathsf{D} \to \bot]^{\mathsf{R}}\right]^{\mathsf{L}} = (\top \fatsemi \bot) \sqcup (\top \fatsemi [\mathsf{C} \to \top \mid \mathsf{D} \to \bot]^{\mathsf{R}}) = [\mathsf{C} \to \top \mid \mathsf{D} \to \bot]^{\mathsf{R}}$. Every information about the left-hand side values has disappeared. The fact that right-hand side values cannot be in the case D remains nevertheless present in the result of the composition.

Let us examine how composition works with homogeneous record correlations, *i.e.*, record correlations that relate values of the same types:

$$
\begin{array}{l}
\left\{ \begin{array}{l} \mathsf{f} \to \{\ \mathsf{f} \to C_{11};\quad \mathsf{g} \to C_{12}\ \}^{\mathsf{R}} \\ \mathsf{g} \to \{\ \mathsf{f} \to C_{21};\quad \mathsf{g} \to C_{22}\ \}^{\mathsf{R}} \end{array} \right\}^{\mathsf{L}} \fatsemi \left\{ \begin{array}{l} \mathsf{f} \to \{\ \mathsf{f} \to C'_{11};\quad \mathsf{g} \to C'_{12}\ \}^{\mathsf{R}} \\ \mathsf{g} \to \{\ \mathsf{f} \to C'_{21};\quad \mathsf{g} \to C'_{22}\ \}^{\mathsf{R}} \end{array} \right\}^{\mathsf{L}} \\
= \left\{ \begin{array}{l} \mathsf{f} \to \{\ \mathsf{f} \to (C_{11} \fatsemi C'_{11}) \sqcap (C_{12} \fatsemi C'_{21});\quad \mathsf{g} \to (C_{11} \fatsemi C'_{12}) \sqcap (C_{12} \fatsemi C'_{22})\ \}^{\mathsf{R}} \\ \mathsf{g} \to \{\ \mathsf{f} \to (C_{21} \fatsemi C'_{11}) \sqcap (C_{22} \fatsemi C'_{21});\quad \mathsf{g} \to (C_{21} \fatsemi C'_{12}) \sqcap (C_{22} \fatsemi C'_{22})\ \}^{\mathsf{R}} \end{array} \right\}^{\mathsf{L}}
\end{array}
$$

We observe that the composition acts as a matrix multiplication, where the scalar multiplication is the composition and the scalar addition is the *meet* operation.

$$\begin{array}{llll}
\mathit{inst} & ::= & \mathsf{nop}() & \text{(Nop)} \\
 & \mid & \mathsf{y} :=_\tau \mathsf{x} & \text{(Assignment)} \\
 & \mid & \mathsf{x}_1 =_\tau \mathsf{x}_2 & \text{(Equality test)} \\
 & \mid & \mathsf{y} :=_\tau \{\mathsf{f}_1 = \mathsf{x}_1\,;\,\ldots\,;\mathsf{f}_n = \mathsf{x}_n\} & \text{(Record creation)} \\
 & \mid & \mathsf{y} :=_\tau \mathsf{x.f} & \text{(Record access)} \\
 & \mid & \mathsf{y} :=_\tau \{\mathsf{x}\ \mathtt{with}\ \mathsf{f} = \mathsf{z}\} & \text{(Record update)} \\
 & \mid & \mathsf{y} :=_\tau \mathsf{A}(\mathsf{x}) & \text{(Variant creation)} \\
 & \mid & [\mathsf{A}_1\colon \mathsf{y}_1 \mid \ldots \mid \mathsf{A}_n\colon \mathsf{y}_n] :=_\tau \mathtt{switch}(\mathsf{x}) & \text{(Variant destruction)} \\
 & \mid & [\lambda^1\colon \mathsf{y}^1_1, \ldots \mathsf{y}^1_{n_1} \mid \ldots \mid \lambda^m\colon \mathsf{y}^m_1, \ldots \mathsf{y}^m_{n_m}] :=_\Gamma \mathtt{call}\, p(\mathsf{x}_1, \ldots, \mathsf{x}_n) & \text{(Call)}
\end{array}$$

Fig. 10. Instructions of the CFG representation.

Interestingly, composition behaves similarly, using join instead of meet, when it is called on variant domains:

$$\begin{bmatrix} \mathsf{A} \to [\ \mathsf{A} \to C_{11} \mid \ \mathsf{B} \to C_{12}\ ]^{\mathsf{R}} \\ \mathsf{B} \to [\ \mathsf{A} \to C_{21} \mid \ \mathsf{B} \to C_{22}\ ]^{\mathsf{R}} \end{bmatrix}^{\mathsf{L}} \fatsemi \begin{bmatrix} \mathsf{A} \to [\ \mathsf{A} \to C'_{11} \mid \ \mathsf{B} \to C'_{12}\ ]^{\mathsf{R}} \\ \mathsf{B} \to [\ \mathsf{A} \to C'_{21} \mid \ \mathsf{B} \to C'_{22}\ ]^{\mathsf{R}} \end{bmatrix}^{\mathsf{L}}$$
$$= \begin{bmatrix} \mathsf{A} \to [\ \mathsf{A} \to (C_{11} \fatsemi C'_{11}) \sqcup (C_{12} \fatsemi C'_{21}) \mid \ \mathsf{B} \to (C_{11} \fatsemi C'_{12}) \sqcup (C_{12} \fatsemi C'_{22})\ ]^{\mathsf{R}} \\ \mathsf{B} \to [\ \mathsf{A} \to (C_{21} \fatsemi C'_{11}) \sqcup (C_{22} \fatsemi C'_{21}) \mid \ \mathsf{B} \to (C_{21} \fatsemi C'_{12}) \sqcup (C_{22} \fatsemi C'_{22})\ ]^{\mathsf{R}} \end{bmatrix}^{\mathsf{L}}$$

We observe again that the composition acts as a matrix multiplication, where the scalar multiplication is the composition and the scalar addition is this time the *join* operation.

It is no surprise that the composition operator relates to matrix multiplication, and to paths computations in graphs. Adjacency matrices and their products are indeed used in algorithms to determine the existence of paths between vertices. Correlations can be understood as bipartite graphs whose two *parts* are the input nodes and the output nodes, and whose edges are equality relations. Then, the composition of correlations resembles the composition of *bipartite* graphs: the (bi-)adjacency matrix of the composition of two bipartite graphs is the product of their adjacency matrices. This simple analogy with graphs, however, does not account for the difference between variant and record correlations. We defer the detailed study of the relationship between correlations and graphs to future work.

The composition of correlations is sound, in the sense that it computes an over-approximation of the composition of relations. We write $R_1; R_2$ to denote the composition of the relations R_1 and R_2, defined as $\{(x, y) \mid \exists z, (x, z) \in R_1 \land (z, y) \in R_2\}$.

Lemma 3.6 (Semantic correctness of composition). $[\![C_1]\!]^{\tau\times\tau'} ; [\![C_2]\!]^{\tau'\times\tau''} \subseteq [\![C_1 \fatsemi C_2]\!]^{\tau\times\tau''}$.

We mechanised in Coq the proofs of the lemmas of § 2 and § 3, on a simpler version of correlations that features binary pairs and sums instead of records and variants. Additionally, we proved in Coq that all the functions we defined in those sections are *total* on well-typed inputs.

4 A LANGUAGE OF CONTROL FLOW GRAPHS

We develop a static correlation analysis for a control flow graph (CFG) representation of programs. This representation is a stripped down version of a language used in the ProvenCore development, that is close to a while language with *immutable* algebraic data-types, and *no heap*. There are no pointers, references, or global variables. In other words, all the variables are local to functions, *i.e.*, the functions are *pure*. We found this combination of features convenient for both expressing functional specifications, and supporting efficient code generation. We believe that the ideas conveyed by our correlation abstract domain and the devised static analysis are general enough to be useful for the analysis of other languages.

CFG nodes contain atomic instructions (Fig 10). Edges are labelled to identify the outgoing edges of a node (*e.g.*, an equality test has two exit labels, true and false). Functions can have several exit points, which are labelled too. Each instruction of a CFG is executed in an environment mapping variables to values. An instruction transforms a value environment into a new one, possibly adding or modifying bindings.

The no-operation nop() instruction transforms an environment into itself. The assignment $\mathsf{y} := \mathsf{x}$ instruction replaces the binding for y (or adds a new one if none is present) so that y now has the value of x in the environment. If no binding for x is found in the environment, the semantics of the instruction is undefined. The equality test $\mathsf{x}_1 = \mathsf{x}_2$ reads the values v_1 and v_2 of x_1 and x_2 in the environment, does not modify the environment, and returns with the exit label true when $v_1 = v_2$, or with the label false otherwise.

There are three instructions for creating, accessing and updating records, respectively. The record creation instruction $\mathsf{y} := \{\mathsf{f}_1 = \mathsf{x}_1\,;\ \dots\ ;\mathsf{f}_n = \mathsf{x}_n\}$ reads the values v_i for every x_i and assigns to y the record value $\{\mathsf{f}_1 = v_1; \dots ; \mathsf{f}_n = v_n\}$. The record access instruction $\mathsf{y} := \mathsf{x}.\mathsf{f}$ reads the value for x in the environment, which must be a record value with a field f, and assigns to y the value that is associated with the field f. The record update instruction $\mathsf{y} := \{\mathsf{x}\ \mathtt{with}\ \mathsf{f} = \mathsf{z}\}$ reads the value for x in the environment, which must be a record value with a field f, and assigns to y the record value for x where the field f has been updated to the value of z.

There are two instructions for creating and matching against variants. The instruction for variant creation $\mathsf{y} := \mathsf{A}(\mathsf{x})$ reads in the environment the value v for x and assigns to y the value $\mathsf{A}(v)$. The instruction for variant destruction $[\mathsf{A}_1\colon \mathsf{y}_1 \mid \dots \mid \mathsf{A}_n\colon \mathsf{y}_n] := \mathtt{switch}(\mathsf{x})$ reads the value of x in the environment—which must be of the form $\mathsf{A}_i(v)$ with $1 \leq i \leq n$—and assigns the value v to y_i and returns with the label λ_{A_i}.

Finally, the semantics of a function call is defined by building a fresh environment with the values of the function's inputs, and then evaluating the body of the function. This gives back an exit label and an output environment, in which all the formal outputs must be defined. The environment of the caller is then updated with the values that were computed for the outputs.

Fig. 11 depicts the CFGs of functions that we introduced in § 2, and which will be analysed in § 6. Entry points are framed in a double rectangle node; exit points are denoted by double circle nodes and are labelled with the output label. The output variables are underlined. The first graph of Fig. 11 defines the `set_r0` function. It takes as input a process p and an integer v, and returns a process as output. It has only one output label (true). The function sets the value of `p.regs.r0` in process p to the integer v. Since updates are functional, the execution of the program actually creates a *new* process `new_p` such that `new_p.regs.r0` equals v and such that p and `new_p` have equal values on all other projection paths. The second graph of Fig. 11 defines the `clear_proc_refs` function. It takes as input a process p and an integer i, and returns a process and a boolean as outputs. Again, it has only one output label (true). The function removes from p any "reference" to the process index i. Such a reference can occur in its field `p.ipc_status` if it is in the case `Sending` and `Receiving`. In such a case, the process is promoted to the Ready status, and `p.regs.r0` is set to some error code by a call to `set_r0`. Finally, `unblocked` is set to the boolean T. In all the other cases, the process is not modified, and `unblocked` is set to the boolean F.

