Generating Precise and Concise Procedure Summaries

Greta Yorsh *
Tel Aviv University, Israel
gretay@post.tau.ac.il

Eran Yahav
IBM T.J. Watson Research Center, USA
eyahav@us.ibm.com

Satish Chandra
IBM T.J. Watson Research Center, USA
satishchandra@us.ibm.com

Abstract

We present a framework for generating procedure summaries that are *precise* — applying the summary in a given context yields the same result as re-analyzing the procedure in that context, and *concise* — the summary exploits the commonalities in the ways the procedure manipulates abstract values, and does not contain superfluous context information.

The use of a precise and concise procedure summary in modular analyses provides a way to capture infinitely many possible contexts in a finite way; in interprocedural analyses, it provides a compact representation of an explicit input-output summary table without loss of precision.

We define a class of abstract domains and transformers for which precise and concise summaries can be efficiently generated using our framework. Our framework is rich enough to encode a wide range of problems, including all IFDS and IDE problems. In addition, we show how the framework is instantiated to provide novel solutions to two hard problems: modular linear constant propagation and modular typestate verification, both in the presence of aliasing. We implemented a prototype of our framework that computes summaries for the typestate domain, and report on preliminary experimental results.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Theory of Computation*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Reliability, Languages, Algorithms

Keywords summarization, composition, relational analysis, symbolic summary, typestate verification, aliasing, dataflow analysis, micro-transformers

1. Introduction

The problem of automatically computing procedure summaries is a fundamental problem in program analysis. Most of the existing general-purpose approaches to interprocedural analysis (Sharir and Pnueli 1981; Cousot and Cousot 1978; Reps et al. 1995; Sagiv et al. 1996b) compute tabulation-based procedure summaries: summaries that are represented as an explicit tabulation of the relation between abstract values at the entry of a procedure, and the corresponding values at its exit. In particular, the commonly used (e.g,. in Dor et al. (2004); Das et al. (2002); Qadeer and Wu (2004); Jhala and Majumdar (2007); Fink et al. (2006); Rinetzky et al. (2005)) framework for solving IFDS problems (Reps et al. 1995) performs explicit tabulation for distributive domains.

However, an explicit input-output table is just one possible representation of the procedure's abstract effect. Indeed, in their seminal works, Cousot and Cousot (1978) with the "relational approach" and Sharir and Pnueli (1981) with the "functional approach" represent a procedure summary as a function from input abstract values to output abstract values. The functional representation of a summary does not necessarily enumerate input abstract values; it can also describe how classes of input abstract values are transformed by the procedure, by referring "symbolically" to abstract values. The difficulty—already acknowledged in (Sharir and Pnueli 1981)—is in generating such a symbolic summary.

Cousot and Cousot (2002) introduced *symbolic relational separate analysis*, a conceptual framework for modular analysis. Our contribution is in identifying a rich class of problems for which it is feasible to generate procedure summaries that are symbolic, and thus, solve these problems using modular analysis. There has been relatively little previous work on symbolic summarization techniques in this setting. Chatterjee et al. (1999) introduced a modular points-to analysis. Our work draws inspiration from their work in identifying relevant behaviors of procedures. Other efforts in this direction include (Cheng and Hwu 2000; Whaley and Rinard 1999; Gulwani and Tiwari 2007; Xie and Aiken 2005; Ball et al. 2005), but we are not aware of a more general framework that computes symbolic procedure summaries for a wider class of problems.

In this paper, we present a new framework for generating symbolic procedure summaries for a rich class of abstract domains and transformers over those domains. Given a procedure and a program-independent description of an abstract domain and its transformers, our framework automatically computes a symbolic summary of that procedure. The summary is applicable in any calling context.

Our framework derives procedure-level transformers by symbolic composition of statement-level transformers, and represents the result of composition in the same form as a statement-level transformer. The procedure-level transformers—or the procedure summaries—computed by our framework have the following properties:

- *Precise:* applying the summary in a given context yields the same result as re-analyzing the procedure in that context; no information is lost during creation of the summary.
- *Concise:* the summary exploits the commonalities in the ways the procedure manipulates different abstract values, and does not contain superfluous context information.
- *Efficient:* applying the summary in a given context is more efficient than re-analyzing the procedure in that context.

* This research was supported in part by an Eshkol Fellowship.

POPL'08, January 7–12, 2008, San Francisco, California, USA.

The procedure itself is a trivial summary of its effect, but of course it is not "efficient". We are interested in a summary that captures the composite effect of a procedure, such that applying the summary does not use statement-level abstract transformers.

The motivation for generating concise summaries is two-fold. First, a concise procedure summary provides a way to capture infinitely many possible contexts in a finite way, and hence, can be used in modular analyses. For example, a concise summary of a library can be used when analyzing any client of that library. Moreover, a summary of a library can be generated before a client code is written, because a summary can refer to possibly unknown calling context symbolically. Second, for interprocedural analyses, a concise summary provides a compact representation of an explicit input-output summary table, without loss of precision. A concise summary, that ignores irrelevant context information, is potentially more compact than, e.g., a shared representation of an explicit input/output table using BDDs.

We have identified sufficient conditions on the structure of abstract domains and their transformers that guarantee that our framework can automatically compute concise and precise summaries. The key idea is that the transformers we support make only finitely many distinctions over input values, and each distinct class of values behaves uniformly. Rather sophisticated abstract domains and transformers can be encoded in this restricted form. Not only can our framework handle the well-known IFDS (Reps et al. 1995) and IDE (Sagiv et al. 1996a) problems, it can also handle problems such as modular linear constant propagation in the presence of aliasing, and modular typestate verification in the presence of aliasing (Fink et al. 2006). We have created a prototype implementation of modular typestate verification based on this framework and have run the analysis on a number of realistic, albeit small programs.

1.1 Overview

Our Approach to Generating Summaries We derive procedure summaries by symbolic composition of statement-level transformers, as follows. For basic statements, the transformers are given as input to our framework. For call statements, the transformers are the summaries of the callees (after replacing formal parameters with actual arguments). For loop statements, the transformer is computed by iterated composition of the transformer for the loop body, until the composite transformer reaches a fixpoint. Recursion and callbacks can be handled similarly to loops. The procedure summary is simply the composite transformer for the procedure body.

A sufficient condition for this approach is that the language of statement-level transformers (summaries) is (a) *closed under composition*, i.e., the result of composition is again represented in the same form as a statement-level transformer, and (b) *finite*, to guarantee that iterated composition terminates.

Challenge of Composition When summaries are represented as explicit relational tables, composition of summaries is easy—albeit possibly inefficient—to compute, as it corresponds intuitively to the relational join of the tables. By contrast, for symbolic summaries, composition of two (symbolic) transformers might not be expressible in the same form as transformers, or finding the representation of transformers might be infeasible to compute.

Given a pair of transformers tr^{12} and tr^{23}, the goal of composition is to return a transformer tr^{13} that precisely captures the composed effect of tr^{12} and tr^{23}. Intuitively, one can think of these transformers as relating values in three domains: (A1), (A2), and (A3), where tr^{12} transforms values between (A1) and (A2), and tr^{23} transforms values between (A2) and (A3). The challenge of creating a precise summary is to relate the values of (A1) and (A3) without explicit mention of the values in (A2). This is shown schematically in Fig. 1 (a). The key to composition is therefore to represent

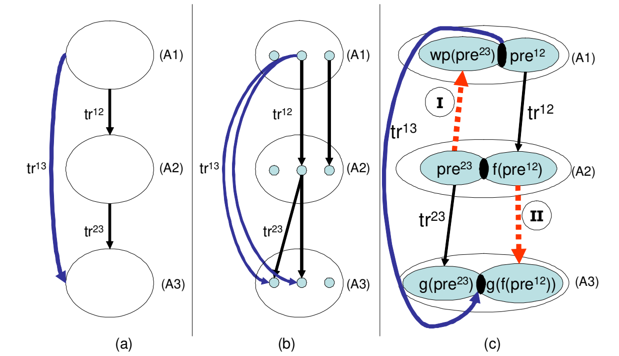


Figure 1. Schematic view of composition with (a) generic transformers (b) finite domain with distributive transformers (c) conditional micro-transformers.

the restriction imposed on values in (A2) due to the composition of tr^{12} and tr^{23} by expressing them as restrictions in (A1) and in (A3).

In general, avoiding restrictions on values in (A2) is a hard problem closely related to that of existential quantifier elimination. (The transformers can be viewed as a symbolic representation of relations, and their composition corresponds to conjunction and existential quantification.) However, when the domain and the transformers are of a certain restricted structure, composition becomes feasible. For example, when the domain is finite and the transformers are distributive, e.g., Fig. 1 (b), the composed effect can be computed via graph reachability (Reps et al. 1995).

Exploiting Structure for Composition The key insight used in this paper is that by exposing the underlying structure of an abstract domain and transformers we can express the domain as an aggregate of simpler domains, for which it is feasible to compute the composition of transformers. That is, the composition of transformers for the aggregate domain can be broken into smaller composition problems of micro-transformers that apply to the sub-domains, and so on. We do allow limited interaction between sub-domains.

Micro-transformers in our framework can be expressed as operating on classes of values in a sub-domain rather than working on individual elements. We call micro-transformers written in this way *conditional micro-transformers*. A conditional micro-transformer consists of several cases, each of which has a *precondition* that defines a distinct class of input values, and a *postcondition* that defines how these values are transformed. The idea is to expose enough structure of the aggregate domain such that all values in the same class are transformed uniformly, and thus can be described by simple postconditions.

Just as statement-level transformers that we support make only finitely many distinctions over input values, the procedure-level transformer—or the summary—distinguishes between only a finite number of classes of these values. This reduces the information that needs to be encoded in a summary from a potentially infinite context into a finite one.

Composition of Conditional Micro-Transformers Our composition algorithm leverages the particular structure of transformers we enforce by considering each class of values separately. Fig. 1 (c) shows how our algorithm composes a single case of tr^{12} with a single case of tr^{23}. The composition is performed in two stages:

(I) expressing the precondition of tr^{23} (shown as pre^{23} in A2) in the domain (A1). This is done by computing the weakest precondition of pre^{23} under tr^{12} as shown by the dotted arrow marked (I). The set $wp(pre^{23})$ expresses the restriction on values imposed by pre^{23} in terms of the domain (A1).

(II) expressing the postcondition of tr^{12} under the effect of tr^{23} in the domain (A3). Technically, this step is performed by substitution and is shown by the dotted arrow marked (II). The set $g(f(pre^{12}))$ expresses the restriction on values imposed by $f(pre^{12})$ in terms of the domain (A3).

The intersection of $wp(pre^{23})$ and pre^{12} expresses the precondition of the composite transformer tr^{13} in terms of the domain (A1). The intersection of $g(pre^{23})$ and $g(f(pre^{12})$ expresses the postcondition of tr^{13} in terms of the domain (A3). Therefore, the result of composition of conditional micro-transformers can also be expressed as a conditional micro-transformer.

We give an elaborate description of both steps in Section 4.

1.2 Contributions

The main contributions of this paper are:

- We introduce a formal framework for generating symbolic summaries for a class of abstract domains and transformers, and show that for those domains and transformers, the framework produces summaries that are concise, precise and efficient.
- We show how the framework is instantiated to provide novel solutions to two hard problems: (i) modular typestate verification in the presence of aliasing, and (ii) modular linear constant propagation in the presence of aliasing. We also briefly describe how the framework can be used to solve a few other problems.
- We present a prototype implementation that computes procedure summaries for the typestate domain. Our experiments show that our implementation successfully generates summaries for procedures of real (albeit small) programs.

