Abstract

1 Introduction

1.1 The frame problem

The frame problem [MH69] was identified and described at least as early as 1969 by McCarthy and Hayes, in the context of Artificial Intelligence (AI). The initial description of the frame problem is the following:

> "In proving that one person could get into conversation with another, we were obliged to add the hypothesis that if a person has a telephone he still has it after looking up a number in the telephone book. If we had a number of actions to be performed in sequence we would have quite a number of conditions to write down that certain actions do not change the values of certain fluents. In fact, with n actions and m fluents we might have to write down mn such conditions."

Unsurprisingly, the frame problem manifests itself in the realm of formal software specification and deductive verification as well [BMR93]. Deductive verification methods consist in producing formal correctness proofs, by first generating a set of formal mathematical proof obligations from the program and its specification, and by subsequently discharging these. Proof obligations can sometimes be discharged automatically by using static analysis and decision procedures, but often they require the use of an interactive prover. Reducing the number of proof obligations in a verification hence becomes an essential task.

A large part of a proof consists in proving that a global invariant is preserved, despite the state changes that a program makes as it executes. A number of proof obligations deal with those parts of the state that actually have been changed. Another set of proof obligations arise from dealing with the part of the state that does not change. These may be particularly tedious to handle because it should be "evident" that the invariant holds for this part of the state. As a consequence, several specification formalisms offer the possibility of specifying what is changed and what does not change. These properties are called *frame properties*.

Frame properties help reducing the number of proof obligations. However, specifying frame properties might not seem dramatic on small examples, but in real-world examples this quickly escalates, leading to the necessity of specifying a plethora of conditions.Writing such conditions is necessary but also notoriously repetitive and tedious. As Kogtenkov et al. [KMV15] so eloquently puts it:

> "It is hard enough to convince programmers to state what their program does; forcing them in addition to specify all that it does not do may be a tough sell."

The tedious, undeserved, manual effort entailed by the specification and verification of frame properties is a manifestation of the frame problem. Though certain conventions and approaches, such as the *implicit frames* approach, for specifying frame properties can alleviate the manual effort imposed, some manifestation of the frame problem will be visible to some extent in the context of any specification language and verification method.

The article proposes a solution to the frame problem based on fully-automatic, static program analyses for inferring the preservation of program invariants. More specifically, we

target the automatic identification of properties that depend only on an input subset that is disjoint from an operation's *frame*, i.e. the state subset it modifies.

1.2 Methodology

To this end, we propose a solution based on static analysis which does not require any additional frame annotations. By detecting the subset on which a property depends and by uncovering the part that is not modified by an operation, as shown in Figure 1, we can automatically discharge proof obligations related to unmodified parts. We employ two different static analyses for this goal.

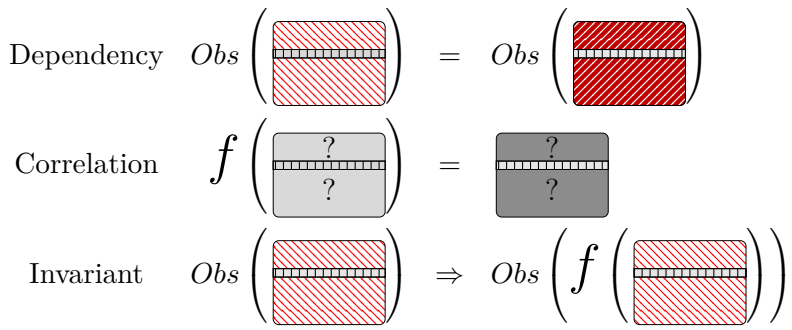

Figure 1: Frame Problem and Solution Strategy

The first analysis of our two-step strategy is a *dependency analysis*, which detects the input subset δ on which the outcome of an operation or of a logical property $\mathscr{L}$ relies. This subset is represented by the grey rectangle with vertical lines in Figure **??**. The second is a *correlation analysis*, meant to detect the subset ξ modified by an operation $\mathscr{O}$, illustrated by the orange rectangles with inclined lines in Figure **??**. By employing these two static analyses, thus detecting δ and ξ automatically, and by subsequently reasoning based on their combined results, we can infer the preservation of the property $\mathscr{L}$ for the post-state of $\mathscr{O}$.

1.3 Application context: Formal verification of systems software

1.4 Organisation of the article

2 The SMART functional specification language

3 Dependency analysis

The dependency analysis delimits the input subset on which the output depends, in the context of an operation with a compound input. We define *dependency* as the observed part of a structured domain and strive to obtain type-sensitive results, distinguishing between the subelements of arrays and algebraic data types and capturing the dependency specific to each. The targeted results are meant to mirror – in terms of dependency – the layered structure of compound data types. Furthermore, the *dependency analysis* must work with *conservative approximations* and it must guarantee that what is marked as not needed is definitely not needed, i.e. it is irrelevant for the obtained output.

In the classification of Hind [Hin01], our dependency analysis is a *flow-sensitive*, *field-sensitive*, *interprocedural* analysis that handles associative arrays, structures and variant data types. Specific dependency results are computed for each of the possible execution scenarios, i.e. for each exit label. Thus, our analysis also shows a form of *path-sensitivity* [Hin01]. However, we favour the term *label-sensitivity* to describe this characteristic, as it seems more appropriate applied to our case and the language we are working with.

Our dependency analysis targets complex transition systems in general, and operating systems and microkernels in particular. These are characterized by states defined by complex compound data structures and by transitions, i.e. state changes, that map an input state to an output state. Automatically proving the preservation of invariants concerning only subelements of the state, i.e. fields, array cells, etc., that have not been altered by a transition in the system would considerably diminish the number of proof obligations. The first step towards achieving this goal consists in automatically detecting dependency summaries and the minimum relevant input information for producing certain outputs.