The concrete examples make use of variants whose cases may have zero or several arguments—as was already the case for types of Fig. 1. For the sake of simplicity, we present instructions for variants that support exactly one argument, following the same choice we made about types in § 2. Adapting the theory to variants with zero or several arguments is straightforward.

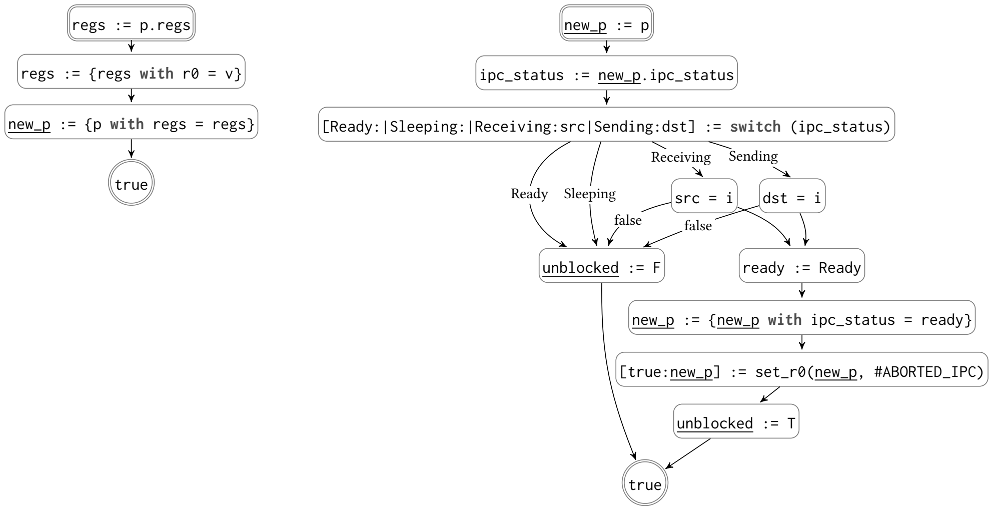
regs := p.regs
regs := {regs with r0 = v}
new_p := {p with regs = regs}
true
new_p := p
ipc_status := new_p.ipc_status
[Ready:|Sleeping:|Receiving:src|Sending:dst] := switch (ipc_status)
Receiving
Sending
Ready
Sleeping
src = i
dst = i
false
false
unblocked := F
ready := Ready
new_p := {new_p with ipc_status = ready}
[true:new_p] := set_r0(new_p, #ABORTED_IPC)
unblocked := T
true

Fig. 11. Left: the `set_r0` function, with signature `(proc p, int v) -> [true: proc new_p]`. Right: the `clear_proc_refs` function, with signature `(proc p, int i) -> [true: proc new_p, bool unblocked]`. Edges with no labels are implicitly labelled with `true`.

5 CORRELATION ANALYSIS

The intra-procedural correlation analysis is a *forward* dataflow analysis that computes for each program point correlations between the inputs to the procedure and the values at that program point, by the means of the iterative computation of a post-fixpoint [Nielson et al. 2010]. The abstract domain for program points consists of correlation maps, *i.e.*, maps from pairs of program variables to correlations. More formally, there are three kinds of intra-procedural abstract values:

- the Unreachable value denotes program points that could not be reached by the analysis;
- the Undef (undefined) value, denotes program points that result from undefined behaviour, such as an instruction that reads uninitialised variables;
- reachable values, are *correlation maps* between input variables and variables defined in the state of the current program point.

Intra-procedural abstract values are ranged over by $\mathcal{K}$. The existence of a binding $(\mathsf{x}, \mathsf{y}) \mapsto C$ between a variable x in the input state and a variable y in the current state denotes that x is defined in the input state and y is defined in the current state and their values are related by the semantics of C. We define a pre-order, written $\sqsubseteq$, on correlation maps such that Unreachable is the least element, Undef is the maximal element, and such that a reachable correlation map $\mathcal{K}_1$ is smaller than $\mathcal{K}_2$ when for every binding for the variables (x, y) in $\mathcal{K}_2$, there is also a binding in $\mathcal{K}_1$, and $\mathcal{K}_1(\mathsf{x}, \mathsf{y}) \sqsubseteq \mathcal{K}_2(\mathsf{x}, \mathsf{y})$. The join is the intersection of maps, using correlation join on the intersection of domains. The meet is the union of maps, using correlation meet on the intersection of domains.

The *transfer functions* of instructions are label specific. For an exit label λ, they have the particular form $\mathcal{K} \mapsto \mathcal{K} \fatsemi \mathcal{T}_\lambda$, where the *transformation* $\mathcal{T}_\lambda$ is a syntactic description of the correlations induced by the instruction when it exits with the label λ, and where $\fatsemi$ is a composition operation. Thus, $\mathcal{T}_\lambda$ can be understood as a *transformation* of correlations. It describes how a finite number of correlations are modified, and also specifies that the remaining correlations—of *co-finite* quantity—are unchanged. For example, the summary for the instruction $\mathsf{y} := \mathsf{x}$ should presumably change correlations that involve the variables x or y, but must surely *not* change any *other* correlation.

$$
\begin{array}{rcl}
\mathcal{K} \fatsemi \mathcal{T} & = & \text{Unreachable} \quad \text{if } \mathcal{K} = \text{Unreachable or } \mathcal{T} = (\mathcal{I}, \text{Unreachable}, \mathcal{O}) \\
\mathcal{K} \fatsemi \mathcal{T} & = & \text{Undef} \quad \text{if } \mathcal{K} = \text{Undef or } \mathcal{T} = (\mathcal{I}, \text{Undef}, \mathcal{O}) \\
\mathcal{K}_1 \fatsemi (\mathcal{I}, \mathcal{K}_2, \mathcal{O}) & = & \begin{cases}
\bigcup_{\{(x,z) \mid \exists y_0, (x,y_0) \in \mathrm{dom}\, \mathcal{K}_1 \wedge (y_0,z) \in \mathrm{dom}\, \mathcal{K}_2\}} \\
\qquad \left\{ (x,z) \mapsto \bigsqcap_{\{y \mid (x,y) \in \mathrm{dom}\, \mathcal{K}_1 \wedge (y,z) \in \mathrm{dom}\, \mathcal{K}_2\}} \mathcal{K}_1(x,y) \fatsemi \mathcal{K}_2(y,z) \right\} \\
\sqcap \bigcup_{\{(x,y) \in \mathrm{dom}\, \mathcal{K}_1 \mid (y,y) \notin \mathrm{dom}\, \mathcal{K}_2 \wedge y \in \mathcal{O}\}} \{(x,y) \mapsto \mathcal{K}_1(x,y) \fatsemi \top\} \\
\sqcap \bigcup_{\{(x,y) \in \mathrm{dom}\, \mathcal{K}_1 \mid (y,y) \notin \mathrm{dom}\, \mathcal{K}_2 \wedge y \notin \mathcal{O}\}} \{(x,y) \mapsto \mathcal{K}_1(x,y)\} \\
\sqcap \bigcup_{y \in \mathcal{O}} \{(\star, y) \mapsto \top\} \\
\qquad \text{when } \forall y \in \mathcal{I} \cup \{\star\}, \exists x, (x,y) \in \mathrm{dom}\, \mathcal{K}_1 \\
\text{Undef} \quad \text{otherwise}
\end{cases} \\
& & \text{when } \mathcal{K}_1 \notin \{\text{Undef}, \text{Unreachable}\} \text{ and } \mathcal{K}_2 \notin \{\text{Undef}, \text{Unreachable}\}
\end{array}
$$

Fig. 12. Intra-procedural composition.

This co-finite aspect of transformations indicates that an effect is localised on a finite number of correlations. This aspect is specific to transformations, and is absent from intra-procedural values.

Transformations $\mathcal{T}$ consist of three parts: a set of input variables $\mathcal{I}$ that are read by the instruction, a correlation map $\mathcal{K}$ that denotes how correlations are transformed, and a set of output variables $\mathcal{O}$ that are written by the instruction. Variables in the complement of $\mathcal{O}$ are assured not to be modified.

The composition of an intra-procedural value and a transformation is defined in Fig. 12. If one of the two operands is Unreachable, so is the result. If one of the two operands is Undef, so is the result. The remaining case is the crux of the definition. It builds a correlation map from the intra-procedural correlation map $\mathcal{K}_1$ and the transformation correlation map $\mathcal{K}_2$, and distinguishes two cases. Let us start with the second case: if some input of the transformation may not be defined by $\mathcal{K}_1$, then the result is Undef. Otherwise, all the inputs are defined, and we are in the first case, where the result is defined as the meet of four correlation maps. We review them one by one.

The first map is—again—reminiscent of a matrix product. It builds mappings between variables of the input state and the output state by exploiting the correlations of the variables y in the middle state such that $\mathcal{K}_1(x, y)$ and $\mathcal{K}_2(y, z)$ are defined, taking their composition $\mathcal{K}_1(x, y) \fatsemi \mathcal{K}_2(y, z)$, and then iterating over all such ys and taking their meet. The use of meet is correct, since we know that all the ys are *defined* in the middle state.

The second map keeps information about the variant cases of the inputs, while forgetting information about the *output variables*, if nothing was specified in $\mathcal{K}_2$. These bindings allows the analysis to track more changes of variant cases. The rule behaves as if $\mathcal{K}_2$ contained an implicit binding $(y, y) \mapsto \top$ for any y in the set of outputs $\mathcal{O}$. Remember that in general, composing with $\top$ returns a correlation that is *not* $\top$ (*cf.* § 3).

The third map is the one that requires that *non-output variables* remain unchanged unless specified otherwise. The rule behaves as if, for any variable $y \notin \mathcal{O}$, there were an implicit binding $(y, y) \mapsto \mathrm{Eq}_\tau$, where τ is the type of y. Here, we exploit (*cf.* § 3) that $C \fatsemi \mathrm{Eq}_\tau = C$.

Finally, the fourth map ensures that for every output variable y, there is a mapping with y on the right-hand side. These mappings record that output variables have been defined. They are important to check that the input variables of the next instruction to execute are all defined, and avoid Undef of being returned later. To this end, we introduce the ghost variable $\star$ on the left-hand side of the binding, but any other defined variable would also work.

We can now define the transfer functions for instructions. Recall that they are all of the form $\mathcal{K} \mapsto \mathcal{K} \fatsemi \mathcal{T}_\lambda$. Hence, we only give their transformations $\mathcal{T}_\lambda$ in Fig. 13.

