CONCERTO: A Framework for Combined Concrete and Abstract Interpretation

JOHN TOMAN, University of Washington, USA
DAN GROSSMAN, University of Washington, USA

Abstract interpretation promises sound but computable static summarization of program behavior. However, modern software engineering practices pose significant challenges to this vision, specifically the extensive use of frameworks and complex libraries. Frameworks heavily use reflection, metaprogramming, and multiple layers of abstraction, all of which confound even state-of-the-art abstract interpreters. Sound but conservative analysis of frameworks is impractically imprecise, and unsoundly ignoring reflection and metaprogramming is untenable given the prevalence of these features. Manually modeling framework behaviors offers excellent precision, at the cost of immense effort by the tool designer.

To overcome the above difficulties, we present CONCERTO, a system for analyzing framework-based applications by soundly combining concrete and abstract interpretation. CONCERTO analyzes framework implementations using concrete interpretation, and application code using abstract interpretation. This technique is possible in practice as framework implementations typically follow a single path of execution when provided a concrete, application-specific configuration file which is often available at analysis time. CONCERTO exploits this configuration information to precisely resolve reflection and other metaprogramming idioms during concrete execution. In contrast, application code may have infinitely many paths of execution, so CONCERTO switches to abstract interpretation to analyze application code. CONCERTO is an analysis *framework*, and can be instantiated with any abstract interpretation that satisfies a small set of preconditions. In addition, unlike manual modeling, CONCERTO is *not* specialized to any specific framework implementation. We have formalized our approach and proved several important properties including soundness and termination. In addition, we have implemented an initial proof of concept prototype of CONCERTO for a subset of Java, and found that our combined interpretation significantly improves analysis precision and performance.

CCS Concepts: • **Software and its engineering → Automated static analysis**; **Frameworks**;

Additional Key Words and Phrases: abstract interpretation, state separation, reflection, framework-based applications, metaprogramming

ACM Reference Format:
John Toman and Dan Grossman. 2019. CONCERTO: A Framework for Combined Concrete and Abstract Interpretation. *Proc. ACM Program. Lang.* 3, POPL, Article 43 (January 2019), 50 pages. https://doi.org/10.1145/3290356

1 INTRODUCTION

Modern applications are no longer batch jobs that use only simple data structures and a small set of standard libraries. Instead, to improve productivity and portability, software engineers increasingly rely on large, complex libraries and frameworks to provide the scaffolding on which an application is built. To maximize flexibility, these frameworks are highly configurable and often use

Authors' addresses: John Toman, University of Washington, USA, jtoman@cs.washington.edu; Dan Grossman, University of Washington, USA, djg@cs.washington.edu.

2475-1421/2019/1-ART43
https://doi.org/10.1145/3290356

```
 1 // FRAMEWORK
 2 AppContext init(String configFile) {
 3   XMLDocument config = XML.parseFile(configFile);
 4   AppContext ctxt = ...;
 5   for(Node n : config.getNode("delegates"))
 6     ctxt.delegates.put(n.get("name"), n.get("class"));
 7   // ...
 8   ctxt.entryPoint = config.getNodeString("entryPoint");
 9   return ctxt;
10 }
11 Object delegate(AppContext ctxt, String delegateName) {
12   String delegateClass = ctxt.delegates.get(delegateName);
13   Object d = Class.forName(delegateClass).newInstance();
14   return d.getClass().getMethod("handle").invoke(d);
15 }
16 void main() {
17   AppContext ctxt = init("conf.xml");
18   Class.forName(ctxt.entryPoint).getMethod("start").invoke(ctxt);
19 }
```

```
 1 // APPLICATION
 2 void start(AppContext ctxt) {
 3   while(true) {
 4     String request = Network.accept();
 5     Object resp;
 6     if(request == "ping") {
 7       resp = delegate(ctxt, "ping");
 8     } else {
 9       resp = delegate(ctxt, "pong");
10     }
11     Network.send(resp);
12   }
13 }
```

Fig. 1. A motivating example demonstrating the difficulty in analyzing framework applications. Specifically, key control-flow decisions in `delegate` and `main` depend on the contents of `conf.xml`.

metaprogramming, reflection, embedded DSLs, etc. This increased flexibility comes at the cost of precision and/or soundness when using abstract interpretation or other static analysis techniques to reason about the behavior of framework-based applications.