As mentioned, our analysis targets fine-grained dependency summaries for arrays, structures and variants, expressed at the level of their subelements. For variants, besides capturing the specific dependency on each constructor and its arguments, we argue that additional, relevant information can be computed, regarding the subset of possible constructors at a given program point. This is not dependency information *per se* but it enriches the footprint of a predicate with useful information. Together with the dependency information, this additional information about constructors is meant to answer the same question, namely, what fragments of the input influence the output, from a different, albeit related point of view. Therefore, we are simultaneously performing a *possible-constructors* analysis. This has an impact on the defined abstract dependency type, making it more complex, as we will see in the following section. The *possible-constructors* analysis could be performed separately, as a stand-alone analysis. By performing the two analyses simultaneously, we lose some of the precision that would be attained if the two were performed separately, but we reduce overhead and present relevant information in a unified manner.

Designing the analysis as a tool to be used in the context of interactive program verification, on both code and specifications, has led to specific traits. One of them concerns the treatment of arrays. In contrast to dependence and liveness analyses used for code optimizations [GS90], which require precision for every array cell, we compute dependency information referring to all cells of the array or to all but one cell, for which an exceptional dependency is computed. In practice, a considerable number of relevant properties and operations involving arrays fall into this spectrum.

3.1 Targeted Dependency Information

We briefly present two examples of αSmil predicates thread and start_address, manipulating structures, variants and arrays and describe the dependency information that we are targeting. Both predicates manipulate inputs of type process, and handle values of type thread and memory_region.

The first predicate, thread, having the control flow graph shown in Figure 5 and whose implementation is shown in Figure 4, receives a process p and an index i as inputs. It reads the i-th element in the threads array of the input process p. If this element is active, then the predicate exits with the label true and outputs the corresponding thread ti. Otherwise, it exits with the label None and no output is generated.

```
type memory_region = {
 // Start address
 start : int;
 // Region length
 length : int
}
```

```
type thread = {
 // Identifier
 id : int;
 // Current state
 crt_state : state;
 // Stack
 stack : memory_region
}
```

Figure 2: Example Data Types – Thread and Memory Region

```
type option<A> =
  | None
  | Some (A a)
```

```
type process = {
 // Array of associated threads
 threads : array<option<thread>>;
 // Internal id
 pid : int;
 // Currently running thread
 crt_thread : int;
 // Address space
 adr_space : address_space
}
```

Figure 3: Input Type – Process

```
predicate thread(process p, int i)
 -> [true: thread ti|None|oob]
{{array<option<thread>> th, option<thread> tio}} {
  th := p.threads : [true -> 1];
  tio := th[i] : [true -> 2, false -> 5];
  switch (tio) as [ |ti] : [None -> 4, Some -> 3];
  [true];
  [None];
  [oob]
}
```

Figure 4: Predicate thread – Implementation

Our dependency analysis should be able to distinguish between the different exit labels of the predicate. For the label true for instance, it should detect that only the field threads is read by the predicate, while all others are irrelevant to the result. Furthermore, it should detect that for the threads array of the input p only the i-th element is inspected. Additionally, since we are considering the label true, the i-th element is necessarily an *active* thread, indicated by the constructor Some. The other constructor, None, is *impossible* for this execution scenario. On the contrary, for the exit label None, the constructor Some is impossible. For the exit label oob, nothing but the index i and the "support" or "length" of the associated threads array