Instruction	Label	Transformations (inputs, correlations, outputs)
$\mathsf{nop}()$	true	$(\emptyset, \{\}, \emptyset)$
$\mathsf{y} :=_\tau \mathsf{x}$	true	$\big(\{\mathsf{x}\}, \{(\mathsf{x},\mathsf{y}) \mapsto \mathsf{Eq}_\tau\}, \{\mathsf{y}\}\big)$
$\mathsf{x}_1 =_\tau \mathsf{x}_2$	true	$\big(\{\mathsf{x}_1,\mathsf{x}_2\}, \{(\mathsf{x}_1,\mathsf{x}_2) \mapsto \mathsf{Eq}_\tau\} \sqcap \{(\mathsf{x}_2,\mathsf{x}_1) \mapsto \mathsf{Eq}_\tau\}, \emptyset\big)$
	false	$(\{\mathsf{x}_1,\mathsf{x}_2\}, \{\}, \emptyset)$
$\mathsf{y} :=_\tau \{\mathsf{f}_1 = \mathsf{x}_1\,;\ldots;\mathsf{f}_n = \mathsf{x}_n\}$ where $\tau = \{\mathsf{f}_1 : \tau_1;\ldots;\mathsf{f}_n : \tau_n\}$	true	$\Big(\{\mathsf{x}_1,\ldots,\mathsf{x}_n\}, \bigsqcap_{i\in\{1,\ldots,n\}}\{(\mathsf{x}_i,\mathsf{y}) \mapsto C_i\}, \{\mathsf{y}\}\Big)$ where $C_i = \{\mathsf{f}_1 \to \top;\ldots;\mathsf{f}_i \to \mathsf{Eq}_{\tau_i};\ldots;\mathsf{f}_n \to \top\}^{\mathsf{R}}$
$\mathsf{y} :=_\tau \mathsf{x}.\mathsf{f}_k$ where $k \in \{1,\ldots,n\}$ and $\tau = \{\mathsf{f}_1 : \tau_1;\ldots;\mathsf{f}_n : \tau_n\}$	true	$(\{\mathsf{x}\}, \{(\mathsf{x},\mathsf{y}) \mapsto C\}, \{\mathsf{y}\})$ where $C = \{\mathsf{f}_1 \to \top;\ldots;\mathsf{f}_k \to \mathsf{Eq}_{\tau_k};\ldots;\mathsf{f}_n \to \top\}^{\mathsf{L}}$
$\mathsf{y} :=_\tau \{\mathsf{x}\ \mathtt{with}\ \mathsf{f}_k = \mathsf{z}\}$ where $k \in \{1,\ldots,n\}$ and $\tau = \{\mathsf{f}_1 : \tau_1;\ldots;\mathsf{f}_n : \tau_n\}$	true	$(\{\mathsf{x},\mathsf{z}\}, \{(\mathsf{z},\mathsf{y}) \mapsto C\} \sqcap \{(\mathsf{x},\mathsf{y}) \mapsto C'\}, \{\mathsf{y}\})$ where $C = \{\mathsf{f}_i \to \top;\ldots;\mathsf{f}_k \to \mathsf{Eq}_{\tau_k};\ldots;\mathsf{f}_n \to \top\}^{\mathsf{R}}$ and $C' = \left\{\overline{\mathsf{f}_i \to \{\overline{\mathsf{f}_j \to C''_{ij}}^{j\in\{1,\ldots,n\}}\}^{\mathsf{R}}}^{i\in\{1,\ldots,n\}}\right\}^{\mathsf{L}}$ and C''_{ij} = if $i = j$ and $i \neq k$ then Eq_{τ_i} else $\top$
$\mathsf{y} :=_\tau \mathsf{A}_k(\mathsf{x})$ where $k \in \{1,\ldots,n\}$ and $\tau = [\mathsf{A}_1 : \tau_1 \mid \ldots \mid \mathsf{A}_n : \tau_n]$	true	$(\{\mathsf{x}\}, \{(\mathsf{x},\mathsf{y}) \mapsto C\} \sqcap \{(\star,\mathsf{y}) \mapsto C'\}, \{\mathsf{y}\})$ where $C = [\mathsf{A}_1 \to \bot \mid\ldots\mid \mathsf{A}_k \to \mathsf{Eq}_{\tau_k} \mid\ldots\mid \mathsf{A}_n \to \bot]^{\mathsf{R}}$ and $C' = [\mathsf{A}_1 \to \bot \mid\ldots\mid \mathsf{A}_k \to \top \mid\ldots\mid \mathsf{A}_n \to \bot]^{\mathsf{R}}$
$[\mathsf{A}_1\!: \mathsf{y}_1 \mid\ldots\mid \mathsf{A}_n\!: \mathsf{y}_n] :=_\tau \mathtt{switch}(\mathsf{x})$ where $\tau = [\mathsf{A}_1 : \tau_1 \mid\ldots\mid \mathsf{A}_n : \tau_n]$	λ_{A_k}	$(\{\mathsf{x}\}, \{(\mathsf{x},\mathsf{y}_k) \mapsto C\} \sqcap \{(\mathsf{x},\star) \mapsto C'\}, \{\mathsf{y}_k\})$ where $C = [\mathsf{A}_1 \to \bot \mid\ldots\mid \mathsf{A}_k \to \mathsf{Eq}_{\tau_k} \mid\ldots\mid \mathsf{A}_n \to \bot]^{\mathsf{L}}$ and $C' = [\mathsf{A}_1 \to \bot \mid\ldots\mid \mathsf{A}_k \to \top \mid\ldots\mid \mathsf{A}_n \to \bot]^{\mathsf{L}}$
Function call		By instantiation of summaries.

Fig. 13. Transformations for instructions.

Nop. The transformation for nop() specifies no change at all. By unfolding the definition of composition, one can check that the induced transfer function for nop() is the identity— as expected.

Assignment. The transformation for $\mathsf{y} := \mathsf{x}$ defines x as input, y as output, and specifies that the value of x in the input state is equal to the value of y in the output state.

Equality Test. Since the instruction $\mathsf{x}_1 = \mathsf{x}_2$ has two output labels—true and false—we need to define two transformations. Both define that x_1 and x_2 are the inputs and that there is no output. In the true case, the transformation specifies that the value of x_1 read in the input state must equal the value of x_2 read in the output state, and that the same must also be true when swapping x_1 and x_2. In the false case, the transformation does not specify anything.

Record Creation. The transformation for $\mathsf{y} := \{\mathsf{f}_1 = \mathsf{x}_1\,;\ldots;\mathsf{f}_n = \mathsf{x}_n\}$ specifies that the inputs are the variables x_i and the only output is y. The correlation map uses R-sided record correlations to require that the values x_i in the input state equal the value of $\mathsf{y}.\mathsf{f}_i$ in the output state.

Record Access. The transformation for $\mathsf{y} := \mathsf{x}.\mathsf{f}_k$ specifies x as the only input variable, and y as the only output. The correlation map uses L-sided record correlations to require that the value of $\mathsf{x}.\mathsf{f}_k$ in the input equals the value of y in the output.

Record Update. The transformation for $\mathsf{y} := \{\mathsf{x}\ \mathtt{with}\ \mathsf{f}_k = \mathsf{z}\}$ defines x and z as the input variables, and y as the only output. The correlation map specifies two things. First, it uses an R-sided record correlation to enforce that the value of z read in the input state equals the value of $\mathsf{y}.\mathsf{f}_k$ in the output state. Second, it uses a record matrix—encoded with L- and R-sided record correlations—to specify that the values of $\mathsf{x}.\mathsf{f}_i$ in the input state equal the values of $\mathsf{y}.\mathsf{f}_i$ when $i \neq k$. The correlation matrix is a square matrix that has Eq on the diagonal except at "index" $(\mathsf{f}_k, \mathsf{f}_k)$ which has $\top$ instead, and contains $\top$ again everywhere outside the diagonal.

Variant Creation. The transformation for $\mathsf{y} := \mathsf{A}_k(\mathsf{x})$ defines x as the only input and y as the only output, and specifies two things. First, it uses an R-sided variant correlation to ensure that the value of x in the input state equals the value of $\mathsf{y}@\mathsf{A}_k$, and that A_k is the only possible case for y. The impossible cases are specified using the empty correlation $\bot$. Second, the correlation map ensures one more time that the value of y in the output state must be in the case A_k, by relating $\star$ with y. We will see later on that the use of $\star$ enables tracking constructor changes—when combined with the correlation of variant destruction—by using $\star$ as an intermediary.

Variant Destruction. The instruction $[\mathsf{A}_1\colon \mathsf{y}_1 \mid \ldots \mid \mathsf{A}_n\colon \mathsf{y}_n] := \mathtt{switch}(\mathsf{x})$ has as many output labels as there are constructors in the variant. We focus on one output label A_k. The transformation defines x as the only input and y_k as the only output, and specifies two correlations. First, it uses a L-sided variant correlation to ensure that the value of $\mathsf{x}@\mathsf{A}_k$ in the input state equals the value of y_k in the output state and that x must necessarily be in the case A_k. Second, the correlation map asserts again that x must be in the case A_k by relating x with $\star$.

Inter-Procedural Analysis. Because correlations relate input to output, the extension from an intra-procedural to an inter-procedural correlation analysis is rather simple. The "summary correlation" for a function is obtained from the correlation computed by the intra-procedural analysis for the exit points of the function. The summary correlation is a mapping from exit labels to transformations $\lambda \mapsto \mathcal{T}_\lambda$, where transformations $\mathcal{T}_\lambda$ are obtained from the intra-procedural correlation $\mathcal{K}_\lambda$ that was computed at the exit node λ and from which we remove bindings that involve local variables. Thus, the correlations that remain in summaries are between input variables of the function (or the ghost variable) and the output variables of the function (or the ghost variable).

Given the summary correlations, we can now define how to analyse function calls. The transformation for a call to the function f for the output label λ, amounts to instantiation: take in the summary correlation for f the transformation for the label λ, rename the formal parameters of each unary map into the actual parameters, and combine them using $\sqcap$.

Soundness. The transformations defined in Table 13 are correct in the following sense:

Theorem 5.1 (Soundness). *Let inst be an instruction in which the variable $\star$ does not occur. Let $(\mathcal{I}, \mathcal{K}, \mathcal{O})$ be the transformation for inst on output label λ. If executing inst on a state s produces a state s' and the output label λ, then:*

- *for any x of type τ_x and any y of type τ_y, $(s(\mathsf{x}), s'(\mathsf{y})) \in [\![\mathcal{K}(\mathsf{x},\mathsf{y})]\!]^{\tau_\mathsf{x} \times \tau_\mathsf{y}}$, and*
- *for any $\mathsf{z} \notin \mathcal{O}$, $s(\mathsf{z}) = s'(\mathsf{z})$.*

The soundness theorem makes explicit the fact that *only the outputs* of an instruction—*i.e.*, the variables in $\mathcal{O}$—might be modified by its execution. The relations between inputs and outputs recorded in the correlation map $\mathcal{K}$ specify *how* the outputs are modified. We proved the soundness theorem on paper. The proof is long but not difficult.