Outline In Sec. 2, we illustrate our method by a simple example of composition, and introduce our running example. In Sec. 3, we define the class of abstract domains and transformers handled by our framework. In Sec. 4, we present the composition algorithm and outline its properties. In Sec. 5, we describe how the composition algorithm is put to use in a general framework for generating procedure summaries and show how to instantiate the framework for several applications. In Sec. 6, we report on preliminary experimental results obtained using our prototype for generating summaries for typestate domain. Finally, in Sec. 7, we survey related work.

2. Illustrative Examples

In this section, we present our main ideas at a semi-technical level using examples. We present two examples: the first one illustrates our entire approach on a simple analysis problem, and the other one introduces the important but complex analysis problem of typestate verification, and is developed gradually throughout the paper.

2.1 Nullness of References

The analysis problem modeled in this example tracks whether the value of a given reference must be null at runtime. Although meant for expository purposes, it may also serve as part of an analysis targeted to statically identify potential null-dereferences.

Domain We use an abstract domain in which the abstract value is a set of access paths, such that all elements in the set must have the value null. For a given program, an example value in this abstract domain could be $\{p.f, q.\epsilon\}$, meaning that $p.f$ and q are both null. Here, $p.f$ is an access path of length 1, and $q.\epsilon$ is an access path of length 0. We write q instead of $q.\epsilon$ when no confusion is likely. To guarantee termination of the nullness analysis, the length of the access paths represented in the nullness domain is bounded. To simplify the presentation, we bound the length to be at most 1.

```
class DataReader {
 private FileComp f;
 void readData(FileComp p) {
  init(p);
  process();
 }
 void setComponent(FileComp p) { this.f = p; }
 FileComp getComponent() { return this.f; }
 void init(FileComp p) {
  this.f = p;
  f.open();
 }
 void process() {
  FileComp q = this.f;
  while (?)
   q.read();
  q.close();
 }
}
```

Figure 2. A program using the type FileComp.

Let AP denote the set of all access paths of length at most 1; the abstract domain for this example is therefore $\mathcal{P}(AP)$. The domain AP is defined as $\mathit{VarId} \times (\mathit{FieldId} \cup \{\epsilon\})$, where VarId and $\mathit{FieldId}$ are parameters of the domain, that denote variable names and field names, respectively: unless they are instantiated for a specific program, we do not know which specific program variables and fields they contain. Since AP relies on these parametric domains, it too is a parametric domain.

Transformers Consider the `setComponent` procedure in the example program of Fig. 2. Denoting the set of access paths of the abstract value by $M \in \mathcal{P}(AP)$, the concise summary (transformer) for this procedure is:

$$tr_{\mathcal{P}(AP)}(M) = \bigcup_{d \in M} tr_{AP}(d)$$

where $tr_{AP}(d)$ is a "micro-transformer" that describes how an element of the set is individually manipulated by the transformer, and maps d to a set of resulting elements. This illustrates the general pattern of expressing transformers on abstract values in some domain A in terms of micro-transformers on abstract values in subdomains that comprise A. As we shall see later, other examples use additional domain constructors.

The micro-transformer $tr_{AP}(d)$ for `setComponent` is:

$$tr_{AP}(d) = \begin{cases} \mathtt{this.f}, \mathtt{p} & d = \mathtt{p} \\ d & d \neq \mathtt{this.f} \wedge d \neq \mathtt{p} \end{cases}$$

The format of this transformer is a series of preconditions (shown on right) and the corresponding output (shown on left). For clarity, we have written the contents of the returned sets (the left-hand-side) without the enclosing set brackets, and we have omitted cases in which the resulting set is empty.

The intuitive meaning of the procedure summary (described by the micro-transformer) is that if the access path $\mathtt{p}$ is known to have a null value before the procedure, then it has null value after the procedure, and in addition, the access path $\mathtt{this.f}$ must also have a null value; for all other access paths d such that ($d \neq \mathtt{this.f} \wedge d \neq \mathtt{p}$) before the transformer, their membership in M remains unchanged.

Composition Now consider an additional procedure `nop` added to the class `DataReader` as follows:

```
void nop() {
  FileComp t = getComponent();
  setComponent(t);
}
```

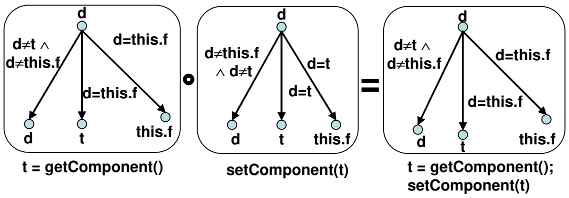


Figure 3. Micro-transformers for nullness of references.

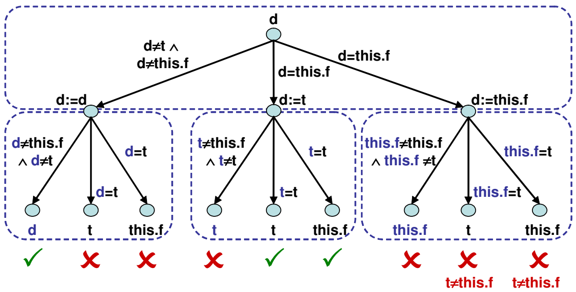


Figure 4. Composition of `t=getComponent();setComponent(t)`.

The micro-transformer for the call `t=getComponent()` is:

$$tr_{AP}^{12}(d) = \begin{cases} \texttt{t}, \texttt{this.f} & d = \texttt{this.f} \\ d & d \neq \texttt{t} \wedge d \neq \texttt{this.f} \end{cases}$$

The micro-transformer for call `setComponent(t)` is:

$$tr_{AP}^{23}(d) = \begin{cases} \texttt{this.f}, \texttt{t} & d = \texttt{t} \\ d & d \neq \texttt{this.f} \wedge d \neq \texttt{t} \end{cases}$$

The summary of the entire code sequence inside `nop` is:

$$tr_{AP}^{13}(d) = \begin{cases} \texttt{t}, \texttt{this.f} & d = \texttt{this.f} \\ d & d \neq \texttt{t} \wedge d \neq \texttt{this.f} \end{cases}$$

The summary of the whole procedure (after the effect on the local variable `t` is removed) amounts to the identity function.

Each of the conditional micro-transformers in the composition example is shown in Fig. 3 in the form of a one-level tree: the root of the tree represents an input value d, the leaves of the tree correspond to different output values, whose preconditions appear on the edges leading to the leaves. In the micro-transformer for `t=getComponent()`, the input value is a single access path d. The edges leading to the leaves `t` and `this.f` are labeled with the precondition $d = \texttt{this.f}$, representing the first case of the transformer. The edge leading to the leaf d is labeled with the precondition $d \neq \texttt{t} \wedge d \neq \texttt{this.f}$, representing the second case of the transformer. Note that every leaf corresponds to a term of the postcondition, and the edge leading to it is labeled with the precondition that is guarding that term.

We illustrate how our composition algorithm generates a composite micro-transformer for `t=getComponent();setComponent(t)` using the conditional micro-transformers for `t=getComponent()` and `setComponent(t)`, denoted by tr_{AP}^{12} and tr_{AP}^{23}, respectively. For each term $t(d)$ in a postcondition of $tr_{AP}^{12}(d)$, we make a copy of $tr_{AP}^{23}(d)$, in which we replace d with $t(d)$. Fig. 4 depicts this operation. The top dotted rectangle in the figure depicts the transformer for `t=getComponent()`. The three dotted rectangles at the bottom depict the copies of the transformer for `setComponent(t)` in which d was replaced with $t(d)$ as described above. We label the root of each tree on the second level with the corresponding assignment of $t(d)$ to d. For example, the rightmost tree corresponds to a case in which $t(d)$ is `this.f`.

The two-level tree in Fig. 4 can be used to obtain the result of composition. A conjunction of preconditions on the path from the root to a leaf defines a new precondition. We label the leaves for which the new preconditions are consistent with a checkmark. For example, the new precondition for the rightmost branch of the tree is incosistent because the access paths `t` and `this.f` are distinct. [1] We simplify the result by removing inconsistent leaves, and replacing paths from the root to the remaining leaves with edges. The result of composition is shown as the rightmost transformer in Fig. 3.

Other analysis problems present much more challenging tasks of simplification and subsequent normalization—sometimes requiring domain-specific axioms—but conceptually the idea is the same.

Properties The summaries that we compute for `getComponent()` and `setComponent()` individually, as well as for their composition, have all the desired properties: they are concise, precise, and efficient. The properties depend crucially on the strength of the composition algorithm: for example, a sound but imprecise algorithm could have lost the fact that the effect of `nop()` is the identity function, and in particular, it might fail to establish the fact that `this.f` must have a null value after the execution of the code sequence if it initially had a null value.

To represent a complete summary of `setComponent` by explicit tabulation, one would need to describe input-output pairs for the possibly large number of elements of $\mathcal{P}(AP)$, that would depend on the actual program variables and field names outside of this procedure. In addition, even a partial summary table for this procedure might contain redundant information.

2.2 Typestate Verification in the Presence of Aliasing

We introduce typestate verification as an example of a more complicated domain. A typestate specification (Strom and Yemini 1986) for a type constrains the sequences of procedure calls that can be invoked on an object of that type. In the example program of Fig. 2, the typestate specification for the type `FileComp` contains the following non-error transitions: *init* $\xrightarrow{\texttt{open()}}$ *open*, *open* $\xrightarrow{\texttt{read()}}$ *open*, and *open* $\xrightarrow{\texttt{close()}}$ *closed*; all other transitions lead to a designated error state.

Formally, a typestate specification is a deterministic finite-state automaton with alphabet Σ, states $\mathcal{Q}$, initial state $init \in \mathcal{Q}$, final state $err \in \mathcal{Q}$, also called "error state", and transition function $\delta_\sigma : \mathcal{Q} \to \mathcal{Q}$ for each $\sigma \in \Sigma$.

Given a program and a typestate specification, the purpose of typestate verification is to ensure that in all possible executions of the program, no object can enter an error state. Typestate verification is a well studied problem (e.g., (Foster et al. 2002; Das et al. 2002; Dor et al. 2004; Fink et al. 2006; DeLine and Fähndrich 2004; DeLine and Fähndrich 2002; Field et al. 2003)), but most existing solution provide a limited treatment of aliasing, or limited scalability. A modular implementation of typestate analysis is therefore of significant practical interest.

We focus on one abstraction that forms the core of the typestate verification system of (Fink et al. 2006), which carried out typestate verification as a non-modular, whole-program interprocedural analysis using this abstraction. In this abstract domain, an abstract value is a set of dataflow facts. Each dataflow fact refers to a single allocation site and combines information about the typestate of an object allocated at that site, and pointer information related to the same object. In particular, a dataflow fact is of the form $\langle a, s, M \rangle$ where a is an allocation site, s is a typestate from

[1] Here, the equality of access paths is a syntactic, as opposed to checking aliasing between access paths.

$\{init, open, closed\}$, and M is a set of access paths (similar to those used in the null-dereference example) that must point to the tracked allocation site a. Abstract transformers for this domain are distributive and, therefore, can be represented pointwise for the set of dataflow facts. See (Fink et al. 2006) for the rationale behind this abstraction; suffice it to say that tracking must alias access paths is crucial for getting a low false positive rate for this verification problem.

Subsequent sections of the paper show how we model this composite domain (Sec. 3.1), what are the transformers (Sec. 3.2), and the details of the composition algorithm (Sec. 4). In particular, we will see how the composition algorithm obtains the typestate summary of `readData` from those of `init` and `process`: our composition algorithm is powerful enough to establish that each `FileComp` always goes through correct typestate transitions.