For example, consider the invented code fragment in Fig. 1, which exemplifies code patterns commonly found in frameworks. Here `start` is part of the application, whereas `delegate`, `init`, and `main` are provided by the framework. The framework first calls `init`, which parses an application-specific XML configuration file, then constructs an `AppContext` to hold the framework state. This state consists of a `delegates` map, which maps names found in the configuration file to corresponding class names, including the application's entry point. The framework invokes the application's entry point on line 18. When the application calls `delegate` from `start`, the framework uses `delegates` to instantiate the class associated with the `delegateName` argument and then reflectively invokes the `handle()` method on the newly constructed object. Thus, the callees on lines 14 and 18 are determined entirely by the configuration file, which is opaque to standard call-graph construction algorithms.

To analyze `delegate` and the reflective method invocation on line 18, analysis authors can choose to: 1) unsoundly ignore the reflective call, 2) be extremely imprecise, e.g., by allowing reflective calls to resolve to *any* method, or 3) based on the contents of `conf.xml`, build an application-specific model of the framework behavior. Option 1 misses most of the application's behavior, yielding many false negatives; in Fig. 1, *none* of the application will be analyzed. Option 2 has the opposite problem: many false positives and infeasible control-flow paths. Finally, option 3 requires significant manual effort. Although this effort can be alleviated with framework-specific model generators [Sridharan et al. 2011], creating a model generator for a framework is itself a monumental task.

Other analysis techniques also struggle with framework-based applications. Given a concrete configuration file, the framework methods in Fig. 1 follow only one execution path, suggesting a partially-concrete approach, such as concolic execution [Godefroid et al. 2005; Sen and Agha 2006; Sen et al. 2005]. However, the infinite "accept" loop in `start` is challenging even for state-of-the-art concolic executors. Finitization of the loop can yield false negatives, and some execution engines may fail to terminate. In contrast, a static analysis built using the monotone framework [Kam and Ullman 1977] or abstract interpretation [Cousot and Cousot 1977, 1979b, 1992b] can soundly approximate the infinite loop by computing a least fixed point over a series of equations.

A key insight of our approach is that many applications and frameworks follow this pattern: framework implementations are difficult to analyze statically, but large parts are *statically executable* given a concrete, application-specific configuration file. Statically executable refers to a program fragment that: a) can be completely and deterministically evaluated at analysis time, and b) will yield the same program state after evaluation at both runtime and analysis time; the `init` method is an example of statically executable code. Conversely, application code contains unbounded control-flow paths. As a result, a one-size-fits-all approach to program analysis is unwise for framework-based applications.

We present Concerto, a system for soundly combining *mostly*-concrete interpretation, an extension to concrete interpretation we introduce and formalize in this paper, and abstract interpretation. By combining these two techniques, Concerto leverages the strengths of both approaches while avoiding their weaknesses. Concerto analyzes framework implementations using mostly-concrete interpretation, and application code using abstract interpretation. Mostly-concrete interpretation supports nondeterminism and over-approximation of sets of values, ensuring our combined interpretation is sound while still terminating.

Our combined interpreter is itself an abstract interpreter that operates over a combined domain of abstract and mostly-concrete states. By formalizing our combined approach within the theory of abstract interpretation (AI) we can directly use techniques from the AI literature to prove soundness, termination, etc. Concerto differs from partial evaluation, as the abstract and concrete interpreters may yield into one another on demand, which allows greater concrete execution within framework code (we illustrate this point further in Section 2). Concerto is *analysis agnostic* and can be used with any analysis that satisfies a modest set of conditions. It is *provably sound*: integrating any sound abstract interpretation that satisfies these conditions into Concerto yields a sound, combined analysis. In addition, we have shown that abstract interpretations that satisfy a small, additional set of conditions can *provably* expect equal or greater precision when used with Concerto.

Key to our approach is the observation that framework code does not directly manipulate application state and vice versa. This *state separation* allows Concerto to partition a program state into two disjoint representations: a mostly-concrete representation used to model the framework state, and an abstract representation for application state. The mostly-concrete component of Concerto may therefore manipulate its portion of the program state while remaining agnostic about the abstract representation used by the abstract interpretation component, and vice versa.

We have implemented an initial proof of concept of Concerto for a subset of Java. We have demonstrated the flexibility of Concerto by incorporating three different analyses with different abstract domains into this prototype. We found that using Concerto significantly increased precision across all analyses when applied to a difficult-to-analyze framework implementation.

2 OVERVIEW

We will first informally describe how Concerto operates on the program in Fig. 2, which is written in a simple language we will call Map. Among other features, Map supports I/O, reflection, maps which are sufficient to illustrate the core of our technique. This program and language will also serve as our running example as we formalize our approach in the remainder of this paper. We will later formalize a language parameterized by base constants and operations into which we can embed the Map language.