6 EXAMPLES

Now that we have defined the main ingredients of the correlation analysis, we can return to our examples from Fig. 11. In this section, for the purpose of making the reading of large correlations easier, we adopt the following *convention*: we do not print the mappings of record fields to $\top$ in record correlations, and we do not print mappings to $\bot$ in variant correlations. Moreover, we do not print the type subscript of Eq correlations either.

To run the analysis on the `set_r0` function, we initialise the entry point with the intra-procedural value $\{(\mathsf{p},\mathsf{p}) \mapsto \mathsf{Eq}; (\mathsf{v},\mathsf{v}) \mapsto \mathsf{Eq}; (\star,\star) \mapsto \top\}$, that relates input variables with themselves. After the projection `p.regs`, we have gained the new correlation $(\mathsf{p}, \mathtt{regs}) \mapsto \{\mathsf{regs} \rightarrow \mathsf{Eq}\}^{\mathsf{L}}$. Then, after the update of `regs.r0` with v, we gain the two correlations $(\mathsf{v}, \mathtt{regs}) \mapsto \{\mathsf{r0} \rightarrow \mathsf{Eq}\}^{\mathsf{R}}$ and $(\mathsf{p}, \mathtt{regs}) \mapsto \{\mathsf{regs} \rightarrow C_{\mathsf{regs}}\}^{\mathsf{L}}$ where $C_{\mathsf{regs}} = \{\mathsf{r1} \rightarrow \{\mathsf{r1} \rightarrow \mathsf{Eq}\}^{\mathsf{R}}; \mathsf{r2} \rightarrow \{\mathsf{r2} \rightarrow \mathsf{Eq}\}^{\mathsf{R}}; \mathsf{r3} \rightarrow \{\mathsf{r3} \rightarrow \mathsf{Eq}\}^{\mathsf{R}}\}^{\mathsf{L}}$. At this point of the program, the analysis has inferred that `p.regs` and `regs` only differ on their `r0` field, and that v equals `regs.r0`. Then, the code updates `p.regs` with `regs`, and we get two correlations

$$(\mathsf{p}, \mathsf{new\_p}) \mapsto \left\{ \begin{array}{l} \mathsf{nr} \rightarrow \{\mathsf{nr} \rightarrow \mathsf{Eq}\}^{\mathsf{R}} \\ \mathsf{regs} \rightarrow \{\mathsf{regs} \rightarrow C_{\mathsf{regs}}\}^{\mathsf{R}} \\ \mathsf{exe\_name} \rightarrow \{\mathsf{exe\_name} \rightarrow \mathsf{Eq}\}^{\mathsf{R}} \\ \mathsf{ipc\_status} \rightarrow \{\mathsf{ipc\_status} \rightarrow \mathsf{Eq}\}^{\mathsf{R}} \end{array} \right\}^{\mathsf{L}} \qquad (\mathsf{v}, \mathsf{new\_p}) \mapsto \left\{\mathsf{regs} \rightarrow \{\mathsf{r0} \rightarrow \mathsf{Eq}\}^{\mathsf{R}}\right\}^{\mathsf{R}}$$

They constitute the final result of the analysis for `set_r0`. In the end, the analysis has inferred that the input p and the result $\mathsf{new\_p}$ have the same values everywhere but on the path `.regs.r0`, and that `new_p.regs.r0` has been set to the value of the input variable v. The analysis therefore inferred for `set_r0` a sound *and complete* relation between input and output.

Our next example `clean_proc_refs` illustrates the tracking of constructor cases—enabled by variant correlations. The analysis is again initialised with $\{(\mathsf{p},\mathsf{p}) \mapsto \mathsf{Eq}; (\mathtt{i},\mathtt{i}) \mapsto \mathsf{Eq}; (\star,\star) \mapsto \top\}$ on the entry node. The entry node simply forwards the equality between p and $\mathsf{new\_p}$. Then, after the projection `new_p.ipc_status`, the analysis infers $(\mathsf{p}, \mathtt{ipc\_status}) \mapsto \{\mathsf{ipc\_status} \rightarrow \mathsf{Eq}\}^{\mathsf{L}}$. Then, a case analysis is performed on `ipc_status`. In the case of the Ready outgoing edge, the new inferred correlation is $(\mathsf{p}, \star) \mapsto \{\mathsf{ipc\_status} \rightarrow [\mathsf{Ready} \rightarrow \top]^{\mathsf{L}}\}^{\mathsf{L}}$, denoting that `p.ipc_status` is in the case Ready. Similar correlations are inferred on the other branches. All these branches are eventually *joined* at the entrance of the instruction `unblocked := F`. At this join point, we get $(\mathsf{p}, \star) \mapsto \{\mathsf{ipc\_status} \rightarrow [\mathsf{Ready} \rightarrow \top \mid \mathsf{Sleeping} \rightarrow \top \mid \mathsf{Receiving} \rightarrow \top \mid \mathsf{Sending} \rightarrow \top]^{\mathsf{L}}\}^{\mathsf{L}}$ which provides no useful information on the variant case of `p.ipc_status`. This correlation is however exploited by the transfer function for `unblocked := F`, which gives back

$$(\mathsf{p}, \mathsf{unblocked}) \mapsto \left\{ \mathsf{ipc\_status} \rightarrow \left[\begin{array}{l} \mathsf{Ready} \rightarrow [\mathsf{F} \rightarrow \top]^{\mathsf{R}} \mid \mathsf{Sleeping} \rightarrow [\mathsf{F} \rightarrow \top]^{\mathsf{R}} \\ \mathsf{Receiving} \rightarrow [\mathsf{F} \rightarrow \top]^{\mathsf{R}} \mid \mathsf{Sending} \rightarrow [\mathsf{F} \rightarrow \top]^{\mathsf{R}} \end{array} \right]^{\mathsf{L}} \right\}^{\mathsf{L}}$$

That correlation was obtained by using $\star$ as a middle variable in the composition. It reads: "whichever is the case of `p.ipc_status`, the value of $\mathsf{unblocked}$ is necessarily in the case F".

Some more precise information about the cases of `p.ipc_status` is kept in the `Receiving` and `Sending` branches of the case analysis, so that at the join point `ready := Ready`, we get $(\mathsf{p}, \star) \mapsto \{\mathsf{ipc\_status} \rightarrow [\mathsf{Sending} \rightarrow \top \mid \mathsf{Receiving} \rightarrow \top]^{\mathsf{L}}\}^{\mathsf{L}}$. Then, after the definition of `ready`, the correlations $(\mathsf{p}, \mathtt{ready}) \mapsto \{\mathsf{ipc\_status} \rightarrow [\mathsf{Sending} \rightarrow [\mathsf{Ready} \rightarrow \top]^{\mathsf{R}} \mid \mathsf{Receiving} \rightarrow [\mathsf{Ready} \rightarrow \top]^{\mathsf{R}}]^{\mathsf{L}}\}^{\mathsf{L}}$ and $(\mathsf{p}, \star) \mapsto \{\mathsf{ipc\_status} \rightarrow [\mathsf{Sending} \rightarrow \top \mid \mathsf{Receiving} \rightarrow \top]^{\mathsf{L}}\}^{\mathsf{L}}$ are obtained, meaning that `p.ipc_status` is either `Sending` or `Receiving`, and that `ready` is set to `Ready`.

The update of `new_p.ipc_status` will interestingly exploit two correlations: first, the composition of the former correlation for the pair $(\mathsf{p}, \mathtt{new\_p})$ with the transfer correlation for the pair (new_p, new_p), and second, the composition of the former correlation of the pair (p, ready) and the transfer correlation of the pair (ready, new_p). The first composition is a correlation saying that p and new_p may only differ on the path `.ipc_status`, whereas the second composition says that between `p.ipc_status` and `new_p.ipc_status`, the only possible evolutions of constructors are from `Receiving` to Ready, or from `Sending` to Ready. By taking their *meet*, we obtain

$$(\mathsf{p}, \mathsf{new\_p}) \mapsto \left\{\begin{array}{l} \mathsf{nr} \rightarrow \{\mathsf{nr} \rightarrow \mathsf{Eq}\}^{\mathsf{R}}; \mathsf{regs} \rightarrow \{\mathsf{regs} \rightarrow \mathsf{Eq}\}^{\mathsf{R}}; \mathsf{exe\_name} \rightarrow \{\mathsf{exe\_name} \rightarrow \mathsf{Eq}\}^{\mathsf{R}} \\ \mathsf{ipc\_status} \rightarrow \left\{ \mathsf{ipc\_status} \rightarrow \left[\begin{array}{l} \mathsf{Receiving} \rightarrow [\mathsf{Ready} \rightarrow \top]^{\mathsf{R}} \\ \mathsf{Sending} \rightarrow [\mathsf{Ready} \rightarrow \top]^{\mathsf{R}} \end{array}\right]^{\mathsf{L}} \right\}^{\mathsf{R}} \end{array}\right\}^{\mathsf{L}}$$

which is then transformed by the call to `set_r0` into

$$(\mathsf{p}, \mathsf{new\_p}) \mapsto \left\{\begin{array}{l} \mathsf{nr} \rightarrow \{\mathsf{nr} \rightarrow \mathsf{Eq}\}^{\mathsf{R}}; \mathsf{regs} \rightarrow \{\mathsf{regs} \rightarrow C_{\mathsf{regs}}\}^{\mathsf{R}}; \mathsf{exe\_name} \rightarrow \{\mathsf{exe\_name} \rightarrow \mathsf{Eq}\}^{\mathsf{R}} \\ \mathsf{ipc\_status} \rightarrow \left\{ \mathsf{ipc\_status} \rightarrow \left[\begin{array}{l} \mathsf{Receiving} \rightarrow [\mathsf{Ready} \rightarrow \top]^{\mathsf{R}} \\ \mathsf{Sending} \rightarrow [\mathsf{Ready} \rightarrow \top]^{\mathsf{R}} \end{array}\right]^{\mathsf{L}} \right\}^{\mathsf{R}} \end{array}\right\}^{\mathsf{L}}$$