3. Parametric Domains and Transformers

In this section, we describe the abstract domains and transformers supported by our framework. First, we restrict attention to parametric abstract domains that are built using simple domain constructors. Then, we restrict the abstract transformers to those defined via micro-transformers operating on the components of an abstract value. In particular, in Sec. 3.2, we define the notion of conditional micro-transformers which is the key to concise representation.

Expressing transformers using micro-transformers allows us to leverage their structure, and implement a composition algorithm for generating precise and concise summaries, as described in Sec. 4.

3.1 Domain Constructors

Abstract interpretation (Cousot and Cousot 1977) computes, for each program point, an abstract value that overapproximates the sets of concrete program states that actually arise at that program point. Abstract values are drawn from an abstract domain that usually depends on the program under analysis. For example, for a program with variables `x`, `y`, and `z`, an abstract value for the abstract domain of constant propagation is a mapping $env\colon \{x,y,z\} \to (\mathbb{Z} \cup \{\top\})$. Abstract values may also refer to elements of an explicitly defined, program-independent, potentially infinite set of *atomic values*, such as the states $\mathcal{Q}$ of a typestate automaton, or the set of all naturals $\mathbb{N}$.

A *parametric abstract domain* provides a program-independent description of abstract values in the domain, using domain parameters. A *domain parameter* is a symbolic place-holder for program-specific values that are to be bound at a later time. For example, a parametric abstract domain for constant propagation will introduce a domain parameter *VarId* as a place holder for the set of program variable identifiers.

A parametric abstract domain can be instantiated for a specific program by binding the domain parameters to program-specific values (e.g., set of program variables). We refer to an instantiated parametric domain as a *specific abstract domain*.

In this paper, we consider abstract domains defined using the domain constructors of the following definition.

DEFINITION 3.1 (Domain Constructors). *An abstract domain A can be constructed using the following (non-recursive) domain constructors:*

$A(\chi)$ ::=	$\mathcal{P}(A_1(\chi_1))$	*powerset*
$\mid$	$\mathcal{P}(A_1(\chi_1) \times A_2(\chi_2))$ *with Prop*	*binary relation (with properties)*
$\mid$	$A_1(\chi_1) \times \ldots \times A_k(\chi_k)$ *with IR*	*(reduced) product*
$\mid$	$A_1(\chi_1) \cup A_2(\chi_2)$	*union*
$\mid$	Val_j	*set of atomic values*
$\mid$	X_i	*parameter*

where $\chi = \{X_1, \ldots, X_n\}$ is a set of parameters; X_i is one of the parameters, for $1 \leq i \leq n$; $\chi_1, \ldots, \chi_k$ are subsets of χ; $Val_1, \ldots, Val_m$ are sets of atomic values; $1 \leq j \leq m$.

For a binary relation, we support the following standard relational properties Prop: deterministic, reflexive, symmetric, and transitive. For a product domain, we use integrity rules IR to specify restrictions on how domains are combined.

Note that these domain constructors can construct both parametric and specific abstract domains (when $\chi = \emptyset$). Also, the constructors can use both parameters and atomic values as basic domain building blocks.

A binary relation $\mathcal{P}(A_1(\chi_1) \times A_2(\chi_2))$ with deterministic property denotes a (partial) function $A_1(\chi_1) \to A_2(\chi_2)$. Binary relation without properties can be defined using powerset and product constructors.

Integrity rules allow us to better approximate the reduced product (Cousot and Cousot 1979) when combining domains using the product constructor. That is, if the component domains are not independent (e.g., use the same *VarId* parameter), then some of the tuples in the product domain might represent inconsistent concrete information. The integrity rules define consistent tuples. For brevity, we omit the syntax of integrity rules, and the (standard) definition of the satisfaction relation $x \models$ *IR*, for a concrete value x.

In Section 4 we explain how properties and integrity rules are used by the composition algorithm. The composition algorithm requires decidability of checking certain queries about the set of abstract values that satisfy *Prop* and *IR*.

The following example shows how several standard domains are expressed using our domain constructors.

EXAMPLE 3.2 (**Nullness of References).** A set of access paths of length at most 1 is $AP(\mathit{VarId}, \mathit{FieldId}) \stackrel{\text{def}}{=} \mathit{VarId} \times (\mathit{FieldId} \cup \{\epsilon\})$, where *VarId* and *FieldId* are domain parameters.

The parametric abstract domain for tracking nullness of references is $\mathit{NR}(\mathit{VarId}, \mathit{FieldId}) \stackrel{\text{def}}{=} \mathcal{P}(AP(\mathit{VarId}, \mathit{FieldId}))$; an abstract value $M \in \mathit{NR}(\mathit{VarId}, \mathit{FieldId})$ is a set of access paths that must have null value. □

EXAMPLE 3.3 (**Typestate).** The typestate abstract domain, parametric in *VarId*, *FieldId*, and AS, is a powerset of $\mathcal{D}$, defined as follows (omitting the domain parameters).

$$\begin{array}{l} \mathcal{D} = AS \times \mathcal{Q} \times MustSet \times Pts \times Alias \text{ with } IR_{\mathcal{D}} \\ MustSet = \mathcal{P}(AP) \\ Pts = \mathcal{P}(AP \times AS) \\ Alias = \mathcal{P}(AP \times AP) \text{ with reflexive, symmetric, transitive} \end{array}$$

Here, $a \in AS$ is an allocation site, $s \in \mathcal{Q}$ is a state of the typestate automaton, $M \in MustSet$ is a set of access paths that must point to a, $pts \in Pts$ is a flow-insensitive points-to information, and $alias \in Alias$ is a flow-insensitive alias information with reflexive, symmetric, and transitive properties.

A tuple $\langle a, s, M, pts, alias\rangle$ satisfies integrity rules $IR_{\mathcal{D}}$ iff
- if $p \in M$ then $\langle p, a\rangle \in pts$, and,
- if $\langle p_1, a\rangle \in pts$ and $\langle p_2, a\rangle \in pts$ then $\langle p_1, p_2\rangle \in alias$. □

Notations For a domain $A \stackrel{\text{def}}{=} A_1 \times \ldots \times A_k$, we use $\pi_i(A)$ to denote A_i, the i-th component in the product, and $\pi_i(a)$ to denote a_i the i-th component of the tuple $a = \langle a_1, \ldots, a_k\rangle$ in A. For access paths, we use $p_1.p_2$ as a shorthand for the tuple $\langle p_1, p_2\rangle$ in AP. We write p instead of $p.\epsilon$ when no confusion is likely. We omit the domain parameters from domain identifiers when they are clear from the context.

Given a parametric abstract domain constructed as above, we can instantiate it into a specific domain by binding its domain parameters to values.

DEFINITION 3.4 (Domain Binding). *Given a parametric abstract domain A, and a code fragment s, we use* $[\![A]\!]_S$ *to denote the abstract values of the specific abstract domain obtained from A for s. The set* $[\![A]\!]_S$ *is defined inductively:*

$$\begin{array}{ll} [\![A_1 \times \ldots \times A_k]\!]_S & \stackrel{\text{def}}{=} \{\langle c_1, \ldots, c_k\rangle \mid \langle c_1, \ldots, c_k\rangle \models \mathit{IR}, \forall i. c_i \in [\![A_i]\!]_S\} \\ [\![\mathcal{P}(A_1)]\!]_S & \stackrel{\text{def}}{=} \{X \mid X \textit{ is finite}, X \models \mathit{Prop}, X \subseteq [\![A_1]\!]_S\} \\ [\![A_1 \cup A_2]\!]_S & \stackrel{\text{def}}{=} [\![A_1]\!]_S \cup [\![A_2]\!]_S \end{array}$$

If A is a set of atomic values, then $[\![A]\!]_S = A$*. Finally,* $[\![\chi]\!]_S$ *is the set of program-specific values for* χ*, extracted from the code.*

For instance, $[\![\mathit{VarId}]\!]_S$ is the set of names of program variables, $[\![\mathit{FieldId}]\!]_S$ is the set of names of pointer fields, and $[\![AS]\!]_S$ is the set of names of allocation sites that appear in *s*.

EXAMPLE 3.5. For the parametric domain $AP(\mathit{VarId}, \mathit{FieldId}) \stackrel{\text{def}}{=} \mathit{VarId} \times (\mathit{FieldId} \cup \{\epsilon\})$, we get the set of values $[\![AP]\!]_{\texttt{x:=y.f}}$ is $\{x.f, x.\epsilon, y.f, y.\epsilon\}$, because $[\![\mathit{FieldId}]\!]_{\texttt{x:=y.f}}$ is $\{f\}$, and $[\![\mathit{VarId}]\!]_{\texttt{x:=y.f}}$ is $\{x, y\}$.

3.2 Micro-Transformers

Our method is restricted to abstract transformers described in terms of *micro-transformers*. Micro-transformers, defined in this section, operate on the components of an abstract value. If abstract transformers for all basic statements are expressed via the micro-transformers as defined in this section, then the algorithm presented in Section 4 generates composite transformers, which are also expressed via micro-transformers.

Let $\mathcal{A} = \mathcal{P}(Q)$ be the (top-level) abstract domain. We require that for every basic statement $stmt$ the abstract transformer $tr^{stmt} \colon \mathcal{A} \to \mathcal{A}$ be expressed using a micro-transformer that operates pointwise on the values from the domain Q:

$$tr^{stmt} = \lambda X. \bigcup_{x \in X} tr_Q^{stmt}(x)$$

where tr_Q^{stmt} is a micro-transformer for the domain Q. The micro-transformer tr_Q^{stmt} takes a value in Q and returns a set of values in Q.

A micro-transformer tr_Q^{stmt} can be defined using other micro-transformers that operate on the components of a value from Q, and so on. Before we proceed to formally define micro-transformers we first define the notion of query parameters that can be used in a micro-transformer.

One of the important concepts in this paper is the ability to make micro-transformers conditional on some information about the context. That is, a micro-transformer that operates on a component a of the value $q \in Q$ can also query another component c of q to determine the effect of a statement on a. We say that q is the context in which a is transformed, and c is a query parameter.

Formally, given $q \in Q$, $cl(q)$ is a tuple of components of q that can be queried, and $cl(Q)$ is a tuple of the corresponding domains. The operations $\rho_j(q)$ and $\rho_j(Q)$ return the j-th element of $cl(q)$ and $cl(Q)$, respectively. We say that $\rho_j(q)$ is a *query parameter*.

In the following examples, we assume that queried components and the corresponding domains are named, and refer to them by their names (rather than using ρ_j). For example, for typestate, we assume that the domain $\mathcal{D}$ has the following named components:

$$\begin{array}{l} cl(\langle a, s, M, pts, alias\rangle) = \langle a, M, pts, alias\rangle \\ cl(\mathcal{D}) = \langle AS, MustSet, Pts, Alias\rangle \end{array}$$

Given a domain Q, and query parameters for it, we formalize the notion of micro-transformers. A mapping $tr_A^S \colon Q \to A \to \mathcal{P}(A)$ that takes context $q \in Q$, and value $a \in A$, and returns a set of values from A, is a micro-transformer for a code fragment *s* and a domain A if it can be expressed in the syntax defined below.

In what follows, the same context q is used for all micro-transformers that are invoked (possibly indirectly) by $tr_Q(q)$. To avoid clutter of notations, we refer to value q in the body of micro-transformers without passing q explicitly as a parameter. That is, we use the following short syntax of micro-transformers: $tr_A^S \colon A \to \mathcal{P}(A)$. Here, *s* denotes the code fragment whose effect is captured by the transformer. We omit the superscript *s* when no confusion is likely.