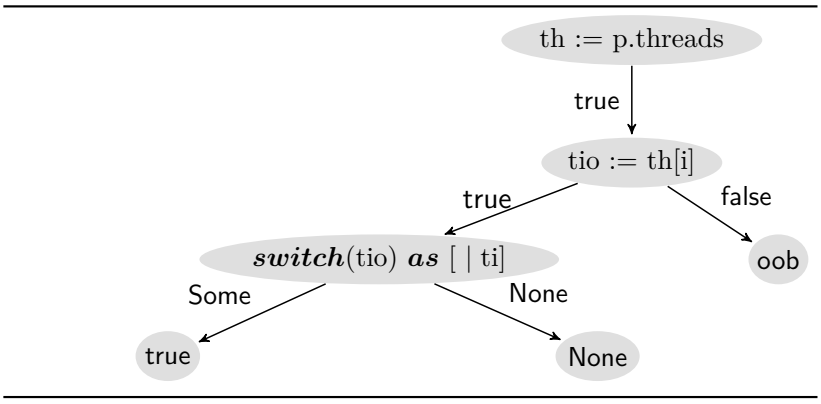

Figure 5: G_{thread} – Control Flow Graph of Predicate thread

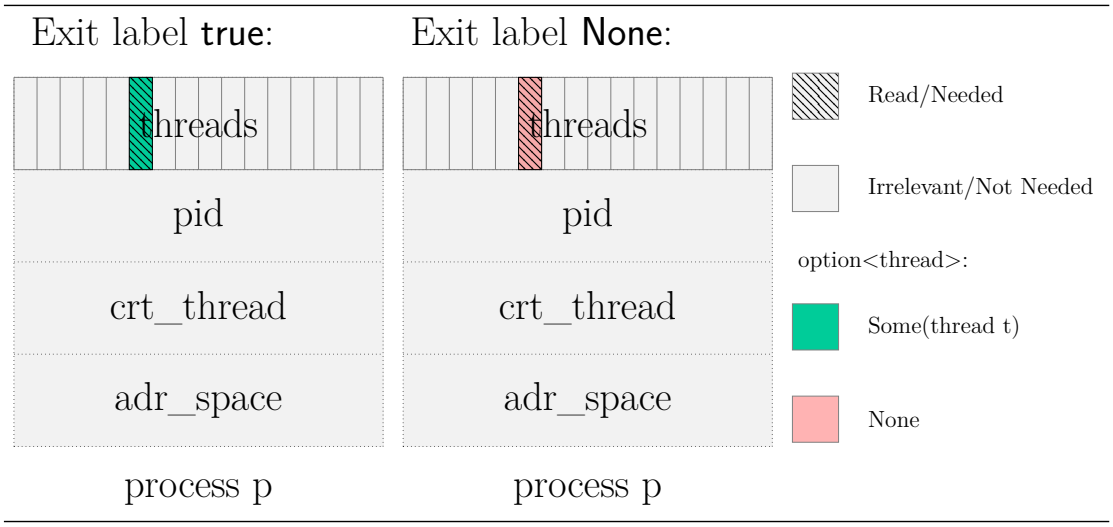

Figure 6: Targeted Dependency Results for Predicate thread

is read. The targeted dependency results for the predicate thread are depicted in Figure 6.

The second predicate, start_address, whose control flow graph is shown in Figure 7, receives a process p and an index j as inputs and finds the start address of the stack corresponding to an active thread. It makes a call to the predicate thread, thus reading the j-th element of the threads array of its input process. If this is an active element, it further accesses the field stack, from which it only reads the start address start. Otherwise, if the element is inactive, the predicate forwards the exit label None of the called predicate thread and generates no output. When given an invalid index i, the predicate exits with label oob. The predicate's implementation is shown in Figure 8.

The dependency information for this predicate should capture the fact that on the true execution scenario, only the field start of the input's j-th associated thread is read. Furthermore, the only possible constructor on this execution path is the Some constructor. On the contrary, for the None execution scenario the only possible constructor is the None constructor. The targeted dependency results for the start_address predicate are depicted in Figure 9. We remark that for the oob execution scenario, only the "support" or "length" of

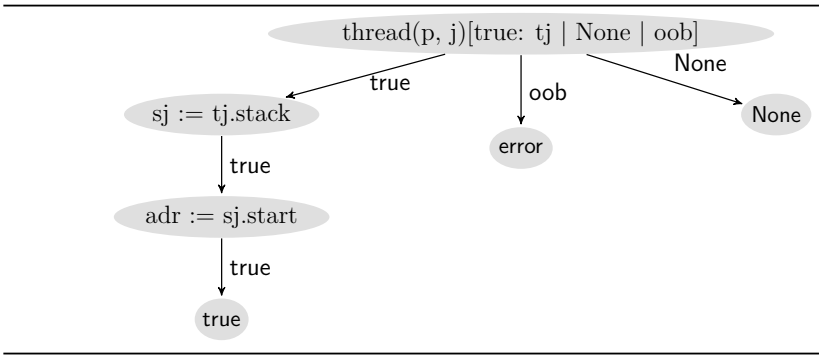

Figure 7: $G_{start\_address}$ – Control Flow Graph of Predicate start_address

```
predicate start_address(process p, int j)
 -> [true: int adr|None]
{{thread tj, memory_region sj}} {
  thread(p, j)[true: tj | None | oob] : [true -> 1,
   None -> 4, oob -> 5];
  sj := tj.stack : [true -> 2];
  adr := sj.start : [true -> 3];
  [true];
  [None];
  [error]
}
```

Figure 8: Predicate start_address – Implementation

the threads array is read.

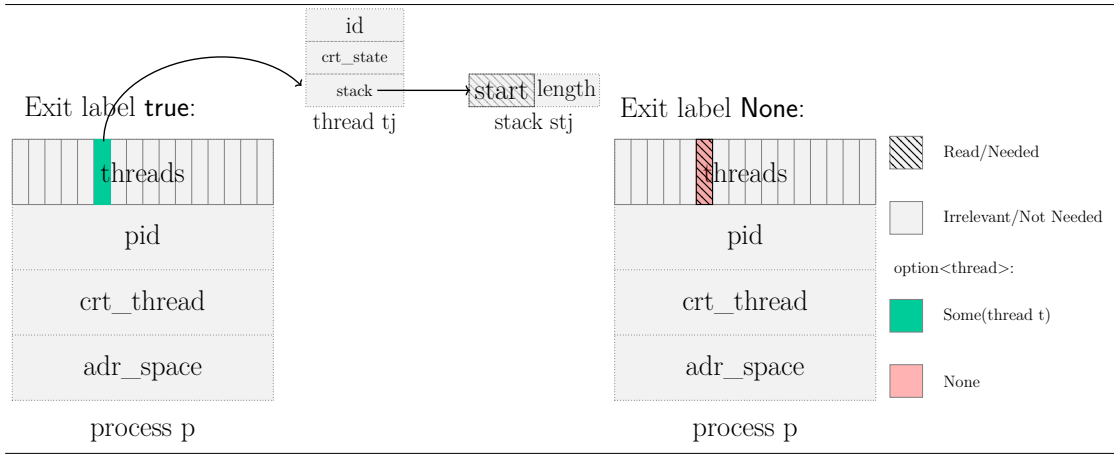

Figure 9: Targeted Dependency Results for Predicate start_address

3.2 The dependency abstract domain

The first step towards inferring expressive, type-sensitive results that capture the dependency specific to each subelement of an algebraic data type or an associative array, is the definition of an *abstract dependency domain* $\mathcal{D}$, that mimics the structure of such data types. The dependency domain $\delta \in \mathcal{D}$, shown below, is defined inductively from the three atomic cases — $\top$, $\oslash$ and $\bot$ — and mirrors the structure of the concrete types.