The difference with the previous correlation is that the value at the path `.regs.r0` might have changed. The next instruction on this branch is `unblocked := T`. The analysis infers

$$(\mathsf{p}, \mathsf{unblocked}) \mapsto \left\{ \mathsf{ipc\_status} \rightarrow \left[\mathsf{Receiving} \rightarrow [\mathsf{T} \rightarrow \top]^{\mathsf{R}} \mid \mathsf{Sending} \rightarrow [\mathsf{T} \rightarrow \top]^{\mathsf{R}} \right]^{\mathsf{L}} \right\}^{\mathsf{L}}$$

by exploiting one more time correlations that involve the ghost variable. Finally, at the exit point we join the correlations that were computed in the two incoming branches and get

$$(\mathsf{p}, \mathsf{new\_p}) \mapsto C_{\mathsf{cleared}} \quad (\mathsf{p}, \mathsf{unblocked}) \mapsto \left\{ \mathsf{ipc\_status} \rightarrow \left[\begin{array}{l} \mathsf{Ready} \rightarrow [\mathsf{F} \rightarrow \top]^{\mathsf{R}} \\ \mathsf{Sleeping} \rightarrow [\mathsf{F} \rightarrow \top]^{\mathsf{R}} \\ \mathsf{Receiving} \rightarrow [\mathsf{F} \rightarrow \top \mid \mathsf{T} \rightarrow \top]^{\mathsf{R}} \\ \mathsf{Sending} \rightarrow [\mathsf{F} \rightarrow \top \mid \mathsf{T} \rightarrow \top]^{\mathsf{R}} \end{array}\right]^{\mathsf{L}} \right\}^{\mathsf{L}}$$

where

$$C_{\mathsf{cleared}} = \left\{\begin{array}{l} \mathsf{nr} \rightarrow \{\mathsf{nr} \rightarrow \mathsf{Eq}\}^{\mathsf{R}}; \mathsf{regs} \rightarrow \{\mathsf{regs} \rightarrow C_{\mathsf{regs}}\}^{\mathsf{R}}; \mathsf{exe\_name} \rightarrow \{\mathsf{exe\_name} \rightarrow \mathsf{Eq}\}^{\mathsf{R}} \\ \mathsf{ipc\_status} \rightarrow \left\{ \mathsf{ipc\_status} \rightarrow \left[\begin{array}{l} \mathsf{Ready} \rightarrow \left[\mathsf{Ready} \rightarrow \mathsf{Eq}\right]^{\mathsf{R}} \\ \mathsf{Sleeping} \rightarrow \left[\mathsf{Sleeping} \rightarrow \mathsf{Eq}\right]^{\mathsf{R}} \\ \mathsf{Receiving} \rightarrow \left[\mathsf{Ready} \rightarrow \top \mid \mathsf{Receiving} \rightarrow \mathsf{Eq}\right]^{\mathsf{R}} \\ \mathsf{Sending} \rightarrow \left[\mathsf{Ready} \rightarrow \top \mid \mathsf{Sending} \rightarrow \mathsf{Eq}\right]^{\mathsf{R}} \end{array}\right]^{\mathsf{L}} \right\}^{\mathsf{R}} \end{array}\right\}^{\mathsf{L}}$$

In the end, the analysis inferred the following pieces of information for `clean_proc_refs`:

(1) only the paths `.regs.r0` and `.ipc_status` might have changed between p and new_p;
(2) between these two values, the field `ipc_status` was not modified if it was initially in the cases Ready or `Sleeping`;
(3) the field `ipc_status` was either not modified, or was promoted to Ready if it was initially in the cases `Receiving` of `Sending`;
(4) if the value `p.ipc_status` was either in the Ready or in the `Sleeping` case, then unblocked is necessarily in the case F.

The correlation for the pair (p, unblocked) is ⊑-equivalent to the correlation $(\mathsf{p}, \mathsf{unblocked}) \mapsto [\mathsf{T} \rightarrow \{\mathsf{ipc\_status} \rightarrow [\mathsf{Receiving} \rightarrow \top \mid \mathsf{Sending} \rightarrow \top]^{\mathsf{L}}\}^{\mathsf{L}} \mid \mathsf{F} \rightarrow \top]^{\mathsf{R}}$ which reads: "if unblocked was eventually set to T, then `p.ipc_status` was initially `Receiving` or `Sending`".

7 EXTENDING THE ANALYSIS TO FUNCTIONAL ARRAYS

The analysis has been extended to work with *functional* arrays. We have proved on paper the soundness of the *whole* analysis, including the support for arrays. The proof covers the order theoretic properties, the semantic soundness of the operations on correlations, and the soundness of all the transfer functions. Due to space limitations, we only give the main ideas behind the array extension. The proposed solution is not straightforward but it is necessary in order to maintain an acceptable precision for the analysis of our OS case study. Our solution for arrays is *on purpose* not the most expressive: we restricted the expressiveness to keep the analysis theoretically simple and computationally reasonable.

Types and Values. Types are extended with array types $\langle \tau_1 \longrightarrow \tau_2 \rangle$ whose indices have type τ_1 and cells have type τ_2. Values are extended with array values—*i.e.*, finite maps from values to values—and the syntax of paths is extended with an array access path $[\mathtt{i}]p$. The projection of values on paths now takes an environment as extra parameter, in order to evaluate the indices in paths.

Instructions. We add two new instructions to the language.

$$\begin{array}{rcll} \mathit{inst} & ::= & \ldots & \\ & \mid & \mathtt{y} :=_\tau \mathtt{x[z]} & \text{(Array access)} \\ & \mid & \mathtt{y} :=_\tau [\mathtt{x}\ \mathtt{with}\ \mathtt{z} = \mathtt{x}'] & \text{(Array update)} \end{array}$$

The array *access* instruction $\mathtt{y} := \mathtt{x[i]}$ reads the value v_1 of $\mathtt{x}$—which must be an array—and the value v_2 of $\mathtt{i}$—which must be a value whose type is the type of the indices of v_1—and then defines $\mathtt{y}$ to be the value extracted from v_1 at the index v_2. If there is no such value, then the instruction raises the label false. The array *update* instruction $\mathtt{y} := [\mathtt{x}\ \mathtt{with}\ \mathtt{i} = \mathtt{x}']$ reads the value v_1 of $\mathtt{x}$—which must be an array—and the value v_2 of $\mathtt{i}$—which must be a value whose type is the type of indices of v_1—and also reads the value v_3 of $\mathtt{x}'$. Then, the value for $\mathtt{y}$ is defined to be the array v_1 in which the value at index v_2 has been replaced with v_3. If v_2 is an invalid index, then the instruction raises the label false.

Correlations. We extend the syntax of correlations with three new cases.

$$C \quad ::= \quad \ldots \mid \langle \mathtt{i} \longrightarrow C \rangle^{\mathcal{S}} \mid \langle \mathtt{i} \Rightarrow C; * \Rightarrow C \rangle \mid \langle * \Rightarrow C \rangle$$

The array access $\langle \mathtt{i} \longrightarrow C \rangle^{\mathcal{S}}$ specifies that the value on the $\mathcal{S}$ side must be an array value, and that the value of the variable $\mathtt{i}$ *must be a valid index* in that array, and the value of the array at this index must be related by C to the value on the other side. This correlation is typically used with the L side in the transformation for the array access instruction for the true label, and with the R side in the transformation for the array update instruction for the true label.

The pointwise correlation $\langle \mathtt{i} \Rightarrow C_{\mathsf{exn}}; * \Rightarrow C_{\mathsf{def}} \rangle$ denotes a relation between two arrays *with the same domains*, such that the two arrays are pointwise related by the *default* correlation C_{def}, except at the index that is equal to the value of $\mathtt{i}$—if it ever is a valid index—where the two cells must be related by the *exceptional* correlation C_{exn}. This correlation is used in the transformation for the array update instruction to specify that the value at only one index might have changed.

Finally, the correlation $\langle * \Rightarrow C \rangle$ is an over-approximation of the previous one, that appears during joins. It again relates two arrays *with the same domains*, that are pointwise related by C.

It is a design choice to track correlations about *at most one* index in arrays. Though it restricts the expressiveness and the precision of the analysis, it has the advantage of keeping the theory tractable. Supporting several array indices would indeed require to determine precisely, at every program point, which indices have identical or distinct values. Moreover, handling several indices in the definition of $\sqsubseteq$ would require that we quantify over *all the equality classes* of variables that

Instruction	Label	Transformations (inputs, correlations, outputs)
$\mathtt{y} :=_{\langle\tau\to\tau'\rangle} \mathtt{x[i]}$	true	$(\{\mathtt{x},\mathtt{i}\}, \{(\mathtt{x},\mathtt{y}) \mapsto \langle \mathtt{i} \to \mathtt{Eq}_{\tau'}\rangle^{\mathsf{L}}\}, \{\mathtt{y}\})$
	false	$(\{\mathtt{x},\mathtt{i}\}, \{\}, \emptyset)$
$\mathtt{y} :=_{\langle\tau\to\tau'\rangle} [\mathtt{x}\ \mathtt{with}\ \mathtt{i} = \mathtt{x}']$	true	$(\{\mathtt{x},\mathtt{i},\mathtt{x}'\}, \{(\mathtt{x},\mathtt{y}) \mapsto C\} \sqcap \{(\mathtt{x}',\mathtt{y}) \mapsto C'\}, \{\mathtt{y}\})$ where $C = \langle \mathtt{i} \Rightarrow \top; * \Rightarrow \mathtt{Eq}_{\tau'}\rangle$ and $C' = \langle \mathtt{i} \to \mathtt{Eq}_{\tau'}\rangle^{\mathsf{R}}$
	false	$(\{\mathtt{x},\mathtt{i},\mathtt{x}'\}, \{\}, \emptyset)$

Fig. 14. Transformations for array instructions.

occur in correlations. This could be prohibitively costly. Our experimental results show that we obtain satisfactory precision in spite of the limitation to one index per array (§ 8).

Transformations. Array-related transformations are shown in Fig. 14. As expected, the transformation for array access $\mathtt{y} := \mathtt{x[i]}$ specifies that $\mathtt{x[i]} = \mathtt{y}$, while the transformation for the array update $\mathtt{y} := [\mathtt{x}\ \mathtt{with}\ \mathtt{i} = \mathtt{x}']$ specifies that $\mathtt{x}' = \mathtt{y[i]}$ and that $\mathtt{x}$ and $\mathtt{y}$ may only differ at index $\mathtt{i}$.