In Fig. 2 the framework code is in the left column and the application code in the right. On lines 2–9, the framework opens an (application-provided) configuration file, and creates a map `m` from application-specific identifiers to procedure names. The application start point, `s`, is then called with `m` as its argument. The only other framework code is `dispatch`, which uses the map produced in `main` to look up a procedure name associated with `k` and invoke the named method. The `invoke` intrinsic calls the procedure named by the first argument with the remaining arguments. The application-specific logic is implemented in the procedures `s`, `f`, `g`, `h`, and `i`.

```
proc main() {
  f = open("config");
  m = empty;
  k = read f;
  while(k != "") {
    v = read f;
    m = set m k v;
    k = read f;
  }
  // [m ↦ ["b" ↦ "f", "a" ↦ "i"]]
  s(m);
}
proc dispatch(k, arg, m) {
  // from g:
  // [m ↦ ["b" ↦ "f", "a" ↦ "i"], arg ↦ {+}]
  // from h:
  // [m ↦ ["b" ↦ "f", "a" ↦ "i"], arg ↦ {−}]
  callee = get m k
  invoke(callee, arg, m);
}
```

```
config
b
f
a
i
```

```
proc s(m) {
  x = ★;
  while(x >= 0) {
    g(x, m);
    x = ★;
  }
  h(x, m);
}
proc g(p, m) {
  p = p + 1;
  // [m ↦ ["b" ↦ "f", "a" ↦ "i"], p ↦ {+}]
  dispatch("b", p, m);
}
proc f(p, m) {
  if(p <= 0) {
    error();
  } else {
    print(p);
  }
}
proc h(q, m) {
  dispatch("a", -4, m);
}
proc i(q, m) {
  print(q);
}
```

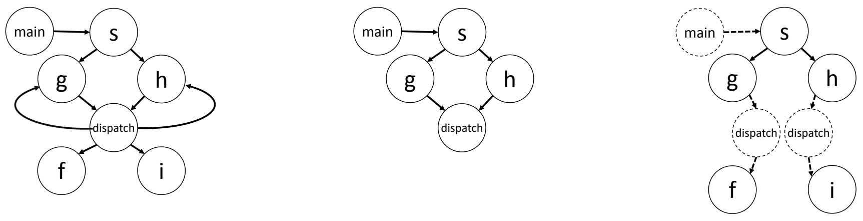
Fig. 2. Motivating example. The framework implementation (`main` and `dispatch`) uses many of the same implementation idioms found in Fig. 1. The comments in green show the abstract and mostly-concrete states that reach those program points during combined execution.

Although the program, framework, and language are significantly simpler than the full Java language and the example shown in Fig. 1, they together pose many of the same challenges. In particular, the reflective operation in `dispatch` depends on information in a configuration file stored in a map. Many difficult-to-analyze framework features, such as dependency injection [Fowler 2004], or Android Intents [Barros et al. 2015], follow a similar pattern.

This program exemplifies the *state separation* hypothesis, which requires that the application state is opaque to the framework implementation and similarly for framework state and application code. Thus, the application state (in this case, the integer arguments to `dispatch`) are not interrogated or manipulated by the framework, only threaded through the `dispatch` procedure and then passed back into application procedures. Similarly, the framework state encapsulated in the dispatch map `m` is not directly manipulated by the application, only threaded through application code to calls back into `dispatch`.

In our experience, this hypothesis applies to most real-world implementations; to promote reusability, applications and frameworks rarely directly manipulate each other's state, instead communicating via method calls on opaque interface types. To validate this hypothesis, we performed an informal evaluation of two large Java web frameworks, Spring[1] and Struts.[2] The majority of framework state is contained in non-`public` object fields. We found that across the two frameworks only 0.5% (64/12304) of fields had `public` visibility and thus the application cannot mutate the vast majority of framework state. Our experience with other frameworks suggests this pattern is the norm. Similarly, frameworks do not directly mutate application state; to maximize flexibility, frameworks avoid depending on predetermined field names or class layouts, instead relying on well

[1]https://spring.io/
[2]https://struts.apache.org/



(a) Sound but Imprecise Call-Graph (b) Unsound Call-Graph (c) Call-Graph with Concerto

Fig. 3. Call graphs produced by different analysis schemes. In (a), calls from `dispatch` to `main` and `s` can be ruled out by matching argument arities. In (c), procedures executed (mostly-)concretely are given a dashed outline.

defined interfaces to communicate with the application. Together, these facts suggest that modern Java framework implementations are a natural fit for our state separation hypothesis.