Definition 1. *Dependency Domain* $\delta \in \mathcal{D}$.

$$
\begin{array}{lll}
\delta := & \mid \top & \text{Everything – } \textit{atomic case} \quad \textit{(i)} \\
& \mid \oslash & \text{Nothing – } \textit{atomic case} \quad \textit{(ii)} \\
& \mid \bot & \text{Impossible – } \textit{atomic case} \quad \textit{(iii)} \\
& \mid \{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\} & f_1, \ldots, f_n \textit{ fields} \quad \textit{(iv)} \\
& \mid [C_1 \mapsto \delta_1; \ldots; C_m \mapsto \delta_m] & C_1, \ldots, C_m \textit{ constructors} \quad \textit{(v)} \\
& \mid \langle \delta \rangle & \textit{(vi)} \\
& \mid \langle \delta_{def} \triangleright i \ : \ \delta_{exc} \rangle & i \textit{ array index} \quad \textit{(vii)}
\end{array}
$$

As reflected by the above definition, the dependency for atomic types is expressed in terms of the domain's atomic cases: $\top$ (least precise), denoting that *everything* is needed and $\oslash$, denoting that *nothing* is needed. The third atomic case $\bot$, denoting *impossible*, is introduced for the *possible constructors* analysis performed simultaneously, and is further explained below.

The dependency of a structure (iv) describes the dependency on each of its fields. For instance, revisiting our thread example from Section 3.1, we could express an over-approximation of the dependency information depicted for the process p in Figure 6 using the following dependency:

$$\{\mathit{threads} \mapsto \top;\ \mathit{pid} \mapsto \oslash;\ \mathit{crt\_thread} \mapsto \oslash;\ \mathit{adr\_space} \mapsto \oslash\}.$$

This captures the fact that all fields except the threads field are irrelevant, i.e. they are not read and *nothing* in their contents is needed. The dependency for the threads field is an

over-approximation and expresses the fact that it is entirely necessary, i.e. *everything* in its value is needed for the result.

For arrays we distinguish between two cases, namely arrays with a general dependency applying to all of the cells given by (vi) and arrays with a general dependency applying to *all but one* exceptional cell, for which a specific dependency is known given by (vii). For instance, for the threads field of the previous example, the following dependency:

$$\langle \oslash \triangleright i : \top \rangle$$

would be a less coarse approximation, capturing the fact that only the i-th element of the associated threads array is needed, while all others are irrelevant.

For variants (v), the dependency is expressed in terms of the dependencies of their constructors, expressed in turn in terms of their arguments' dependencies. Thus, a constructor having a dependency mapped to $\oslash$ is one for which nothing but the *tag* has been read, i.e. its arguments, if any, are irrelevant for the execution. For instance, for the i-th element of the threads array of our previous example, the following dependency:

$$[Some \mapsto \top; None \mapsto \oslash]$$

would be a more precise approximation, when considering the exit label true. It is still an over-approximation as it expresses that both constructors are possible. The argument of the Some constructor is entirely read, while for None only the tag is read.

For variants, we want to take a step further and to also include the information that certain constructors cannot occur for certain execution paths. *Impossible*, the third atomic case — $\bot$ — is introduced for this purpose. As mentioned previously in Section **??**, in order to obtain this additional information, we perform a "possible-constructors" analysis simultaneously, which computes for each execution scenario, the subset of possible constructors for a given value, at a given program point. All constructors that cannot occur on a given execution path are marked as being $\bot$. In contrast, constructors for which *only* the tag is read are marked as $\oslash$. The difference between $\bot$ and $\oslash$ can be illustrated by considering a polymorphic option type *option*<*A*>, having two constructors, $None$ and $Some(A\,val)$, respectively, and a Boolean predicate that pattern matches on an input of this type and returns *false* in the case of $None$ and *true* in the case of $Some$, unconditioned by the value val of its argument. For the *true* execution scenario, the dependency on the $Some$ constructor would be $\oslash$. The tag is read and it is decisive for the outcome, but the value of its argument val is completely irrelevant. The dependency on the $None$ constructor however would be $\bot$: the predicate can exit with label *true* if and only if the input matches against the $Some$ constructor. By distinguishing between these two cases we can not only distinguish the input's subelements that have a direct impact on an operation's output, but additionally, we can also obtain a more detailed footprint that highlights the influence exerted by the input's "shape" on the operation's outcome.

For instance, for the i-th element of the threads array of our previous example, a dependency mapping the constructor None to $\bot$ would be a more precise approximation, when considering the label true. Taking into account all the discussed values, we can express the dependency depicted in Figure 6 for the label true as follows:

$$\left\{\begin{array}{lcl} \mathit{threads} & \mapsto & \langle \oslash \triangleright i : \ [\mathit{Some} \mapsto \top; \mathit{None} \mapsto \bot] \rangle \\ \mathit{pid} & \mapsto & \oslash \\ \mathit{crt\_thread} & \mapsto & \oslash \\ \mathit{adr\_space} & \mapsto & \oslash \end{array}\right\}.$$

We remark that $\top$, $\oslash$ and $\bot$ can apply to any type. For instance, $\top$ can be seen as a placeholder for data that is needed in its entirety. Structure, array or variant dependencies whose subelements are all entirely needed and thus, uniformly mapped to $\top$, are transformed to $\top$. The $\bot$ dependency is a placeholder for data that cannot occur on a certain execution scenario. A whole variant value is impossible if all its constructors are mapped to $\bot$. A whole structure or array is impossible if any of its subelements is impossible.