Pre-Order. We give only two representative rules for arrays in the pre-order definition. Paths are extended with array accesses $[\mathtt{i}]p$ and the definitions for projections are extended as well. The Array rule follows the same projection-based style that was already used in the definition of $\sqsubseteq$, but has a new premise $C' \vdash^{\mathcal{S}} [\mathtt{i}]$ valid that checks that $[\mathtt{i}]$ is a valid access path for every value on the $\mathcal{S}$ side of C'. The judgement $C' \vdash^{\mathcal{S}} [\mathtt{i}]$ valid is defined by looking at the presence of a sub-correlation $\langle \mathtt{i} \to C''\rangle^{\mathcal{S}}$ in C'. This check is mandatory to ensure that the pre-order is sound with respect to the semantics of correlations. For example, neither $\top$ nor $\mathtt{Eq}$ are more precise than $\langle \mathtt{i} \to \top\rangle^{\mathsf{L}}$, because none of them ensures that array values on their left-hand sides enjoy $[\mathtt{i}]$ as a valid access path. A second representative rule is the rule that forgets an index in a pointwise array correlation. It says that an array correlation with default C'_{def} and exception C'_{exn} is more precise than an array correlation with default C_{def} when both the default and the exceptional correlations are more precise than C_{def}. Interestingly, $\mathtt{Eq}_{\langle\tau\to\tau'\rangle}$ is *strictly* more precise than $\langle * \Rightarrow \mathtt{Eq}_{\tau'}\rangle$—meaning that the pre-order does not include extensionality on arrays. Note also that $\top$ is *not* more precise than $\langle * \Rightarrow \top\rangle$, because $\top$ does not ensure that the related array values have the same domains.

$$\textsc{Array}\quad \frac{C' \vdash^{\mathcal{S}} [\mathtt{i}]\ \text{valid} \qquad C' \Downarrow^{\mathcal{S}} [\mathtt{i}] \sqsubseteq C}{C' \sqsubseteq \langle \mathtt{i} \to C\rangle^{\mathcal{S}}} \qquad\qquad \textsc{ArrayDiagForget}\quad \frac{C'_{\mathrm{exn}} \sqsubseteq C_{\mathrm{def}} \qquad C'_{\mathrm{def}} \sqsubseteq C_{\mathrm{def}}}{\langle \mathtt{i} \Rightarrow C'_{\mathrm{exn}}; * \Rightarrow C'_{\mathrm{def}}\rangle \sqsubseteq \langle * \Rightarrow C_{\mathrm{def}}\rangle}$$

Join. We present a few selected rules for the extension of join with arrays. The join of two single-sided array correlations $\langle \mathtt{i} \to C\rangle^{\mathcal{S}}$ and $\langle \mathtt{j} \to C'\rangle^{\mathcal{S}}$, for example, is defined as $\langle \mathtt{i} \to C \sqcup C'\rangle^{\mathcal{S}}$ when the two variables $\mathtt{i}$ and $\mathtt{j}$ are equal. This is correct, because *both sides* of the join ensure that the access at index $\mathtt{i}$ is valid. When the variables are different, however, the join is defined to be $\top$, and there is no better correlation. In particular, the result cannot be an array correlation, because it would require the array access at index $\mathtt{i}$ (or $\mathtt{j}$) to be valid in both sides of the join.

$$\begin{aligned}
\langle \mathtt{i} \to C\rangle^{\mathcal{S}} \sqcup \langle \mathtt{j} \to C'\rangle^{\mathcal{S}} &= \text{if } \mathtt{i} = \mathtt{j} \text{ then } \langle \mathtt{i} \to C \sqcup C'\rangle^{\mathcal{S}} \text{ else } \top \\
\langle \mathtt{i} \to C\rangle^{\mathcal{S}} \sqcup \langle * \Rightarrow C'\rangle &= \top \\
\begin{array}{l}\langle \mathtt{i} \Rightarrow C_{\mathrm{exn}}; * \Rightarrow C_{\mathrm{def}}\rangle \\ \sqcup \langle \mathtt{j} \Rightarrow C'_{\mathrm{exn}}; * \Rightarrow C'_{\mathrm{def}}\rangle\end{array} &= \begin{cases} \langle \mathtt{i} \Rightarrow C_{\mathrm{exn}} \sqcup C'_{\mathrm{exn}}; * \Rightarrow C_{\mathrm{def}} \sqcup C'_{\mathrm{def}}\rangle & \text{if } \mathtt{i} = \mathtt{j} \\ \langle * \Rightarrow (C_{\mathrm{exn}} \sqcup C_{\mathrm{def}}) \sqcup (C'_{\mathrm{exn}} \sqcup C'_{\mathrm{def}})\rangle & \text{otherwise} \end{cases}
\end{aligned}$$

The join of a single-sided and a pointwise array correlation also results in $\top$. Indeed, it cannot be a single-sided array correlation, because only one side ensures the validity of the array access, and it cannot be a pointwise correlation, since only one side ensures that the related arrays have the same domains. The join of two pointwise correlations with exceptions is recursively defined on each component when the two exceptions deal with the same variable. Otherwise, both pointwise correlations are over-approximated to forget their exceptional correlation, before being joined.

Composition. The rule for composing an R-sided array correlation with an L-sided one is reminiscent of the composition of record correlations when the two indices are the same. For this rule to be sound, it is crucial that the index be a *valid access* in the arrays, which is precisely why we enforced the validity of indices in single-sided correlations. When the indices are different, composition with $\top$ is used: $C \mathbin{⨟} \top$ exploits the correlation for the index `i`, whereas $\top \mathbin{⨟} C'$ exploits the correlation of `j`. Finally, composing two pointwise correlations is the pointwise correlation of their composition.

$$\begin{array}{lcl} \langle \mathtt{i} \rightarrow C\rangle^{\mathsf{R}} \mathbin{⨟} \langle \mathtt{j} \rightarrow C'\rangle^{\mathsf{L}} & = & \text{if } \mathtt{i} = \mathtt{j} \text{ then } C \mathbin{⨟} C' \text{ else } (C \mathbin{⨟} \top) \sqcap (\top \mathbin{⨟} C') \\ \langle * \Rightarrow C\rangle \mathbin{⨟} \langle * \Rightarrow C'\rangle & = & \langle * \Rightarrow C \mathbin{⨟} C'\rangle \end{array}$$

Transfer Functions. With the array extension, the transfer functions become of the form $\mathcal{K} \mapsto \mathrm{erase}_O\,(\mathcal{K} \mathbin{⨟} \mathcal{T})$ where O is the set of outputs in $\mathcal{T}$, and the erasure operator is responsible for removing a set of indices from all the array correlations of an intra-correlation. The erasure is necessary for the analysis to remain sound. The reason is that the values of indices might have been modified, but the indices in array correlations could refer to the values *before* the update of indices. The erasure operator forgets the pieces of information about array indices that were possibly modified, since they might be out of sync. Erasure is defined on array correlations as follows, and is naturally extended to other correlations.

$$\begin{array}{lcl} \mathrm{erase}_V\,(\langle \mathtt{i} \Rightarrow C_{\mathrm{exn}}; * \Rightarrow C_{\mathrm{def}}\rangle) & = & \begin{cases} \langle \mathtt{i} \Rightarrow \mathrm{erase}_V\,(C_{\mathrm{exn}})\,; * \Rightarrow \mathrm{erase}_V\,(C_{\mathrm{def}})\rangle & \text{when } \mathtt{i} \notin V \\ \langle * \Rightarrow \mathrm{erase}_V\,(C_{\mathrm{exn}}) \sqcup \mathrm{erase}_V\,(C_{\mathrm{def}})\rangle & \text{when } \mathtt{i} \in V \end{cases} \\ \mathrm{erase}_V\,\left(\langle \mathtt{i} \rightarrow C\rangle^{\mathcal{S}}\right) & = & \begin{cases} \langle \mathtt{i} \rightarrow \mathrm{erase}_V\,(C)\rangle^{\mathcal{S}} & \text{when } \mathtt{i} \notin V \\ \top \mathbin{⨟} \mathrm{erase}_V\,(C) & \text{when } \mathtt{i} \in V \text{ and } \mathcal{S} = \mathsf{L} \\ \mathrm{erase}_V\,(C) \mathbin{⨟} \top & \text{when } \mathtt{i} \in V \text{ and } \mathcal{S} = \mathsf{R} \end{cases} \end{array}$$

Erasure removes offending variables from pointwise array domains by joining the exceptional and default domains, thereby creating an exceptionless pointwise correlation. For array access correlations, erasure removes the offending variables by unboxing the inner correlation, and composing with $\top$ so as to keep the operation well-typed.

Summaries. Turning intra-procedural correlations into transfer summaries now requires performing erasure too, in order to only keep occurrences of input variables in the correlations. For this operation to be sound, the input variables must never have been written by any instruction. This amounts to an easy syntactic check.

Examples, Continued. Using the examples from § 6 we can finally complete the code of `kill_proc`, that we introduced in § 2.1. First, Fig. 15 defines some basic building blocks: the function `get_proc`—that extracts from the process table the process descriptor at a given index if there is any, or raises `absent` otherwise—and the function `rm_proc`—that puts `None` at some index of the process table—and also the function `set_proc`—that sets a process descriptor at some index of the process table. We assume the existence of two functions `enqueue` and `dequeue` that might only modify the field sched of the state, *i.e.*, the scheduler state. Then, we define `clear_all_refs` in Fig. 16 by iterating over all present processes and applying `clear_proc_refs` on each of them and—if the process was

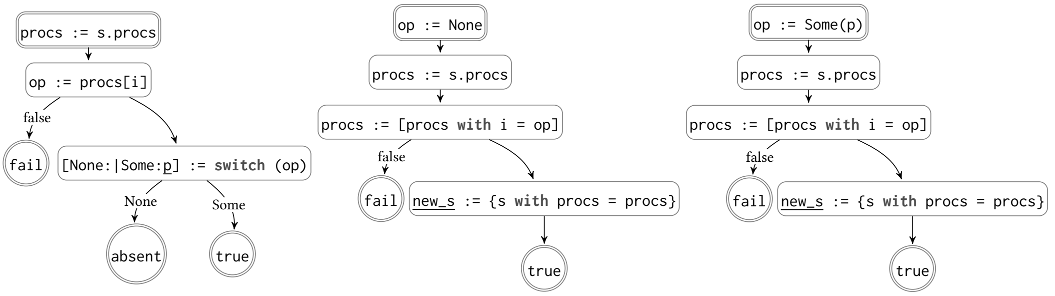
procs := s.procs
op := procs[i]
false
fail
[None:|Some:p] := switch (op)
None
Some
absent
true
op := None
procs := s.procs
procs := [procs with i = op]
false
fail
new_s := {s with procs = procs}
true
op := Some(p)
procs := s.procs
procs := [procs with i = op]
false
fail
new_s := {s with procs = procs}
true

Fig. 15. Left: the function get_proc with signature (state s, int i) -> [true:proc p|absent|fail]. Center: the function rm_proc with signature (state s, int i) -> [true:state new_s|fail]. Right: the set_proc function, with signature (state s, int i, proc p) -> [true:state new_s|fail].