DEFINITION 3.6 (**Micro-transformer Syntax**). *A micro-transformer for A is a mapping that can be defined in the following syntax:*

$$\begin{array}{lll} tr_A ::= & \lambda a.\{a\} & \textit{identity} \\ & \mid \lambda X.\{\bigcup_{x \in X} tr_{A_1}(x)\} & \textit{pointwise} \\ & \mid \lambda\langle a_1, \ldots, a_k\rangle. tr_{A_1}(a_1) \times \ldots \times tr_{A_k}(a_k) & \textit{separable} \\ & \mid \lambda a. \left\{ \begin{array}{ll} post_1(a) & pre_1(a, q) \\ \ldots & \ldots \\ post_n(a) & pre_n(a, q) \end{array} \right. & \textit{conditional} \end{array}$$

where the syntax of conditional micro-transformers is given in Definition 3.11.

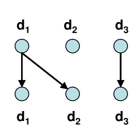
The identity micro-transformer returns the singleton of its input a. A pointwise micro-transformer for $A \stackrel{\text{def}}{=} \mathcal{P}(A_1)$ applies a micro-transformer tr_{A_1} for A_1 pointwise on every element of the input set $X \in A$. Note that tr_A returns a singleton whose element is a set of values from A_1. A separable micro-transformer for a product domain $A \stackrel{\text{def}}{=} A_1 \times \ldots \times A_k$ applies a micro-transformer tr_{A_i} for A_i to each component a_i of the input value separately (but possibly refers to the context q). If A is a union domain $A \stackrel{\text{def}}{=} A_1 \cup A_2$, then a micro-transformer for A can either be identity or conditional.

A conditional micro-transformer consists of n cases: for $i = 1, \ldots, n$, pre_i is a (non-empty) conjunction of literals, where a literal is a precondition query or its negation. A precondition query is a function from a and q to *true* or *false*. We require that preconditions are disjoint and total. Each precondition pre_i defines an equivalence class of input values with a uniform behavior, and the corresponding postcondition $post_i$ specifies how a value in this class is transformed. Each $post_i$ is a (possibly empty) set of postcondition terms. A postcondition term defines an output value of the micro-transformer as function of the input value a, without referring to the query parameters q. An important restriction of this syntax is that conditional micro-transformers cannot invoke other micro-transformers.

The restricted syntax of postcondition terms and precondition queries, and requirements on preconditions are detailed in Section 3.2.1.

The following example shows how we can easily encode any specific IFDS problem using the conditional micro-transformer syntax just defined.

EXAMPLE 3.7. In a specific IFDS problem, the dataflow facts are known, e.g., $D = \{d_1, d_2, d_3\}$, An example input/output relation on these dataflow facts, shown on the right, can be encoded as a conditional micro-transformer tr_D, as shown on the left. □



$$tr_D(d) = \left\{ \begin{array}{ll} d_1, d_2 & d = d_1 \\ d_3 & d = d_3 \end{array} \right.$$

For a specific IFDS problem, we can easily compose micro-transformers and get a composite micro-transformer in our syntax. In general, however, some restrictions are required on conditional micro-transformers to guarantee that the language of micro-transformers is closed under composition.

$$tr_{AS\times\mathcal{Q}}(r) = \begin{cases} \langle a, \delta_{open}(s)\rangle & p \in M \\ \langle a, \delta_{open}(s)\rangle, r & p \notin M \wedge \langle p, a\rangle \in pts \\ r & \langle p, a\rangle \notin pts \end{cases}$$

$$tr_{AP}(d) = \begin{cases} d & \pi_2(d) = f \wedge \langle \pi_1(d), this\rangle \notin alias \\ d & \pi_2(d) \neq f \wedge d \neq p \\ p, this.f & d = p \end{cases}$$

Table 1. Summary for the procedure `init`. In the typestate micro-transformer, we use a instead of $\pi_1(r)$, and s instead of $\pi_2(r)$, to denote the components of $r = \langle a, s\rangle$.

EXAMPLE 3.8. An abstract value for typestate is a set X of tuples from $\mathcal{D}$, defined in Example 3.3. An abstract transformer $tr\colon \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$ is defined by $tr(X) = \bigcup_{x\in X} tr_{\mathcal{D}}(x)$, where the micro-transformer for $\mathcal{D}$ is separable.

Let x be $\langle a, s, M, pts, alias\rangle$:

$$tr_{\mathcal{D}}(x) = tr_{AS\times\mathcal{Q}}(\langle a, s\rangle) \times tr_{MustSet}(M) \times \{pts\} \times \{alias\}$$

All micro-transformers for pts and $alias$ are identity, because these sets are carrying flow-insensitive information.

Table 1 shows the micro-transformers for the procedure `init` of Fig. 2: a micro-transformer for typestate pairs, denoted by $tr_{AS\times\mathcal{Q}}$, and a micro-transformer for the access paths in the must-set, denoted by tr_{AP}.

The typestate micro-transformer $tr_{AS\times\mathcal{Q}}$ describes the effect of the statement `p.open()`, the only statement in this procedure which alters the typestate. The first case of this transformer describes a strong update of the typestate: if the must-set M of the incoming value contains the access path p, then the resulting value in the typestate domain is $\langle a, \delta_{open}(s)\rangle$, where δ is the transition relation of the typestate automaton for `FileComp`. The second case describes a weak update of typestate: if p may point to the allocation site a, according to a global points-to analysis pts, and there is no must information about p in the incoming value, then the result contains two values, one in which the typestate has changed, and one in which it is preserved. The third case describes that the original value is preserved, when p is known to must not point to the allocation site a.

The must-set micro-transformer $tr_{MustSet}(M)$ is defined pointwise on the access paths in the set M, using the micro-transformer for individual access paths, $tr_{AP}(d)$. It describes the effect of the statement `this.f=p`. Note that the "kill" effect of the transformer is described implicitly: if $this$ may alias to some other variable v, according to the global may-alias analysis $alias$, then the f-field of the object pointed-to by v may change as a result of the destructive update `this.f=p`. In this case, we cannot guarantee that $v.f$ still points to the tracked object after the update, and, thus, an access path of the form $v.f$ where v may be aliased with $this$, will not be propagated by this transformer. □

3.2.1 Restrictions on Conditional Micro-Transformers

In this section, we describe syntactic restrictions on preconditions and postconditions that ensure that conditional micro-transformers are closed under composition.

We start by giving the syntax of terms that we use in preconditions and postconditions. A term can refer to specific values that appear in the transformed code fragment *s*, e.g., it can explicitly refer to program variables that appear in *s*, members of the set $[\![VarId]\!]_{s}$. Values that appear in the rest of the code are referred to "symbolically" using the parameters a and q. The fact that we do not explicitly refer to these values in the transformer allows us to concisely describe the behavior of the code fragment *s* when the number of different contexts is large (interprocedural analysis) or even infinite (modular analysis). Moreover, a term can refer to function symbols such as $f\colon B \to B$, for some domain B, whose semantics is independent of the analyzed program.

A term in $T^{s}_{A\to B}(a)$ denotes a value in domain B as a function of the value of the parameter a from domain A.

DEFINITION 3.9. *A term in $T^{s}_{A\to B}(a)$ has the following syntax:*

$$\begin{array}{lll} T^{s}_{A\to B}(a) ::= & a & \textit{if } B = A \\ & \mid v & \textit{if } v \in [\![B]\!]_s \\ & \mid f(t) & \textit{if } t \in T^{s}_{A\to B}(a),\, f\colon B \to B \textit{ is symbolic} \\ & \mid \pi_i(t) & \textit{if } \pi_i(C) = B,\, t \in p^{A}_{C}(a) \\ & \mid \langle t_1, \ldots, t_k\rangle & \textit{if } B = B_1 \times \ldots \times B_k, \textit{ and} \\ & & \forall i.1 \leq i \leq k : t_i \in T^{s}_{A\to B_i}(a) \end{array}$$

For example, in Table 1, $\delta_{open}(\pi_2(r))$ is a term in $T^{\texttt{init}}_{AS\times\mathcal{Q}\to\mathcal{Q}}(r)$, which refers to the function δ_{open} symbolically.

There are no special restrictions on the syntax of terms that can be used in a postcondition of a conditional micro-transformer: for a domain A and a code fragment *s*, any term in $T^{s}_{A\to A}(a)$ can be used in the postcondition. We restrict the precondition queries to guarantee that micro-transformers are closed under composition, and that their composition can be done automatically.

DEFINITION 3.10. *Given a code fragment s, and a domain A, a precondition query is defined by the following syntax:*

$$\begin{array}{lll} C^{s}_{A}(a, q) ::= & t = v & \textit{if } v \in [\![B]\!]_s \\ & \mid t = \rho_j(q) & \textit{if } \rho_j(Q) = B, B \textit{ is invertible}, \rho_j(q) \neq a \\ & \mid t \in \rho_j(q) & \textit{if } \rho_j(Q) = \mathcal{P}(B), B \textit{ is invertible} \end{array}$$

where t is a term in $T^{s}_{A\to B}(a)$.

A precondition is a conjunction of literals, where a literal is a precondition query from $C^{s}_{A}(a, q)$ or its negation. We use $Pre^{s}_{A}(a, q)$ to denote the set of all preconditions.

In the first case, a precondition query is an equality test of a term and a specific value, independent of q. In the second case, it is an equality test of a term and a query parameter. In the third case, it is a set membership test of a term in a set described by a query parameter.

The restricted structure of the query will allow us to compute the weakest precondition of a query and guarantee that the (simplified) result is in the language of our preconditions. The most important restriction is that domain B associated with the query parameter be invertible,[2] as defined in Section 3.2.2.

For example, in Table 1, we use the query $p \in M$ where p is an access path and the corresponding domain AP is required to be invertible.

It is worth noting here that the language of preconditions is closed under conjunction. Also, the language of postcondition terms is closed under substitution. Thus, the language of preconditions is also closed under substitution of these terms. We rely on these facts, among others, to show that the result of our composition algorithm is in the language of our micro-transformers.

Moreover, given a code fragment, the language of (normalized) terms and queries are finite, assuming no symbolic functions are used. We rely on this to show that the algorithm for summary generation terminates and the summaries are finite.

Notations Let s_1 and s_2 be two code fragments. For terms $t_1 \in T^{s_1}_{A\to B}(a)$ and $t_2 \in T^{s_1}_{A\to A}(a)$, we use $t_1[a \to t_2]$ to denote the term in $T^{s_1\cup s_2}_{B\to A}(a)$ obtained by substitution of t_2 instead of a in t_1. For a term $t \in T^{s_1}_{A\to B}(a)$ and a specific value $w \in [\![A]\!]_{s_2}$, we use $t(w)$ to denote the (unique) value of the term obtained by replacing a with w in t. Note that $t(w) \in [\![B]\!]_{s_1\cup s_2}$. Similarly, for a precondition query $c \in C^{s_1}_{A}(a, q)$, and a value $u \in [\![cl(Q)]\!]_{s_2}$ of the query parameters referred to by the query, $c(w, u)$ denotes

[2] For the second test, the requirement on B can be weakened.

the value *true* or *false*. We lift this interpretation to boolean combinations of queries in the usual way. For a micro-transformer $tr^{s_1}_A$, values w and u as above, if s_1 contains s_2, we use $tr^{s_1}_A(w)$ to denote the set of specific values in $[\![A]\!]_{s_2}$ returned by the transformer when its input value is w and the the query parameters has value u.