The $\bot$ atomic value is the lower bound of our domain and hence, the most precise value. The final abstract dependency is a closure of all these combined recursively. To give an intuition of the shape of our dependency lattice, we illustrate below in Figure 10 the Hasse diagram of the order relation between pairs of atomic dependency values. Intuitively, if the two analyses would be performed separately, the upper "diamond" shape would correspond to the dependency analysis, and the lower one to the possible-constructors analysis. The $\oslash$ element would be the lower bound for the dependency domain and the upper bound for the possible-constructors domain. By performing them simultaneously, $\bot$ becomes the domain's lower bound.

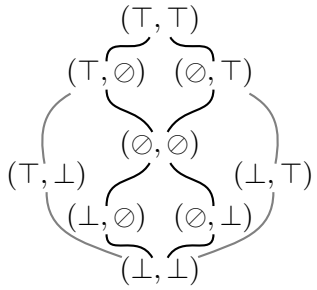

Figure 10: Order Relation on Pairs of Atomic Dependencies

3.3 Dependency flow equations

The intraprocedural dependency analysis keeps dependency information at each point of the control flow graph, for each input, output and local variables.

Definition 2. *Intraprocedural Dependency Domain $\Delta \in \mathscr{D}$. The* intraprocedural *dependency domain $\mathscr{D}$ is defined as*

$$\Delta \ni \mathscr{D} = \mathcal{V} \to \mathcal{D}$$

An element of this domain is a mapping from variables to dependencies.

Our dependency analysis is a *backward* data-flow analysis. For each exit label, it traverses the control flow graph starting with its corresponding exit node and it marks all other exit points as *Unreachable*, since exit labels are mutually exclusive. The intraprocedural domain for the currently analysed label is initialized with its associated output variables mapped to $\top$. Thereby, the analysis starts by making a conservative approximation and by considering that all the input has been observed and the output depends on it entirely. Typically, dependence

analyses are *forward* analyses. However, given our goal to express *label-specific* dependencies as input-output relations and taking into consideration the characteristics of the αSmil language, choosing to design our analysis as a backward data-flow analysis seemed a pertinent choice. In αSmil, outputs are associated to a particular exit label and they are generated if and only if the predicate exits with that particular label. By traversing the control flow graph backwards, we can use this information and consider, starting with the initialisation phase, only the outputs that are relevant for the analysed exit label.

After the initialisation, the analysis then traverses the control flow graph and gradually refines the dependencies until a fixed point is reached. Table **??** summarizes the representation and general equation of the statements. For each statement, the presented data-flow equation operates on the intraprocedural domains of the statement's *successor* nodes. The intraprocedural domain at the *entry point* of the node is obtained by *joining* the contributions of each *outgoing* edge as shown in Figure 11.

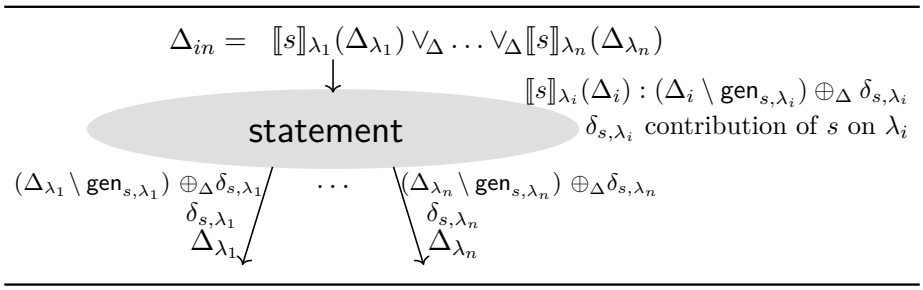

Figure 11: Computation of the Intraprocedural Domain at a Node's Entry Point

Table 1 presents the transfer functions for statements which are not type-specific. For equality tests (1) both of the inputs e_1, e_2 are completely read, whether the test returns true or false. The transfer functions therefore, reduce the domain of the corresponding successor node with a domain consisting of e_1 and e_2 both mapped to $\top$. In the case of assignment (2), the dependency of the written output variable o is forgotten from the successor's intraprocedural domain, thus being mapped to $\oslash$ and forwarded to the input variable e. The transfer function for the ***nop*** operation (3) is simply the identity.

Statement		$[\![s]\!]_{\lambda_i}(\Delta)$	
Equality test	(1)	$[\![e_1 = e_2]\!]_{\mathsf{true}}(\Delta) = \Delta \ \oplus_\Delta \ dep$ $[\![e_1 = e_2]\!]_{\mathsf{false}}(\Delta) = \Delta \ \oplus_\Delta \ dep$	where $dep = \left\{ \begin{array}{ll} e_1 & \mapsto \top \\ e_2 & \mapsto \top \end{array} \right\}$
Assignment	(2)	$[\![o := e]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus o) \ \oplus_\Delta \ \{e \mapsto \Delta(o)\}$	
No Operation	(3)	$[\![\boldsymbol{nop}]\!]_{\mathsf{true}}(\Delta) = \Delta$	

Table 1: Generic Statements – Data-Flow Equations

The data-flow equations given in Table 2 correspond to structure-related statements. For

the equations (4), (5), (6) and (7) we assume that the variable r is of type $\textbf{\textit{struct}}\{f_1 : \tau, \ldots, f_n : \tau\}$ for some fields f_i, $1 \leq i \leq n$. The equation (4) refers to the creation of a structure: each input e_i is read as much as the corresponding field f_i of the structure is read. The destructuring of a structure is handled in (5): each field f_i is needed as much as the corresponding variable o_i is. When accessing the i-th field of a structure r (6), only the field f_i is read, and only as much as the access' result o itself. The equation (7) treats field updates: the variable e_i is read as much as the field f_i is. The structure r is read as much as all the fields other than f_i are read in r'. Finally, the equations given in (8) handle partial structure equality tests, and the transfer functions are the same for the labels true or false: for both compared structures r' and r'', all the fields in the given set $f_1, \ldots, f_k$ are completely read, and only those.