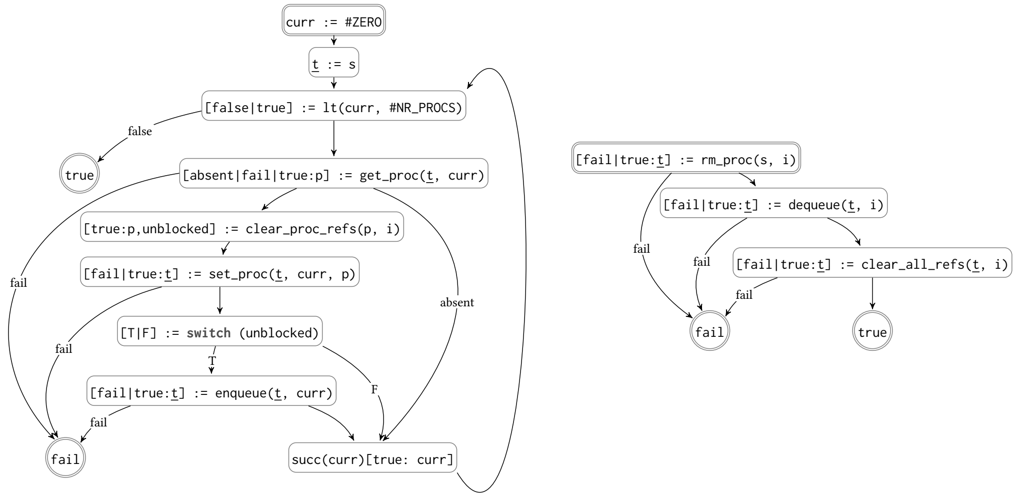
curr := #ZERO
t := s
[false|true] := lt(curr, #NR_PROCS)
false
true
[absent|fail|true:p] := get_proc(t, curr)
[true:p,unblocked] := clear_proc_refs(p, i)
[fail|true:t] := set_proc(t, curr, p)
fail
fail
absent
[T|F] := switch (unblocked)
T
F
[fail|true:t] := enqueue(t, curr)
fail
fail
succ(curr)[true: curr]
[fail|true:t] := rm_proc(s, i)
[fail|true:t] := dequeue(t, i)
fail
fail
[fail|true:t] := clear_all_refs(t, i)
fail
fail
true

Fig. 16. Left: the clear_all_refs function, with signature (state s, int i) -> [true:state t|fail]. Right: the kill_proc function, with signature (state s, int i) -> [true:state t|fail].

unblocked—calling enqueue, thereby inserting that process in the scheduling queue. Registering the processes that were unblocked is performed in order to preserve an invariant that is often found in an OS: the scheduling queue contains exactly the processes that are ready to run. Finally, kill_proc is defined by removing the process that ought to be killed, then by calling dequeue and finally by calling clear_all_refs to remove any reference to the killed process. The analysis infers for the function clear_all_refs the summary $(s, t) \mapsto \{\mathsf{procs} \to \{\mathsf{procs} \to \langle * \Rightarrow C_{\mathrm{opt}}\rangle\}^{\mathsf{R}}\}^{\mathsf{L}}$ where $C_{\mathrm{opt}} = \big[\mathsf{None} \to [\mathsf{None} \to \mathsf{Eq}]^{\mathsf{R}} \mid \mathsf{Some} \to [\mathsf{Some} \to \{\mathsf{x} \to \{\mathsf{x} \to C_{\mathrm{cleared}}\}^{\mathsf{R}}\}^{\mathsf{L}}]^{\mathsf{R}}\big]^{\mathsf{L}}$. The correlation C_{cleared} was inferred between the variables p and new_p for clear_proc_refs in § 6. The result of the analysis ensures that no process has been deleted or created and also that every process was changed according to the correlation C_{cleared}. It also specifies that the sched field might have been modified in an arbitrary way.

Finally, the result of the analysis for kill_proc is

$$(s, t) \mapsto \{\mathsf{procs} \to \{\mathsf{procs} \to \langle \mathsf{i} \Rightarrow \top; * \Rightarrow C_{\mathrm{opt}}\rangle\}^{\mathsf{R}}\}^{\mathsf{L}} \quad (\star, t) \mapsto \{\mathsf{procs} \to \langle \mathsf{i} \to [\mathsf{None} \to \top]^{\mathsf{R}}\rangle^{\mathsf{R}}\}^{\mathsf{R}}$$

which says that the field sched arbitrarily changed, and that the value at index i in the table of processes was set to None, and that no other process was removed or created, and all of them possibly changed as specified by C_{cleared}. This result is precise enough to be exploited for proof purposes. For instance, it becomes easy to prove that kill_proc preserves the invariant that get_proc(s, i).nr = i for any index where get_proc succeeds.

8 EXPERIMENTAL RESULTS

Implementation. We implemented the analysis in OCaml [Leroy et al. 2017]. In order to reduce memory usage, we used a sparse encoding of correlations, following the same convention that we used in § 6: we ensure there are no bindings to $\top$ in record correlations, and no binding to $\bot$ in variant correlations. We used Kildall's algorithm [Kildall 1973] to solve the dataflow constraints by computing a post-fixpoint. We used *widening* to ensure termination in the presence of loops. We also implemented support for recursive functions by iterating until an inter-procedural post-fixpoint is reached. Our implementation also features some simplifications of correlations that are *semantically* equivalent but not necessarily equivalent in the sense of $\sqsubseteq$. For example, any record correlation that contains a mapping to $\bot$ is simplified to $\bot$ because the two correlations are semantically equal. The latter is however strictly smaller than the former. Therefore, we never perform simplification during the computation of $\sqsubseteq$ or $\sqcup$ since that might break the pre-order and upper-bound properties, which are required by the dataflow solver. Everywhere else, performing these simplifications is sound, since correctness does not rely on order-theoretic properties.

The inferred correlations can be quite large. An upper bound of the size of a correlation is the product of the size of its input type and of the size of its output type. We adopted two strategies to reduce the size of correlations while preserving their semantics, that we found satisfying in practice. A first strategy is to rebalance *homogeneous* correlations—that relate values of the same types—using an alternation of Ls and Rs. This is the strategy we applied in all the examples of this paper. A noteworthy property is its ability to materialise diagonal matrices, which are especially compact in our sparse encoding. We applied a second strategy for *heterogeneous* correlations, that attempts to reduce the sizes of correlations by consecutive swaps of L and R correlations. Using these two strategies, we gained a few percents on the running time and memory consumption of the analysis, but also we substantially improved the readability of the inferred correlations.

Paper Examples. The full example used throughout the paper takes in total about 200 lines of code, structured in 15 function definitions. It is fully analysed in less than 6 ms on a recent computer equipped with a 64-bit 2.80GHz quad-core CPU and running Linux—a negligible time.

Real-World Example. We ran the analysis on a larger scale example, extracted from the ProvenCore [Lescuyer 2015] development. It comprises about 58000 lines of code, split over about 2900 functions and 500 type definitions. Those declarations define the actual low-level code of the OS, as well as a more abstract version of that code, that uses functional data-structures such as lists—as opposed to low-level encodings of linked lists using arrays. The type of the low-level OS state is a large record with nesting of arrays, records and variants which, when seen as a tree, counts about 240 leaves. The type of the higher-level OS state is twice smaller. The invariants of both systems as well as other proof-related definitions are also part of this considerably large example. On the same machine we used before, the whole artefact is analysed in approximately 14s. The actual version of Kill is of course much more complex than the one described in the article. Still, the inferred correlations remain precise enough to show that at most one process has been removed, and to show which fields of process descriptors were possibly modified and how they were modified.

System call	% Preserved	# Preserved	# Remaining
`ipc_send`	56%	28	22
`ipc_notify`	52%	26	24
`ipc_receive`	48%	24	26
`ipc_sendrec`	48%	24	26
`ipc_sleep`	80%	40	10
`authorize`	94%	47	3
`revoke`	94%	47	3
`change_effector`	96%	48	2
`change_target`	96%	48	2
`change_revoker`	96%	48	2
`brk`	72%	36	14
`copy`	78%	39	11
`exec`	8%	4	46
`exit`	12%	6	44
`fork`	14%	7	43
`getinfo`	100%	50	0
`kill`	12%	6	44
`iomap`	66%	33	17
`iounmap`	70%	35	15
`shm_alloc`	58%	29	21
`shm_register`	58%	29	21
`shm_unregister`	62%	31	19
`shm_transfer`	80%	40	10
`smc`	98%	49	1
`irq_acquire`	98%	49	2
`irq_release`	98%	49	2
`irq_rmpolicy`	86%	43	7
`irq_setpolicy`	92%	46	4
Total	68%	961	439

Fig. 17. Experimental results. Min: 8% (`exec`). Max: 100% (`getinfo`). Median: 72% (`brk`). Mean: 68%.

Measuring the Precision of Results. We could not reasonably compare our analysis to existing ones with similar features, because—to our knowledge—correlations are the first relational domain that supports variants. Analysing ProvenCore using domains that do not support records and variants would give poor results, because most of ProvenCore deals with such structures. For lack of a reasonable baseline for comparison, we looked at quantitative measures (size of correlations, number of inferred equalities...) to estimate the precision, but they are not very informative. Instead, to assess the precision of our analysis, we tested whether the correlation analysis actually lowered the proof effort, by designing a simple procedure that decides whether a property P is preserved by a program f. The procedure computes a *correlation* for f, which gives an upper bound of f's effect. Then, it runs a *dependency* analysis [Andreescu et al. 2015] for P, which gives a relation on states that preserves P. Finally, the procedure performs an inclusion test between the two relations using $\sqsubseteq$, to check that the changes performed by f preserve P. We considered the 28 system calls of ProvenCore and its 50 invariants (Fig. 17). The decision procedure automatically proved 68% of the 1400 preservation lemmas. Half of the system calls automatically preserve more than 70% of the invariants, and only 4 of the system calls preserve less than 50% of the invariants automatically. Those 4 system calls indeed required most of the mechanisation effort in the proof of ProvenCore: they either change large parts of the state (`exit`, `kill`), or almost completely modify *one* process (`exec`) or two processes (`fork`). We conclude that our correlation analysis effectively helps focusing the proof effort on the hard parts.

Verification. The entire meta-theory—including the extension with arrays—was formalised and proved sound on paper. We also completed the mechanised verification in Coq of § 2 and § 3 for a simpler definition of correlations with binary products and sums instead of records and variants.

Testing. In addition to proving the soundness of the analysis, we spent time to devise tests for the actual implementation. Using random generation *à la* QuickCheck [Hughes 2007], we tested two kinds of properties. We first tested *algebraic* properties, like the pre-order and upper-bound properties. Generating well-typed correlations was not much of an issue, and the tests reported for example that the simplification rules that we wrote about at the beginning of the current section broke the transitivity property of $\sqsubseteq$. By removing uses of simplifications in the definitions of comparison and join, the code more closely followed the formalisation, and the tests indeed never found any counter-example. The second family of properties that we tested are the semantic ones. By generating pairs of values (or pairs of states) in the semantics of correlations, we were able to test the soundness lemmas for the operations of comparison, join, meet, compose, and we even tested the soundness of the transfer functions. Devising the tests themselves was not problematic. Generating pairs of values in correlations, however, turned out to be quite difficult. This required solving disjunctions of conjunctions of equality constraints. We put these constraints in disjunctive normal form (DNF), to represent them as lists of unification problems [Knight 1989], which we solved using persistent union-find data-structures [Conchon and Filliâtre 2007]. We put an explicit upper bound on the number of handled disjunctions, to cut the exponential blow-up of DNF normalisation.

9 RELATED WORK

Correlation analysis of input-output relations in functional specifications was introduced by Andreescu *et al.* [Andreescu et al. 2016; Andreescu 2017] who present a static analysis that computes a safe approximation of what part of an input state of a function is copied to the output state, for a functional language similar to ours. That work has the shortcoming that the analysis is unable to track the possible cases of variants which leads to the analysis being unsound. More precisely, the comparison and composition operators are unsound in the presence of variants, and comparison is not transitive. The present abstract domains differ considerably: their vector-like nature is original, and they can express transitions between different cases of variant—an *essential* feature for the composition operator. Moreover, our machine-checked proof of semantic soundness is an additional difference compared to the previous work which provided no formal account. On a different note, that previous work described a backward analysis, whereas we present a forward analysis. We made this change because a forward analysis seemed easier to model and more natural to explain.