Now that we have defined the restrictions on terms and queries, we are finally ready to define the restrictions on conditional micro-transformers.

Given a code fragment s, the effect of a conditional micro-transformer on A can be decomposed into equivalence classes of values in $[\![A]\!]_s$ with a uniform behavior, and each equivalence class is described by one of the cases in the micro-transformer. The following conditions require that every input value satisfies the precondition of exactly one case in a micro-transformer.

DEFINITION 3.11 (Conditional Micro-Transformer Syntax). *Given a code fragment s and a domain A, a conditional micro-transformer tr^s_A is defined by*

$$tr^s_A = \lambda a. \begin{cases} post_1 & pre_1 \\ \ldots & \ldots \\ post_n & pre_n \end{cases}$$

where

- *for all $i = 1, \ldots, n$, $post_i \subseteq T^s_{A \to A}(a)$,*
- *for all $i = 1, \ldots, n$, $pre_i \in Pre^s_A(a, q)$,*
- *for every code fragment s_1 that contains s, for every value $u \in [\![cl(Q)]\!]_{s_1}$, $w \in [\![A]\!]_{s_1}$, there exists a unique i, $1 \leq i \leq n$, such that $pre_i(w, u)$ is* `true`.

We rely on these conditions to provide an efficient composition algorithm for conditional transformers, as explained in Sec. 4.

Remark. A conditional transformer can represent the ''kill" effect of a statement using cases whose postcondition is an empty set (in the examples, these cases are omitted). A transformer can represent the "gen" effect of a statement using a designated value Λ, propagated by every pointwise transformer (similar to the use of Λ in (Reps et al. 1995; Sagiv et al. 1996b)). In typestate abstraction, for example, we use Λ cases to model allocation. The transformer $tr_{AS \times Q}(r)$ for an allocation statement `x = new FileComp()` at allocation site a_L contains the case with precondition $r = \Lambda$, whose postcondition is the singleton $\langle a_L, init\rangle$. For brevity, we omit the discussion about Λ from this paper.

3.2.2 Invertible Micro-Transformers

As we will see in Section 4, the composition algorithm for conditional micro-transformers uses the $invert$ operation to compute the reverse-image of some micro-transformers. The success of the composition algorithm relies on the ability of $invert$ to compute simple precondition on the input values of a micro-transformer for which the micro-transformer yields a certain value. This motivates the following definition.

DEFINITION 3.12 (Invertible). *Let $tr^{s_1}_B$ be a micro-transformer for domain B and code fragment s_1. The micro-transformer $tr^{s_1}_B$ is invertible iff there exists a computable operation $invert$ such that for every domain A that queries B, for every code fragment s_2 and term t in $T^{s_2}_{A \to B}(a)$, the result of $invert(tr^{s_1}_B, t)$ is $\bigvee_{i=1}^k \psi_i$ where*

- *for all $i = 1, \ldots, k$, ψ_i is of the form $(b = t') \wedge pre$, where $t' \in T^{s_1 \cup s_2}_{A \to B}(a)$, $pre \in Pre^{s_1 \cup s_2}_A(a, q)$, and $(b = t')$ is optional;*
- *for every code fragment s that contains both s_1 and s_2, for every $u \in [\![cl(Q)]\!]_s$, $w \in [\![A]\!]_s$, and $v \in [\![B]\!]_s$,*

  $t(w) \in tr^{s_1}_B(v)$ *iff there exists i such that $\psi_i(w, v, u)$ is* `true`.

Domain B is invertible *when for all basic statements, the micro-transformers for B are invertible.*

$$compose(tr^{12}_A, tr^{23}_A) = \begin{cases} tr^{12}_A & tr^{23}_A \text{ is identity} \\ tr^{23}_A & tr^{12}_A \text{ is identity} \\ \lambda X.\{\bigcup_{x \in X} tr^{13}_{A_1}(x)\} & tr^{12}_A, tr^{23}_A \text{ are pointwise}, \\ & A = \mathcal{P}(A_1), \\ & tr^{13}_{A_1} = compose(tr^{12}_{A_1}, tr^{23}_{A_1}) \\ \lambda\langle a_1, \ldots, a_k\rangle. & tr^{12}_A, tr^{23}_A \text{ are separable}, \\ \quad tr^{13}_{A_1}(a_1) \times \ldots \times tr^{13}_{A_k}(a_k) & A = A_1 \times \ldots \times A_k, \forall i.1 \leq i \leq k: \\ & tr^{13}_{A_i} = compose(tr^{12}_{A_i}, tr^{23}_{A_i}) \\ composeCond(tr^{12}_A, tr^{23}_A) & tr^{12}_A, tr^{23}_A \text{ are conditional} \end{cases}$$

Table 2. Composition algorithm for micro-transformers. The input is a pair of micro-transformers $tr^{12}_A, tr^{23}_A : A \to \mathcal{P}(A)$. The output is a micro-transformer for A. The subroutine $composeCond$ is given in Fig. 5.

The following lemma defines syntactic restrictions on a conditional micro-transformer that guarantee that they are invertible.

LEMMA 3.13. *A conditional micro-transformer tr^s_B is* invertible *if every case i in tr^s_B, and every term t in $post_i$ satisfy one of the following:*

- *t is b,*
- *pre_i is of the form $b = w \wedge pre'$ where $w \in [\![B]\!]_s$ and b does not appear in pre' (and we place no restrictions of t),*
- *t is of the form $\langle t_1, \ldots, t_k\rangle$, and there is a set of indexes $J = \{j_1, \ldots, j_m\} \subseteq \{1, \ldots, k\}$, such that pre_i is of the form*

  $$(\pi_{j_1}(b) = w_1) \wedge \ldots \wedge (\pi_{j_m}(b) = w_m) \wedge pre'$$

  where for all $j = 1, \ldots, m$, $w_j \in [\![B]\!]_s$, b does not appear in pre', and for all $j = 1, \ldots, k$, either $j \in J$ or t_j is $\pi_j(b)$,
- *t is $f(t')$, f^{-1} is a function, and t' and pre_i satisfy one of the conditions above (with t' in place of t).*

A degenerate case of invertible domain is immutable domain.

DEFINITION 3.14 (Immutable Domain). *Domain B is* immutable *when for all basic statements, the micro-transformers for B are identity.*

The $invert$ of operation for immutable domain B is essentially identity: $invert(tr_B, t)$ is $(b = t)$.

In our typestate example, the domains Pts and $Alias$ are immutable, and the domain AP is invertible.

To guarantee that the weakest precondition is computable, our method restricts domains of certain query parameters to be invertible, depending on the way the query parameter is used in preconditions. Not all domains need to be invertible.

4. Composition Algorithm

In this section, we describe a composition algorithm for micro-transformers and employ it for computing functional composition of abstract transformers, the basis of our framework for generating procedure summaries.

Given a pair of micro-transformers $tr^{12}_A, tr^{23}_A : A \to \mathcal{P}(A)$, the algorithm, shown in Table 2, returns a micro-transformer for A that precisely captures the composed effect of tr^{12}_A and tr^{23}_A.

The main part of the algorithm is the subroutine `composeCond` shown in Fig. 5, which composes conditional micro-transformers, as described Sec. 4.1. This procedure computes a generalized weakest precondition of queries that appear in the preconditions of a micro-transformer, as described in Sec. 4.2. It relies on a decision procedure for checking consistency of preconditions, and for simplification of summaries, to guarantee that the summary is precise and that its size is bounded, as explained in Sec. 4.3.

```
composeCond(tr_A^12, tr_A^23) {
// tr_A^12 = {⟨pre_i^12, post_i^12⟩ | 1 ≤ i ≤ n}
// tr_A^23 = {⟨pre_j^23, post_j^23⟩ | 1 ≤ j ≤ m}
 tr_A^13 := ∅
 for each ⟨pre^12, post^12⟩ in tr_A^12
  k := |post^12|
  for each set of indexes I := {i_1, ..., i_k}
   s.t. 1 ≤ i_l ≤ m for all l = 1, ..., k {
   // cover I with maximally-consistent
   // (possibly-overlapping) sets
   cover := ∅ // map: P(I) to pairs of pre/post conds
   for each J ⊆ I {
    if not exists K ∈ domain(cover) s.t. J ⊆ K {
    // J not subsumed by other index set in cover
     pre^13 := simplify-pre(pre^12 ∧ ⋀_{j=1}^{|J|} wp(pre_{i_j}^23, t_j))
     post^13 := ⋃_{j=1}^{|J|} {simplify-term(t[a → t_j]) | t ∈ post_{i_j}^23}
     if pre^13 is consistent {
      // remove all index sets K subsumed by J
      cover := cover \ {K ↦ ⟨..⟩ | K ⊆ J}
      // add J to cover
      cover := cover ∪ [ J ↦ DDNF(⟨pre^13, post^13⟩) ]
     }
    }
   }
   tr_A^13 := tr_A^13 ∪ image(cover)
  }
 }
 return tr_A^13
}
```

Figure 5. Composition algorithm for conditional transformers.

4.1 Composition of Conditional Transformers

The composed transformers can be viewed as relating values in three domains: (A1), (A2), and (A3), where tr^{12} transforms values between (A1) and (A2), and tr^{23} transforms values between (A2) and (A3). The key to composition is to express the restriction imposed by the composition of tr_A^{12} and tr_A^{23} on the values of (A2) as restrictions on (A1) and (A3).

Intuitively, our composition algorithm operates in two stages, depicted in Fig. 1(c):

(I) computing the reverse image of preconditions of tr_A^{23} under the transformer tr_A^{12}. The result is that all preconditions in tr_A^{23}, previously expressed in (A2), are now expressed in (A1).

(II) computing the forward image of the postconditions of tr_A^{12} under the transformer tr_A^{23}. The result is that all postconditions in tr_A^{12}, previously in (A2), are now expressed in (A3).

To make the composition process feasible, our approach leverages the structure of transformers.

Fig. 5 shows the pseudo-code of the composition algorithm of conditional micro-transformers. Note that the computation of pre^{13} corresponds to the intuitive step (1) above (realized as computation of the weakest precondition). Note that the computation of $post^{13}$ corresponds to the intuitive step (2) above (realized as substitution). Both steps are making calls to simplification procedures. Also note that the iteration in the algorithm is required to handle postconditions with multiple terms, and it guarantees that all cases are covered.

The following example illustrates how the composition algorithm obtains a summary using the typestate abstraction of Example 3.8.

EXAMPLE 4.1. We illustrate how the composition algorithm obtains a summary of the procedure `readData` shown in Fig. 2 from those of `init` and `process`.

The summary $tr^{init}_{AS\times\mathcal{Q}}(\langle a, s\rangle)$ for `init` is shown in Table 1. Consider the composition of the first case of the summary for `init`

$$\langle \pi_1(r), \delta_{open}(\pi_2(r))\rangle \quad \text{if} \quad p \in M$$

with the following case of the summary $tr^{process}_{AS\times\mathcal{Q}}(r)$ for `process`:

$$\langle \pi_1(r), \delta_{close}(\pi_2(r))\rangle \quad \text{if} \quad this.f \in M$$

We use $t(r)$ to denote the term $\langle \pi_1(r), \delta_{open}(\pi_2(r))\rangle$.