Statement		$[\![s]\!]_{\lambda_i}(\Delta)$
Create	(4)	$[\![r := \{e_1, \ldots, e_n\}]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus r) \oplus_\Delta \bigoplus_{1 \leq i \leq n} \{e_i \mapsto \Delta(r).f_i\}$
Destructure	(5)	$[\![\{o_1, \ldots, o_n\} := r]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus \{o_i \mid o_i \in \bar{o}\}) \oplus_\Delta \{r \mapsto \{f_1 \mapsto \Delta(o_1); \ldots; f_n \mapsto \Delta(o_n)\}\}$
Access field	(6)	$[\![o := r.f_i]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus o) \oplus_\Delta \{r \mapsto \{f_1 \mapsto \oslash; \ldots; f_i \mapsto \Delta(o); \ldots; f_n \mapsto \oslash\}\}$
Update field	(7)	$[\![r' := \{r \textbf{ with } f_i = e\}]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus r') \oplus_\Delta \left\{ \begin{array}{lll} e_i & \mapsto & \Delta(r').f_i \\ r & \mapsto & \{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\} \end{array} \right\}$ where $\delta_j = \left\{ \begin{array}{ll} \Delta(r').f_j & \text{if } j \neq i \\ \oslash & \text{otherwise} \end{array} \right.$
Equality	(8)	$[\![r' = \langle f_1, \ldots, f_k \rangle r'']\!]_{\mathsf{true}}(\Delta) = \Delta \oplus_\Delta d$ where $d = \left\{ \begin{array}{lll} r' & \mapsto & \{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\} \\ r'' & \mapsto & \{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\} \end{array} \right\}$ $[\![r' = \langle f_1, \ldots, f_k \rangle r'']\!]_{\mathsf{false}}(\Delta) = \Delta \oplus_\Delta d$ and $\delta_i = \left\{ \begin{array}{ll} \top & \text{if } f_i \in \{f_1, \ldots, f_k\} \\ \oslash & \text{otherwise} \end{array} \right.$

Table 2: Structure-Related Statements – Data-Flow Equations

3.4 Intraprocedural Dependency Analysis Illustrated

To better illustrate our analysis at an intraprocedural level, we exemplify the mechanism behind it, step by step, on the predicate thread, discussed in Section 3.1. We consider the true execution scenario, apply our dependency analysis and compare the actual obtained results with the targeted ones depicted in Figure 6.

Since a predicate can only exit with one label at a time and we are considering the true label, we can map the nodes None and oob to *Unreachable*, as shown in Figure 12. This is an advantage of backwards analyses. For true, we make a pessimistic assumption and map the output ti to $\top$, considering that control on the output is external and hence, out of our reach, and that ti will be entirely needed by a potential caller. Going further up the control flow graph, we analyse the *variant switch*.

In order to compute the dependency for the node corresponding to the variant switch, we apply the data-flow equation, given by (10) in Table **??**. Since we are analysing the true case, we know that all other constructors (only the constructor None in this case) are locally

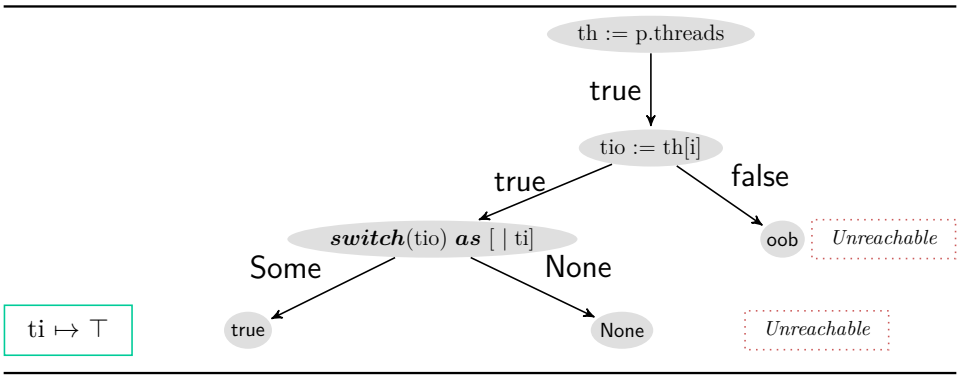

Figure 12: Analysing Predicate thread – Initialisation

impossible. Thus, we map it to $\bot$. We continue by forgetting the dependency information we knew about the output ti. Since its value is needed only in as much as the result of the switch on the corresponding edge is needed, we forward it to the part corresponding to the Some constructor. This is summarized below:

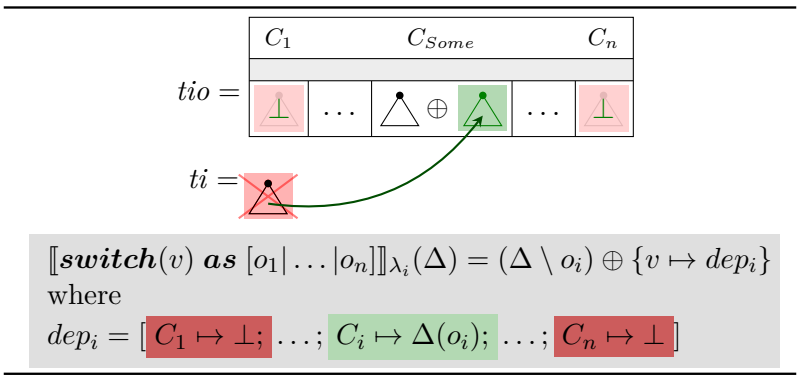

Figure 13: Applying the Variant Switch Equation

Taking all this into account, for the node corresponding to the variant switch, we obtain the dependency shown in Figure 14. For the output ti, we depend entirely on the Some constructor of the node's input variant tio, while the constructor None is impossible.