Illous *et. al.* [Illous et al. 2017] describe a relational shape abstract domain for inferring properties about memory structures in imperative programs. One of their goals is shared with our analysis: infer when parts of memory have not changed. Also, their analysis computes a relation between input structures and output structures. Different from ours, their analysis is based on separation logic and is limited to lists and trees. The experimental evaluation does not report results about applying this approach to a substantial example such as a micro-kernel.

Dietsch *et. al.* [Dietsch et al. 2018] use abstract interpretation to infer relations between array values at a given program point. Their analysis is based on the map equality domain, that can express "up-to" equalities and disequalities between expressions involving map variables. Their domain is geared towards expressiveness rather than scalability.

Our analysis can be seen as a (partial) solution to *the framing problem*: a general challenge in software engineering and verification [Meyer 2015]. Other approaches to verify framing conditions include the congruence closure abstract domain defined by Chang and Leino [Chang and Leino 2005], which is used to infer relations between fields of variables in object-oriented languages. Using a graph representation that is reminiscent of the eDAGs structures found in congruence closure [Gulwani et al. 2005; Nelson and Oppen 1980], they combine base domains with congruences. They support only conjunctions of constraints, whereas we support disjunctions by the means of variant correlations, which are essential to precisely analyse programs like ProvenCore. However, we only support equality as a base domain. This suggests the possibility of extending correlations to relational arithmetic domains, for example.

Points-to analyses [Das 2000; Lhoták and Hendren 2003; Steensgaard 1996; Whaley and Lam 2002] use graph-based representations to express sharing in memory heaps, whereas our correlations define equality graphs with disjunctions. Notice though that points-to analysis is about relating structures in the same state, whereas we are aiming at relating structures in different states.

A framework to infer and check relational abstractions of ML programs was introduced by [Kaki and Jagannathan 2014]. From a few domain-specific primitive relations, they (recursively) define complex relations by means of pattern matching. By comparison, our variant correlations enable case analysis but are less expressive, and do not support recursively defined relations. Their approach requires annotations on recursive functions, and delegates the checking of constraints to an SMT solver, while our correlations are completely inferred, and are solely based on dataflow analysis.

Function summaries are essential to build scalable compositional static analyses. While we have defined function summaries for an equality analysis supporting algebraic data types, function summaries were also successfully developed, for instance, in the context of pointer analysis [Das 2000], shape analysis [Illous et al. 2017; Jeannet et al. 2004], and arithmetic analysis [Farzan and Kincaid 2015; Kincaid et al. 2017].

Liquid types [Rondon et al. 2008; Vazou et al. 2017] express rich specifications of higher-order programs in refinement types. Using SMT solvers, proofs are inferred from user-written specifications. By contrast, correlation analysis infers both specifications *and* proofs, for a more restricted subset of specifications based on partial equalities. Because our specifications might be large, the fact they are *inferred* is essential. Moreover, our analysis does not require us to trust a complex solver.

10 CONCLUSIONS

We have designed, implemented and proved sound a static analysis for inferring equalities between parts of input and output of functions over algebraic data-types and arrays. To this end, we have defined an abstract domain of *correlations*, with a pre-order relation, union, intersection and sequential composition operations. This structure allows defining an inter-procedural correlation analysis in a compositional way. The analysis has been implemented and has been shown to be sufficiently precise and efficient to be able to infer correlations and discharge a large number of proofs of invariant preservation in the code of an industrial-size micro-kernel. We conclude that a relatively simple equality analysis greatly helps the interactive verification of programs.

There are several avenues for further research. First, the analysis is not intended to be guided by human intervention. In the context of a proof IDE, we can nevertheless wish that a programmer could feed the analyser with a manually proven correlation, which could be further exploited by the analyser to increase the precision for the remaining functions. The exploration of such analyser-user interaction is ongoing. Second, the analysis is so far only concerned with inferring relations "in time", *i.e.*, between different program points. Inferring correlations "in space", *i.e.*, between different sub-structures of a *same* state, could further improve the precision of the analysis. Third, a shortcoming of array correlations is their ability to single out *at most one* index in the array. We intend to support several "exceptional" indices in array correlations. This could render the analysis more costly, so there are both logical and algorithmic issues to be investigated. Fourth, we have restricted ourselves from considering higher-order functions, recursive correlations, or side effects. These restrictions are mainly motivated by the intended application, as the targeted OS specification has been constructed without these features. Finally, we intend to enhance the decision procedure sketched in the experiment section and lower the proof burden in large interactive proof projects.

ACKNOWLEDGMENTS

This material is based upon work developed at Prove & Run, and funded by Prove & Run. We thank the anonymous reviewers, and Olivier Delande, whose questions and remarks helped improve the presentation of this work.

REFERENCES

Oana Andreescu, Thomas Jensen, and Stéphane Lescuyer. 2015. Dependency analysis of functional specifications with algebraic data structures. In *Formal Methods and Software Engineering (LNCS)*, Michael Butler, Sylvain Conchon, and Fatiha Zaïdi (Eds.), Vol. 9407. Springer International Publishing, 116–133. https://doi.org/10.1007/978-3-319-25423-4_8

Oana Andreescu, Thomas Jensen, and Stéphane Lescuyer. 2016. Correlating structured inputs and outputs in functional specifications. In *Software Engineering and Formal Methods (LNCS)*, Rocco De Nicola and Eva Kühn (Eds.), Vol. 9763. Springer International Publishing, 85–103. https://doi.org/10.1007/978-3-319-41591-8_7

Oana Fabiana Andreescu. 2017. *Static analysis of functional programs with an application to the frame problem in deductive verification.* Theses. Université Rennes 1. https://tel.archives-ouvertes.fr/tel-01677897

Alexander Borgida, John Mylopoulos, and Raymond Reiter. 1995. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering* 21, 10 (1995), 785–798.

Bor-Yuh Evan Chang and K. Rustan M. Leino. 2005. Abstract Interpretation with Alien Expressions and Heap Structures. In *Verification, Model Checking, and Abstract Interpretation (LNCS)*, Radhia Cousot (Ed.), Vol. 3385. Springer Berlin Heidelberg, Berlin, Heidelberg, 147–163. https://doi.org/10.1007/978-3-540-30579-8_11

Sylvain Conchon and Jean-Christophe Filliâtre. 2007. A persistent union-find data structure. In *Proceedings of the 2007 Workshop on Workshop on ML (ML '07)*. ACM, New York, NY, USA, 37–46. https://doi.org/10.1145/1292535.1292541

Patrick Cousot and Radhia Cousot. 2002. Modular static program analysis. In *Compiler Construction (LNCS)*, R. Nigel Horspool (Ed.), Vol. 2304. Springer Berlin Heidelberg, Berlin, Heidelberg, 159–179.

Manuvir Das. 2000. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 35–46. https://doi.org/10.1145/349299.349309

Daniel Dietsch, Matthias Heizmann, Jochen Hoenicke, Alexander Nutz, and Andreas Podelski. 2018. The Map Equality Domain. In *VSTTE 2018, Proceedings of the 10th Working Conference on Verified Software: Theories, Tools, and Experiments (LNCS)*, Vol. 11294. Springer.

Azadeh Farzan and Zachary Kincaid. 2015. Compositional Recurrence Analysis. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design (FMCAD '15)*. FMCAD Inc, Austin, TX, 57–64. http://dl.acm.org/citation.cfm?id=2893529.2893544

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Programming Languages and Systems (LNCS)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer Berlin Heidelberg, Berlin, Heidelberg, 125–128.

Sumit Gulwani, Ashish Tiwari, and George C. Necula. 2005. Join Algorithms for the Theory of Uninterpreted Functions. In *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science (LNCS)*, Kamal Lodaya and Meena Mahajan (Eds.), Vol. 3328. Springer Berlin Heidelberg, Berlin, Heidelberg, 311–323.

John Hughes. 2007. QuickCheck testing for fun and profit. In *Practical Aspects of Declarative Languages (LNCS)*, Michael Hanus (Ed.), Vol. 4354. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–32.

Hugo Illous, Matthieu Lemerre, and Xavier Rival. 2017. A Relational Shape Abstract Domain. In *NASA Formal Methods (LNCS)*, Clark Barrett, Misty Davies, and Temesghen Kahsai (Eds.), Vol. 10227. Springer International Publishing, 212–229.

Inria 2017. *The Coq proof assistant reference manual.* Inria. https://coq.inria.fr/distrib/current/refman/

Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. 2004. A Relational Approach to Interprocedural Shape Analysis. In *Static Analysis (LNCS)*, Roberto Giacobazzi (Ed.), Vol. 3148. Springer Berlin Heidelberg, Berlin, Heidelberg, 246–264.

Gowtham Kaki and Suresh Jagannathan. 2014. A relational framework for higher-order shape analysis. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 311–324. https://doi.org/10.1145/2628136.2628159

Gary A. Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73)*. ACM, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945

Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. 2017. Compositional Recurrence Analysis Revisited. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 248–262. https://doi.org/10.1145/3062341.3062373

Kevin Knight. 1989. Unification: a multidisciplinary survey. *Comput. Surveys* 21, 1 (March 1989), 93–124. https://doi.org/10.1145/62029.62030

Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 2006. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes* 31, 3 (May 2006), 1–38. https://doi.org/10.1145/1127878.1127884

Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2017. *The Objective Caml system, documentation and user's manual – release 4.06.* INRIA. http://caml.inria.fr/pub/docs/manual-ocaml-4.06/

Stéphane Lescuyer. 2015. ProvenCore: towards a verified isolation micro-kernel. In *International Workshop on MILS: Architecture and Assurance for Secure Systems.*

Ondvrej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using Spark. In *Compiler Construction (LNCS)*, Görel Hedin (Ed.), Vol. 2622. Springer Berlin Heidelberg, Berlin, Heidelberg, 153–169.

J. McCarthy and P. J. Hayes. 1981. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Readings in Artificial Intelligence*, Bonnie Lynn Webber and Nils J. Nilsson (Eds.). Morgan Kaufmann, 431–450. https://doi.org/10.1016/B978-0-934613-03-3.50033-7

Bertrand Meyer. 2015. Framing the frame problem. In *Dependable Software Systems Engineering*. 193–203. https://doi.org/10.3233/978-1-61499-495-4-193

Greg Nelson and Derek C. Oppen. 1980. Fast decision procedures based on congruence closure. *J. ACM* 27, 2 (April 1980), 356–364. https://doi.org/10.1145/322186.322198

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer Publishing Company, Incorporated.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. ACM, New York, NY, USA, 32–41. https://doi.org/10.1145/237721.237727

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 53 (Dec. 2017), 31 pages. https://doi.org/10.1145/3158141

John Whaley and Monica S. Lam. 2002. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Static Analysis (LNCS)*, Manuel V. Hermenegildo and Germán Puebla (Eds.), Vol. 2477. Springer Berlin Heidelberg, Berlin, Heidelberg, 180–195.