First, the composition algorithm finds a new precondition on the values of input parameter r and the query parameter q under which the output value $t(r)$ of `init` satisfies the precondition $this.f \in M$ of `process`. Towards this end, we compute the (generalized) weakest precondition of $this.f \in M$, as described in Section 4.2.

$$wp(this.f \in M, t(r)) = p \in M$$

The result is conjoined with the precondition $p \in M$, which happens to be the same syntactically, in this simple example, therefore, the new precondition is consistent. Under the new precondition, we replace r by $\langle \pi_1(r), \delta_{open}(\pi_2(r))\rangle$ in the postcondition $\langle \pi_1(r), \delta_{close}(\pi_2(r))\rangle$ of `process`. After simplification, we get the postcondition

$$\langle \pi_1(r), \delta_{close}(\delta_{open}(\pi_2(r)))\rangle$$

Now consider the composition of the third case of the summary of `init` with the same case of the summary of `process`. The weakest precondition for $this.f \in M$ is the same as before, because the query does not depend on r. We conjoin the weakest precondition $p \in M$ with the precondition $\langle p, \pi_1(r)\rangle \notin pts$ from `init`. The consistency checker finds out that the new precondition is inconsistent with the integrity rule "if $p \in M$ then $\langle p, \pi_1(r)\rangle \in pts$". Hence, the algorithm does not generate a new postcondition for this combination of cases. □

4.2 Weakest Preconditions

To obtain a composed transformer, we use the operation *wp* to express each precondition pre_j^{23} that appears in tr_A^{23} and refers to the intermediate state in (A2), in terms of the initial state, in (A1).

The weakest-precondition operation *wp* for code fragment s takes as input a precondition pre and a term t, and returns a boolean combination of preconditions.

Formally, given a code fragment s_2, a precondition $pre \in Pre_A^{s_2}(a, q)$, and $t \in T^S_{A\to A}(a)$, we define *wp* inductively on the syntax of pre:

$$wp(pre, t) \stackrel{\text{def}}{=} \begin{cases} wp(pre_1, t) \wedge wp(pre_2, t) & \text{if } pre \text{ is } pre_1 \wedge pre_2 \\ \neg wp(pre_1, t) & \text{if } pre \text{ is } \neg pre_1 \\ lift(\rho_j(q), invert(tr^S_B, t'')) & \text{if } pre \text{ is } t' \in \rho_j(q) \text{ or } t' = \rho_j(q), \\ & \text{and } t' \in T^S_{A\to B}(a), t'' \stackrel{\text{def}}{=} t'[a \to t] \\ pre[a \to t] & \text{otherwise} \end{cases}$$

The weakest precondition is closed under conjunction, as usual. It is also closed under negation, because all transformers that we use are deterministic. For the base case of a precondition, a query in $C^S_A(a, q)$, we use substitution, as usual, when the query is independent of q.

Recall from Definition 3.10 that if $\rho_j(q)$ is used in a precondition query then then micro-transformer tr_B must be invertible. When the query is of the form $t \in \rho_j(q)$, or $t = \rho_j(q)$, we compute weakest precondition using $invert$ for transformer tr_B and *lift*. The idea for handling weakest precondition computation of a set membership query of the form $t \in \rho_j(q)$ is to break it down into equality queries on the underlying domain, compute the weakest precondition in the underlying domain, and lift the result back to a membership query in the powerset domain. Intuitively, this corresponds to observing the effect of an update on a set membership query by observing its effect on individual members of the set.

Technically, we compute weakest precondition pointwise, where $invert(tr_B, t)$ operation, defined in Section 3.2.2, computes the reverse image of tr_B, and the *lift* replaces every equality queries over B by membership queries over $\mathcal{P}(B)$.

The $invert$ operation is implemented as:

$$invert(tr_B, t) \stackrel{\text{def}}{=} \bigvee_{i=1}^{n} \bigvee_{l=1}^{k_i} \textit{simplify-pre}(t = t_l \wedge pre_i)$$

For each term t_l in a postcondition of tr_B, we unify t with t_l and conjoin it with the corresponding precondition to get a condition on b that guarantees that the result of applying transformer tr_B to b is t. Since tr_B is invertible, the result of simplification is a disjunction of preconditions each of which is of the form $b = t' \wedge pre_1$, where $t' \in T_{A \to B}(a)$ and b does not occur in pre_1. This syntactic form allows us, for instance, to lift each pointwise query $b = t'$ that appears in $invert(tr_B, t)$ to the membership query $t' \in \rho_j(q)$. Formally, $\textit{lift}(\rho_j(q), \psi)$ is defined inductively on the syntax of ψ:

$$\textit{lift}(\rho_j(q), \psi) \stackrel{\text{def}}{=}$$
$$\begin{cases} \textit{lift}(\rho_j(q), \psi_1) \vee \textit{lift}(\rho_j(q), \psi_2) & \text{if } \psi \text{ is } \psi_1 \vee \psi_2 \\ \textit{lift}(\rho_j(q), \psi_1) \wedge \textit{lift}(\rho_j(q), \psi_2) & \text{if } \psi \text{ is } \psi_1 \wedge \psi_2 \\ \neg \textit{lift}(\rho_j(q), \psi_1) & \text{if } \psi \text{ is } \neg\psi_1 \\ t' \in \rho_j(q) & \text{if } \psi \text{ is } b = t' \text{ and } \rho_j(Q) = \mathcal{P}(B) \\ t' = \rho_j(q) & \text{if } \psi \text{ is } b = t' \text{ and } \rho_j(Q) \text{is not } \mathcal{P}(B) \\ \psi & \text{otherwise} \end{cases}$$

EXAMPLE 4.2. The weakest precondition used in Example 4.1 is generated by breaking the weakest precondition computation to work pointwise on the underlying domain.

$$wp(this.f \in M, t) = \textit{lift}(M, invert(tr_{AP}^{init}, this.f))$$

To compute $invert(tr_{AP}^{init}, this.f)$, we use the micro-transformer tr_{AP}^{init} shown in Table 1. After unifying $this.f$ with each term in the postcondition of tr_{AP}^{init}, and conjoining it with the corresponding precondition, we get the following cases:

$$\begin{array}{l} this.f = d \wedge \pi_2(d) = f \wedge \langle \pi_1(d), this \rangle \notin alias \\ this.f = d \wedge \pi_2(d) \neq f \wedge d \neq p \\ this.f = this.f \wedge d = p \\ this.f = p \wedge d = p \end{array}$$

We simplify and check consistency of each of these cases. The first case implies that $\langle \pi_1(this.f), this \rangle \notin alias$ and after simplification of car and tuple-constructor, we get $\langle this, this \rangle \notin alias$. The consistency checker detects inconsistency with the reflexivity property of $alias$. The last case is inconsistent because $this.f \neq p$ (recall that this is a syntactic equality of access paths). Only the third case survives the consistency check and we get that $invert(tr_{AP}^{init}, this.f)$ is $d = p$. Lifting this equality constraint back to the powerset domain yields the membership query $p.\epsilon \in M$, as a result of wp. □

Remark. In general, the result of $wp(pre_1, t)$ is a boolean combination of conditions, whereas the syntax of conditional micro-transformers does not allow disjunctions in preconditions. Therefore, in the algorithm shown in Fig. 5, we use the operation *DDNF*, defined as follows:

$$\textit{DDNF}(\langle pre, post \rangle) = \{\langle pre', post \rangle \mid pre' \in \textit{DDNF}(pre)\}$$

where $\textit{DDNF}(pre)$ converts a boolean combination of conditions pre into an equivalent disjunction of disjoint consistent preconditions. In the worst-case, this operation can cause exponential blow-up. In practice, the size of the composite transformers and their preconditions is expected to remain small, because many of the generated preconditions are inconsistent. Our experiments in Section 6 support this hypothesis.

4.3 Consistency Checking and Simplification

The composition algorithm relies on the computable operations *simplify-term* and *simplify-pre* on terms and preconditions, respectively, to limit the size of its result without losing precision. These operations are essential for showing that the language of summaries is finite and that the algorithm for generating summaries (Section 5.1) terminates.

In addition, the weakest precondition computation (described in Section 4.2) depends on the ability of *simplify-pre* to produce a new precondition whose pointwise representation can be *lift*ed and expressed in terms of the query parameters.

Intuitively, the purpose of *simplify-term* and *simplify-pre* is to replace nested terms and complex preconditions by equivalent simpler ones.

The challenge in *simplify-term* is to handle postconditions with nested functions, whose semantics depends on the domain A. For example, the result of composition shown in Example 4.1 contains the term $\delta_{close}(\delta_{open}(\pi_2(r)))$ with nested δ functions, although the input micro-transformers did not have nesting. Subsequent compositions might create deeper nesting of δ functions. In Section 5.2.3, we show how to simplify nested δs and guarantee that their size is bounded.

The main challenge in *simplify-pre* is checking consistency, taking into account integrity rules *IR*, properties *Prop*, and structural rules for tuple-constructors and selectors, e.g., $\pi_i(\langle \ldots, a_i, \ldots \rangle) = a_i$. [3] In particular, the consistency check guarantees that all preconditions in the composite micro-transformer are consistent.

4.4 Properties of the Composition Algorithm

Consider a (top-level) abstract domain $\mathcal{A} \stackrel{\text{def}}{=} \mathcal{P}(Q)$. Given abstract transformers $tr^{12}, tr^{23} \colon \mathcal{A} \to \mathcal{A}$, expressed by micro-transformers tr_Q^{12}, tr_Q^{23}, respectively, we generate an abstract transformer that captures the composed effect of tr^{12} and tr^{23}. The result is an abstract transformer expressed by the micro-transformer that captures the composed effect of tr_Q^{12} and tr_Q^{23}:

$$compose(tr^{12}, tr^{23}) = \lambda X. \bigcup_{x \in X} compose(tr_Q^{12}, tr_Q^{23})(x)$$

The composition algorithm for micro-transformers is in Table 2.

DEFINITION 4.3 (Compatible Transformers). *Abstract transformers tr^{12} and tr^{23} are* compatible *iff they are expressed using compatible micro-transformers. Micro-transformers tr_A^{12} and tr_A^{23} are compatible iff one of them is identity, or both are pointwise, or both are separable, or both are conditional, and the micro-transformers for all components of A are compatible.*

In particular, if tr_A^{12} is a pointwise micro-transformer and tr_A^{23} is a conditional micro-transformer, then they are not compatible.

The following theorem states that given two abstract transformers $tr^{12}, tr^{23} \colon A \to A$ that are compatible and defined by micro-transformers, as in Section 3.2, the $compose$ algorithm computes their functional composition. Moreover, the result is a transformer also expressed by micro-transformers.

THEOREM 4.4 (**Composition Algorithm).** *If $tr^{12}, tr^{23} \colon \mathcal{A} \to \mathcal{A}$ are compatible abstract transformers, the result of $compose(tr^{12}, tr^{23})$ is a transformer that is also expressed via micro-transformers and computes the function $\lambda a.tr^{23}(tr^{12}(a))$.*

Let $\mathcal{T}$ denote the set of abstract transformers of basic statements that satisfy the following properties: (a) every abstract transformer is expressed by micro-transformers, (b) all micro-transformers for

[3] Extension of the Theory of Lists (Nelson and Oppen 1980) to k-tuples.

the same domain A are compatible. The language of summaries, denoted by $\mathcal{L}$, is the closure of $\mathcal{T}$ under composition.

The following theorem defines sufficient conditions of *simplify-term* and *simplify-pre* that guarantee that the language of summaries is finite. If the language $\mathcal{L}$ is finite, then the algorithm for computing procedure summaries terminates and produces finite summaries.