Making a step further up the graph, we access the cell i of the array th and apply the equation (12) given in Table **??**. We begin by forgetting the dependency for the output tio, since this is written. Since we only access the element i, we map all other cells to *Nothing*, i.e. $\oslash$. To the dependency corresponding to the i-th cell, we forward the dependency we knew about tio, since we depend on it to the extent to which the result of the access is needed.

We thus obtain a dependency stating that we depend only on the i-th cell of the array th, for which only the constructor Some is possible and entirely needed. The cell's index i is entirely needed as well. The applied equation is shown in Figure 15 (since i is an input, we use the first case of the equation) and the obtained results are shown in Figure 16.

As a last step, we access the field threads of the input process p and apply the equation (6) given in Table 2 and illustrated in Figure 17. As before, we forget the information for th, the access result. We map all other fields to $\oslash$ and we forward the dependency of the variable th to the dependency part of the field threads.

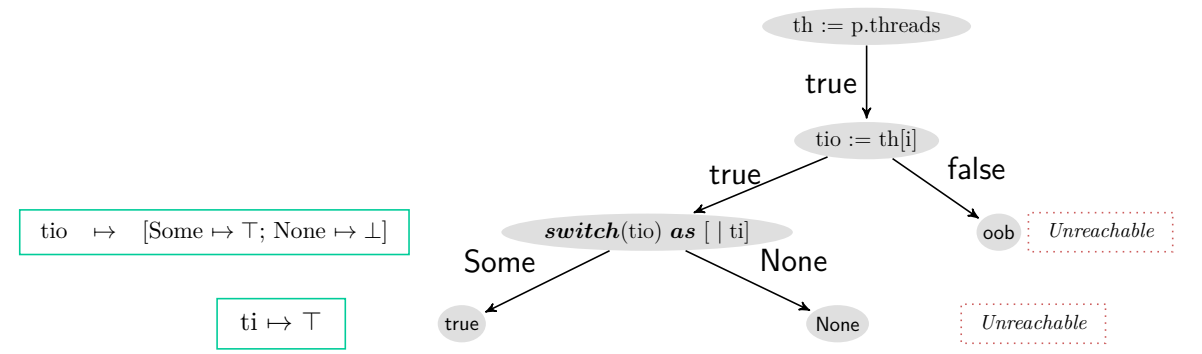

Figure 14: Analysing Predicate thread – Variant Switch

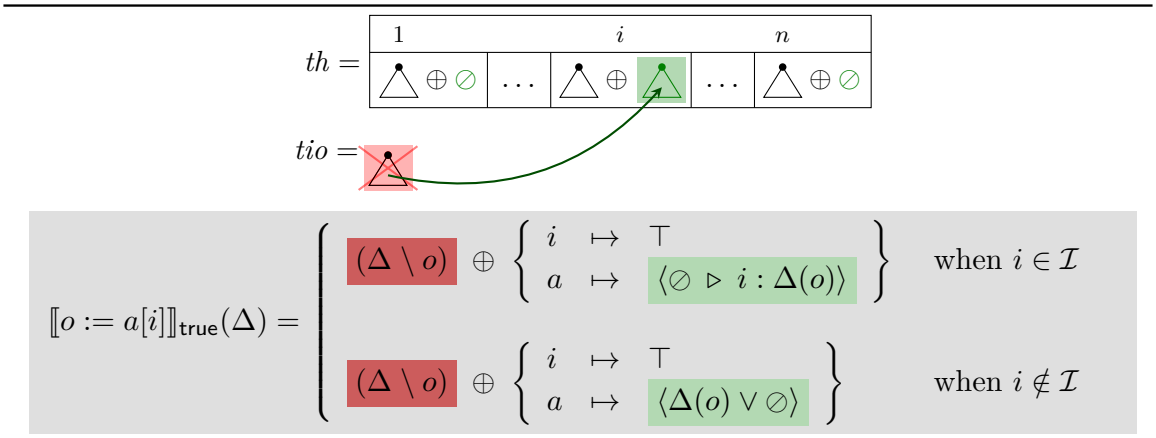

Figure 15: Applying the Array Access Equation

We thus obtain the dependency result shown in Figure 18. This states that for the label true, the output ti depends only on the i-th cell of the field threads of the input process p, for which it depends entirely on the Some constructor. Before returning the predicate's final results, the analysis filters out any dependency information referring to local variables and verifies that the invariant imposed on dependency information related to arrays holds. Since the results refer only to the inputs p and i and the index of the exceptional computed dependency is an input, the invariant holds and the final result can be retrieved. The final dependency results obtained for the thread predicate on the exit label true are identical to the ones that we were targeting and that were depicted in Figure 6. For readability considerations, for structures such as the input process p, we omit dependencies on fields mapped to $\oslash$. We maintain this convention throughout the rest of this chapter, and thus any field of a structure that is omitted from a dependency summary should be interpreted as being mapped to $\oslash$, i.e. *nothing*.