THEOREM 4.5 **(Finite Language of Summaries).** *The language of summaries $\mathcal{L}$ is finite if the following properties hold:*
1. *Bound on the size of terms: for every code fragment $stmt$, there exists a bound K such that for every pair of terms $t_1(a)$ and $t_2(a)$ over $stmt$, $|simplify\text{-}term(t_2(t_1(a)))| \leq K$.*
2. *Bound on the size of preconditions: for every code fragment $stmt$, there exist a bound N such that for every pair of preconditions pre_1 and pre_2 over $stmt$, $|simplify\text{-}pre(pre_2 \wedge pre_1)| \leq N$.*

When the micro-transformers contain function symbols or integrity rules, the user of our framework need to supply appropriate *simplify-term* and *simplify-pre*.

5. Generating Procedure Summaries

In this section we describe how the composition algorithm is put to use in a general framework for generating procedure summaries and show how to instantiate the framework for several applications.

5.1 Framework

Our framework computes procedure summaries by performing abstract interpretation over a domain in which the abstract values are sets of micro-transformers. Intuitively, an abstract value at a program point L represents the abstract transformer for the code fragment between procedure entry and the program point L.

To compute the summary of a procedure, our framework starts with an initial identity transformer that maintains the original value of procedure parameters. It then proceeds to compute the fixed-point of propagating this initial transformer through the procedure. In order to apply the effect of a statement to an incoming abstract value, each incoming micro-transformer is composed with the basic transformer of the statement. In order to apply the effect of a procedure invocation to an incoming abstract value, each incoming micro-transformer is composed with the summary of the invoked procedure (after replacing formals with actuals). When the analysis of a procedure reaches a fixed-point, the summary of the procedure is the set of micro-transformers at the point of procedure exit.

Join Operation Our framework supports an optional join operation for micro-transformers. Given two micro-transformers tr^1_A and tr^2_A, we define a join operation $tr^1_A \sqcup tr^2_A$ such that it emulates the join of the underlying domain A transformed by tr^1_A and tr^2_A. That is, for every a in A, $(tr^1_A \sqcup tr^2_A)(a) = tr^1_A(a) \sqcup tr^2_A(a)$.

Remark. Our framework is designed for abstract domains with finite-height. For infinite-height domains, *precise* procedure summaries are not well-defined because termination of abstract interpreter is achieved using widening, which does not guarantee precision, e.g., result may depend on order of chaotic iteration. It is possible to encode widening in our framework, similarly to join, but we lose the ability to guarantee precision.

5.2 Applications

First, we show that specific IFDS and IDE problems can be encoded in our framework. Second, we show that parametric version of interesting IFDS and IDE problems can be encoded in our framework: typestate verification with aliasing and constant propagation with aliasing (the latter requires join).

5.2.1 IFDS

In a (specific) IFDS problem (Reps et al. 1995), the dataflow facts are known $D = \{d_1, \ldots, d_k\}$. The specific abstract domain is $\mathcal{P}(D)$. The abstract transformers for basic statements are of the form: $tr(X) = \bigcup_{d \in X} tr_D(d)$ where tr_D is a conditional micro-transformer of the form

$$tr_D(d) = \begin{cases} S_1 & d = d_1 \\ \ldots & \ldots \\ S_k & d = d_k \end{cases}$$

and $S_i \subseteq D$ for all $i = 1, \ldots, k$. In particular, all preconditions in the conditional transformers for a specific IFDS domain are always of the limited form $d = d_i$ for some $1 \leq i \leq k$, i.e., the preconditions do not refer to query parameters; all postconditions are subsets of dataflow facts. An example transformer is shown in Example 3.7.

5.2.2 IDE

A (specific) abstract domain for IDE problems (Sagiv et al. 1996b) with environment $D \rightarrow L$ is defined by $Env \stackrel{\text{def}}{=} \mathcal{P}(D \times L)$ with deterministic property, and the symbols are known $D = \{d_1, \ldots, d_k\}$. Note that the abstract domain has no parameters, as we are encoding a specific IFe problem, in contrast to a modular version of an IDE problem, considered in Section 5.2.4.

The abstract transformers are $tr(env) = \bigcup_{\langle d,l \rangle \in env} tr_{D \times L}(\langle d, l \rangle)$ where $tr_{D \times L}$ is a conditional micro-transformer of the form

$$tr_{D \times L}(\langle d, l \rangle) = \begin{cases} post_1 & d = d_1 \\ \ldots & \ldots \\ post_k & d = d_k \end{cases}$$

and $post_i = \{\langle d_j, f_{d_i,d_j}(l) \rangle \mid j = 1, \ldots, k\}$, for all $i = 1, \ldots, k$. The function $f_{d_i,d_j} : L \rightarrow L$ captures the effect that the value of d_i in the input environment has on the value of d_j in the output environment. The efficient representation of functions f_{d_i,d_j}, which is one of the requirements in the IDE framework, guarantees that (i) the join operation on micro-transformers can be implemented using the join for L, and (ii) *simplify-term* can be implemented using the composition for L such that there is a bound on the size of terms.

5.2.3 Typestate

We have described the domains used for our typestate abstraction in Example 3.3, and the structure of typestate transformers in Example 3.8. Fig. 6 shows the conditional micro-transformers for updating a pair $r \in AS \times \mathcal{Q}$ under a typestate operation `x.op()`, and allocation. It uses the query parameters pts and M where the transformer tr_{AP} is invertible. Fig. 7 shows the conditional micro-transformer for updating must information under the basic statements. The query parameter $alias$ is used to perform strong update for `x.f = null`. We do not show statements for which the transformers are identity. Note that the typestate abstraction of (Fink et al. 2006), which we follow in this example, treats branch statements as identity.

We show that for a given procedure P, the summaries for $AS \times \mathcal{Q}$ domain are of bounded size. The difficulty is that after substitution, the postcondition of a case can contain terms t with arbitrarily nested δ's. We simplify these terms using the fact that they represent states of the finite-state automaton $\mathcal{F}$, as follows.

We say that terms $t, t' \in \Sigma^*$ are equivalent when for all states $q \in Q$, $t(q)$ and $t'(q)$ is the same state. The idea is to keep only the shortest term from each equivalence class of this relation. The length of shortest terms t is bounded by the *diameter* of $\mathcal{F}$. (This is not a problem in practice, because the automata we consider, which represent typestate properties, are usually small.)

Statement	$tr_{AS\times Q}(M)(r)$	
$x.op()$	$\langle a, \delta_{op}(s)\rangle$	$\langle x, a\rangle \in pts \wedge x \in M$
	$\langle a, \delta_{op}(s)\rangle, r$	$\langle x, a\rangle \in pts \wedge x \notin M$
	r	$\langle x, a\rangle \notin pts$
$x = new\ a_L$	$\langle a_L, s_{init}\rangle$	$r = \Lambda$
	r	$r \neq \Lambda$

Figure 6. Typestate abstraction: transformers for a pair r of tracked allocation site $a \stackrel{\text{def}}{=} \pi_1(r)$, and its typestate $s \stackrel{\text{def}}{=} \pi_2(r)$.

Statement	$tr_{AP}(a)(d)$	
$x = null$	d	$\pi_1(d) \neq x$
$x = y$	$d, \langle x, \pi_2(d)\rangle$	$\pi_1(d) = y$
	d	$\pi_1(d) \neq y$
$x = y.f$	$d, x.\epsilon$	$d = y.f$
	d	$d \neq y.f$
$x.f = null$	d	$\pi_2(d) = f \wedge \neg\langle \pi_1(d), x\rangle \in alias$
	d	$\pi_2(d) \neq f$
$x.f = y$	$d, x.f$	$d = y.\epsilon$
	d	$d \neq y.\epsilon$
$x = new\ a_L$	$x.\epsilon$	$d = \Lambda \wedge a = a_L$
	d	$d \neq \Lambda \wedge \pi_1(d) \neq x$

Figure 7. Typestate abstraction: transformers for an access path d.

However, it is worth noting that the size of a summary can be exponential in the size of VarId_P, if the procedure distinguishes between different aliasing contexts.

5.2.4 Constant Propagation with Aliasing

Linear constant propagation is a variant of constant propagation that was given an efficient solution by the IDE framework of (Sagiv et al. 1996a). We deal with a generalization of this problem to pointers and aliasing. We solve the parametric version of this problem, which allows us to generate summaries in a modular way, without knowing the rest of the program.

The abstract domain, parametric in *VarId* and *FieldId*, is defined by $Env \times Alias$, where

$$\begin{array}{lcl} Env & = & \mathcal{P}(\mathit{CPA}) \text{ with deterministic} \\ \mathit{CPA} & = & \mathit{VarId} \times (\mathbb{Z} \cup \{\top\}) \\ Alias & = & \mathcal{P}(AP \times AP) \text{ with reflexive, symmetric, transitive} \end{array}$$

An abstract value is $\langle env, alias\rangle$ where env is a set of pairs $\langle d, l\rangle$, d is an access path, l is an integer if the value of the access path is known to be constant l, or $\top$, otherwise; $alias$ is the set of access paths that may be aliased, according to a flow-insensitive information, i.e., the domain $Alias$ is immutable.

Consider the following example procedure:

```
void cpex(T p, T q, int y) {
  if (?) { p.f = y + 5; } else { q.f = 42; }
}
```

The procedure `cpex` takes two parameters of type `T`, and an integer parameter. We assume that the type `T` has an integer field f. The procedure assigns the field `f` for the object pointed to by either `q` or `p`.

An abstract transformer for *CPA* is defined by

$$\begin{array}{l} tr_{Env\times Alias}(\langle env, alias\rangle) = tr_{Env}(env) \times \{alias\} \\ tr_{Env}(env) = \{\bigcup_{\langle d,l\rangle \in env} tr_{\mathit{CPA}}(\langle d, l\rangle)\} \end{array}$$

and $tr_{\mathit{CPA}}(\langle d, l\rangle)$ is a conditional micro-transformer that depends on the statement.

The micro-transformers for `q.f = 42` and `p.f = y + 5` are:

$$tr_{\mathit{CPA}}^{\texttt{q.f = 42}}(\langle d, l\rangle) = \begin{cases} \langle d, 42\rangle & d = q.f \\ \langle d, 42\rangle & d \neq q.f \wedge \pi_2(d) = f \wedge \langle q, \pi_1(d)\rangle \in alias \wedge l = 42 \\ \langle d, \top\rangle & d \neq q.f \wedge \pi_2(d) = f \wedge \langle q, \pi_1(d)\rangle \in alias \wedge l \neq 42 \\ \langle d, l\rangle & \text{otherwise} \end{cases}$$

$$tr_{\mathit{CPA}}^{\texttt{p.f = y + 5}}(\langle d, l\rangle) = \begin{cases} \emptyset & d = p.f \\ \langle d, l\rangle, \langle p.f, l+5\rangle & d = y.\epsilon \\ \langle d, l\rangle & d \neq p.f \wedge d \neq y.\epsilon \wedge \pi_2(d) = f \wedge \\ & \langle p, \pi_1(d)\rangle \in alias \wedge l \neq \top \wedge \langle y, l-5\rangle \in env \\ \langle d, \top\rangle & d \neq p.f \wedge d \neq y.\epsilon \wedge \pi_2(d) = f \wedge \\ & \langle p, \pi_1(d)\rangle \in alias \wedge l \neq \top \wedge \langle y, l-5\rangle \notin env \\ \langle d, l\rangle & \text{otherwise} \end{cases}$$

Note that the micro-transformers use the query parameters $alias$. Also, the micro-transformer $tr_{\mathit{CPA}}^{\texttt{p.f = y + 5}}$, which defines the effect on the elements of env, uses env as a query parameter. This kind of dependency is supported by our composition algorithm, because *CPA* is invertible.