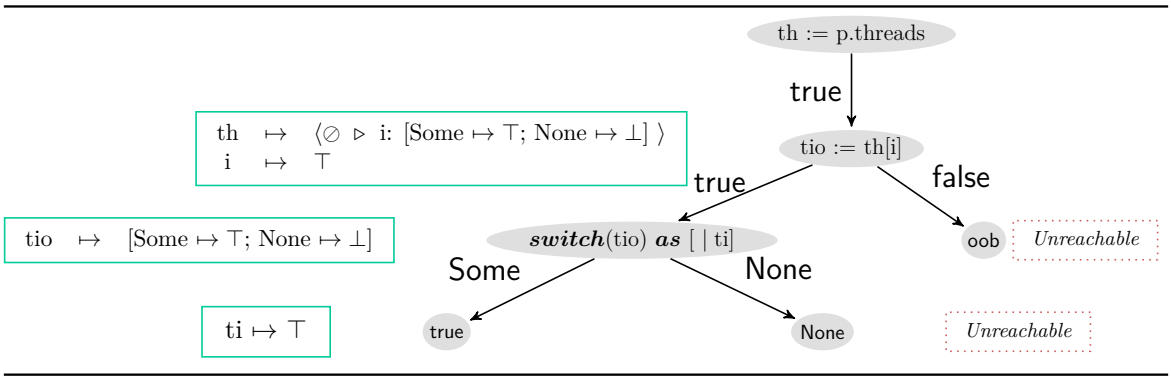

Figure 16: Analysing Predicate thread – Array Access

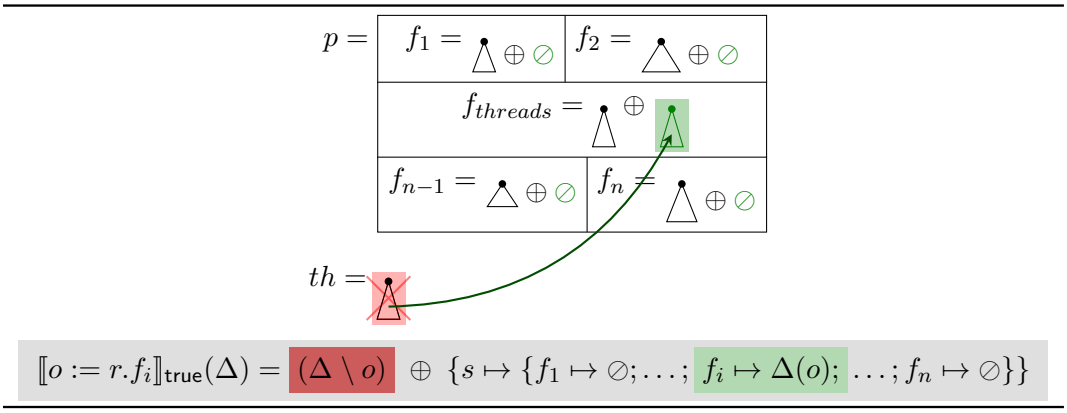

Figure 17: Applying the Field Access Equation

3.5 An inter-procedural extension

4 Correlation analysis

4.1 Introductory example

4.2 Correlations as partial equivalence relations

4.3 Correlation analysis

5 Experimental evaluation

6 Related work

7 Conclusions

References

[BMR93] Alexander Borgida, John Mylopoulos, and Raymond Reiter. "... And nothing else changes": The frame problem in procedure specifications. In *Proceedings of*

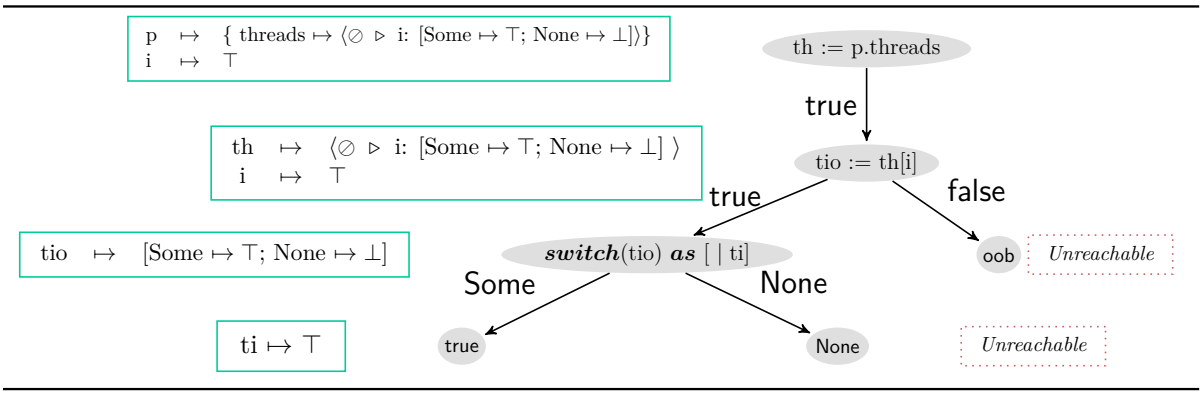

Figure 18: Analysing Predicate thread – Field Access

the 15th International Conference on Software Engineering, Baltimore, Maryland, USA, May 17-21, 1993., pages 303–314, 1993.

[GS90] Thomas R. Gross and Peter Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Softw., Pract. Exper.*, 20(2):133–155, 1990.

[Hin01] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, pages 54–61, 2001.

[KMV15] Alexander Kogtenkov, Bertrand Meyer, and Sergey Velder. Alias calculus, change calculus and frame inference. *Sci. Comput. Program.*, 97(P1):163–172, January 2015.

[MH69] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*. Edinburgh University Press, 1969.