If the input access path d may be aliased with $p.f$, then the assignment `p.f = 42` may modify the value of d in env, denoted by l. If l is the same as the value of $y - 5$ in env, the new value of d is l. Otherwise, the result is $\top$. Note that we do not need to know what the value of y is, only whether it is the same as $l - 5$ or not.

6. Prototype Implementation

We have implemented a prototype as a proof-of-concept for our approach. This prototype is capable of analyzing Java programs and computing procedure summaries. We used our prototype to compute summaries of several small benchmarks.

The goals of our experiments are: (i) to validate the correctness of the derived summaries; (ii) to evaluate the sizes of summaries in practice, and in particular check whether summaries grow exponentially.

We have integrated our algorithm into the analysis framework of (Fink et al. 2006). We use this to drive our summary computation. For some examples, we used this to validate the results of summarization against results of a whole-program analysis.

The heart of our implementation is the symbolic composition algorithm of Section 4. For conditional micro-transformers, we implemented an incremental version of Fig. 5. This algorithm requires non-trivial consistency checking and simplification of formulas. Effectively, the procedure *simplify-term* and *simplify-pre* are implementing consistency checkers specialized for the typestate domain with must information for access paths of length up to 1.

The experiments described in Table 3 were used as a preliminary evaluation of our composition algorithm and for studying the behavior of summaries. We only report a narrow view of the results that is indicative of the maximal sizes of summaries for our benchmarks.

The first three benchmarks in the table are small examples: the running example, a recursive example, and an example of simple composition. Next is the library of Ganymed SSH-2 for Java, followed by benchmarks from the The Ashes suite. Every row in the table corresponds to the analysis of a benchmark with a typestate property. The typestate property describes the correct behavior of the type shown in the table.

For every experiment, we report the number of procedure nodes (i.e., nodes in the call graph) summarized (sum), the number of procedure summarized into skip (skipsum). We refer to the number of compositions used to create a summary as the *rank* of the summary, and also report the maximal number of compositions used to create a summary (MR). We only report data for procedure summaries, and not for intermediate summaries created during the analysis of

bench	property	sum	skip sum	MR	Typestate			Must		
					cases @MR	avg. pre	max #case	cases @MR	avg. pre	max #case
Running	filecomp	15	8	9	13	2.92	13	32	5.16	32
Recursive	filecomp	15	7	6	6	1.67	6	5	1.6	5
Simple	filecomp	16	8	6	6	1.67	6	5	0.4	5
Ganymed	socket	2353	2092	11	704	8.55	704	6	1.83	6
	transmgr	2356	2102	37	145	7.23	145	11	2	11
	session	2356	2133	4	4	1	4	1	1	1
jlex	stack	737	643	5	3	0.66	3	1	0	1
	printstr	697	593	7	40	5.25	64	1	0	1
	enum	733	632	19	268	5.11	272	1	1	1
rhino	printstr	1266	1056	9	512	9	512	1	0	1

Table 3. Sample experimental results.

a procedure. We also report the maximal number of cases in summaries of the maximal rank (cases @MR), and the average number of terms in their preconditions (avg. pre). These are the dominant factors in the size of our summaries, and should give an impression of the maximal summaries used in a program. In addition to these, we also report the maximal number of cases observed in a summary of any rank (max#case).

Our experiments show that our composition algorithm can be used to successfully summarize procedures of real (albeit small) programs. A more subtle point is that they also show that summaries can sometime decrease in size (in this table, only for JLex).

For some of our benchmarks, the required must alias information is rather limited. For example, for the `Stack` type in JLex, only a single instance is created, and it is used without aliases, and never passed as a parameter. Similar simple usage patterns are observed for `Session` in Ganymed.

There are interesting trade-offs between the cost of simplification and the sizes of summaries, as well as many other implementation details. These are beyond the scope, and space limitations, of this paper.

The prototype is not yet engineered to compete with the optimized implementation of (Fink et al. 2006). We plan to develop a robust implementation of our composition algorithm that will enable us to compare modular analysis to (Fink et al. 2006). In practice, a combination of symbolic summaries and explicit input-output tables may be used for optimizing performance.

7. Related Work

Interprocedural and modular analyses General approaches to interprocedural analysis are described in (Sharir and Pnueli 1981; Cousot and Cousot 1978). Already in (Cousot and Halbwachs 1978), it is shown that a procedure's effect can be computed using linear-relation analysis. In (Cousot and Cousot 2002), the general concept of *symbolic relational separate analysis* is described, but does not provide languages for expressing procedure summaries. Our approach could be thought of as a realization of this concept providing specialized languages for expressing procedure summaries.

Precise and efficient interprocedural dataflow analysis algorithms are presented in (Reps et al. 1995; Sagiv et al. 1996b) for special classes of problems. IFDS problems (Reps et al. 1995) can be encoded in our approach, by representing each flow fact as a propositional parameter. IDE problems (Sagiv et al. 1996b) can be encoded by representing each symbol of an environment as a basic parameter.

In contrast to (Sagiv et al. 1996b), we consider in Section 5.2.4 the problem of linear constant propagation in presence of aliasing and in a modular setting, where the variables in the rest of program are unknown. It is not clear whether this problem can be encoded in the IDE framework (in a naïve encoding, the set of symbols would not be finite). Precise and efficient interprocedural analysis of (Müller-Olm and Seidl 2004), specialized for finding all affine relationships between program variables, subsumes the problem of linear constant propagation considered in (Sagiv et al. 1996b), but does not deal with aliasing.

It is challenging to compute precise procedure summaries for an arbitrary calling context. To enable modular analysis, many analyses compute approximate summaries. A common approach is to analyze the procedure in a *symbolic* context. For example, (Ball et al. 2005) introduces auxiliary variables to record the input values of the procedure, and uses predicates defined by both the program variables and the auxiliary variables. Then, the result of the analysis can be interpreted as a relation between the auxiliary variables, which denote input values, and the output values. However, the predicates might not be expressive enough to capture the precise summary.

In interprocedural analysis based on pushdown systems, e.g., (Reps et al. 2005), summaries are created as a byproduct of an analysis.

Another approach is to specialize the summary generation for a particular problem, to discover which contexts are relevant. For example, Xie and Aiken (2005) create summaries for checking correct use of locks. They use a SAT procedures to enumerate all the relevant calling contexts.

Recently, Gulwani and Tiwari (2007) introduced a method for generating precise procedure summaries in the form of constraints on the input variables of the procedure that must be satisfied for some appropriate generic assertion involving output variables of the procedure to hold at the end of the procedure. Their method is based on computing weakest preconditions of a generic assertion. To guarantee termination of the analysis, they preform second-order unification to strengthen and simplify the weakest preconditions. Our composition algorithm computes weakest preconditions of certain queries, as described in Section 4.2, and relies on simplification, but does not use strengthening.

Modular Pointer Analyses Our work has been motivated by modular points-to analysis of (Chatterjee et al. 1999). Their work infers distinct relevant contexts and computes precise information for each such context, parametric in aliasing between abstract locations at the entry of a procedure. This information can be expressed using symbolic summaries with query parameters on *alias* and *non-alias*. We add to these ideas in several ways: (i) we formulate the notion of concise summaries for a more general class of summaries, going beyond the points-to setting; (ii) we add the property that our summaries are as precise as re-analyzing the procedure, and this is a fundamental requirement that drives our treatment of summaries; (iii) we introduce a framework that allows us to generate concise summaries for a class of abstract domains and transformers.

The modular pointer analysis of (Cheng and Hwu 2000) is an adaptation of (Chatterjee et al. 1999) to work with the entire C language. From our perspective, the most notable difference is the fact that (Cheng and Hwu 2000) uses access paths to express context conditions where (Chatterjee et al. 1999) only uses conditions on aliasing between abstract locations at the entry of a procedure.

Modular pointer analyses in (Whaley and Rinard 1999; Salcianu 2006) make best-case aliasing assumptions at the entry of a procedure, but in order to be sound, compute an approximation of the default semantics by not performing strong updates (except in special cases, see (Salcianu 2006)). The summary is later specialized for a given aliasing situation at a call site. Because the summary computation is based on approximation of default semantics, the summary could be less precise than an (ideal) non-modular analysis.

Typestate Verification DeLine and Fähndrich (2004,2002) present a type system for typestate properties for objects. They can assume *must-alias* properties for a limited program scope, and thus apply strong updates allowing typestate transitions. Their approach is limited to programs for which an expression can be assigned a unique type at a given program point. As opposed to that, our modular typestate analysis is context-sensitive, matching the precision of (Fink et al. 2006). In general, our approach for designing modular analysis is an alternative for using type systems for the same domain.

8. Conclusions and Future Work

For proving non-trivial properties of programs, program analyses use expressive abstract domains. We aim at improving scalability of these analyses, without compromising the precision, by using precise, concise and efficient procedure summaries. In this paper, we have described a class of complex abstract domains, for which we can generate summaries with the desirable properties, thereby advancing the state-of-the-art in this area.

The focus of this work has been the automatic composition of summaries (with precision guarantees). This is a key step, but not the only one, in making modular analysis practical. In future work, we plan to explore techniques that can trade-off precision for scalability (e.g., as done in (Chatterjee et al. 1999) by limiting sizes of conditions).

Finally, the expressivity of micro-transformers is inherently limited. Nevertheless, we believe that many interesting problems can be encoded using micro-transformers. We plan to further investigate the expressivity of our framework by applying it to additional problems, such as modular pointer analysis.

Acknowledgments

We thank Mooly Sagiv, Noam Rinetzky, Tal Lev-Ami, and the anonymous reviewers for their comments on earlier drafts of this paper.

References

T. Ball, T. D. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. *ACM Trans. Program. Lang. Syst.*, 27(2):314–343, 2005.

R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *POPL*, pages 133–146, 1999.

B.-C. Cheng and W.-M. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI*, pages 57–69, 2000.

P. Cousot and R. Cousot. Modular static program analysis. In *CC*, pages 159–178, 2002. ISBN 3-540-43369-4.

P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL*, 1977.

P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *Formal Descriptions of Programming Concepts, (IFIP WG 2.2, St. Andrews, Canada, August 1977)*, pages 237–277. North-Holland, 1978.

P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68, 2002.

R. DeLine and M. Fähndrich. Adoption and focus: Practical linear types for imperative programming. In *PDLI*, pages 13–24, June 2002.

R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, pages 465–490, 2004.

N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *ISSTA*, 2004. URL `http://doi.acm.org/10.1145/1007515`.

J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In *SAS*, pages 439–462, 2003.

S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ISSTA*, pages 133–144, 2006.

J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI*, pages 1–12, 2002.

Ganymed SSH-2 for Java. Ganymed SSH-2 for java. http://www.ganymed.ethz.ch/ssh2/.

S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In *ESOP*, pages 253–267, 2007.

R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *POPL*, pages 339–350, 2007.

M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, pages 330–341, 2004.

G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.

T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.

N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *Proc. Static Analysis Symp.*, 2005.

M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1-2): 131–170, 1996a. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(96)00072-2.

M. Sagiv, T. W. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1&2):131–170, 1996b.

A. Salcianu. *Pointer Analysis for Java Programs: Novel Techniques and Applications*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 2006.

M. Sharir and A. Pnueli. Two approaches to interprocedural data ow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1): 157–171, 1986.

The Ashes suite. The ashes suite. http://www.sable.mcgill.ca/ashes/.

J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA*, pages 187–206, 1999.

Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL*, pages 351–363, 2005.