Concerto exploits this state separation to thread abstract values produced by the abstract interpreter through concrete interpretation and the abstract interpretation may do the same for concrete values produced by concrete interpretation. (Concerto also includes support for the rare cases where this hypothesis does not apply, see Section 8.4.) Section 3.1 formalizes a type-based state separation that is natural in languages like Java.

Without additional knowledge about the program in Fig. 2, a standard abstract interpretation not integrated with Concerto must make worst-case assumptions about `read`, and thus use an extremely imprecise abstraction of the framework state in `m`. As a result, analysis of `dispatch` would conclude that `invoke` may call any procedure. Thus, plain abstract interpretation cannot rule out that a negative argument may flow from `h` through `dispatch` to `f` and that the `error()` statement is reachable. On the other hand, ignoring `invoke` as though it is a no-op ignores important application behavior. These two situations are illustrated in Figs. 3a and 3b, respectively.

Suppose now that we have the following domain knowledge about our program:

(1) The contents of the file `config` are available at analysis time and will not change between analysis time and program runtime.
(2) `config` has the contents shown in Fig. 2

This information ensures that `error()` is never executed: `dispatch` will always call `f` with the positive argument passed to it by `g`. However, even if an abstract interpretation has this information, verifying that `error()` is unreachable requires an extremely precise semantics and representation for maps. This can be achieved in this simple language, but, in practice, frameworks use much more complicated data structures and abstractions, making precise analysis via pure abstract interpretation unlikely. In contrast, Concerto integrated with a simple signedness analysis can prove that the `error()` statement is unreachable, using a process we briefly sketch below.

2.1 Analyzing the Example

Concerto begins analysis of the program by concretely executing `main()`. Due to the domain-specific knowledge described above, the initialization loop is statically executable. Thus, Concerto opens the file `"config"` and runs the loop to completion. When the loop terminates, `m` holds the map `["b"` $\mapsto$ `"f", "a"` $\mapsto$ `"i"]`. We stress that Concerto uses no application- or analysis-specific logic here: Concerto simply performs concrete interpretation, opening the listed file and executing the loop.

At the call to the application entry point `s` on line 11, Concerto switches to abstract interpretation, in this example a signedness analysis. A key assumption of Concerto is that framework code, once given a concrete configuration, is almost entirely statically executable. In contrast, application code may deal with nondeterministic inputs, giving rise to unbounded loops like the one on lines 23–26. Although concrete execution will naturally fail to terminate on the loop in `s`, abstract interpretation can easily over-approximate the loop.

When switching to abstract interpretation, Concerto transforms the concrete program state at the call-site to the abstract representation used by the abstract interpreter. We describe this process in more detail in Section 4.3. In this example, the signedness analysis begins in the abstract state: $[m \mapsto [\texttt{"b"} \mapsto \texttt{"f"}, \texttt{"a"} \mapsto \texttt{"i"}]]$. The abstract interpreter has *not* abstracted away the framework state; instead, the signedness analysis has reused the concrete value directly for the value of m. However, the abstract interpreter does *not* need to implement concrete map semantics or otherwise "understand" this representation. Due to the state separation hypothesis described above, any map operations on `m` occur in framework code, which is *not* analyzed using abstract interpretation.

In the while loop of `s`, the signedness analysis analyzes the call to `g`, which itself calls the framework's `dispatch` procedure in the abstract state $[m \mapsto [\texttt{"b"} \mapsto \texttt{"f"}, \texttt{"a"} \mapsto \texttt{"i"}], p \mapsto \{+\}]$. For this call, Concerto switches from abstract interpretation to *mostly*-concrete interpretation. Concerto *cannot* switch back to fully-concrete interpretation soundly because the above abstract state cannot be concretized to a single concrete state (or even a finite set of states): p abstracts the infinite set of all positive integers. To avoid materializing an infinite set of concrete states, our mostly-concrete interpretation supports runtime values that are abstractions of infinite sets of values. To represent the infinite set of possible values of `p`, the mostly-concrete interpreter reuses `p`'s abstract value in the abstract interpreter, i.e., $\{+\}$. Concerto is agnostic to the domain of abstract values; had the analysis chosen instead to represent integers with, say, intervals, the mostly-concrete interpreter would also use intervals. As with the embedding of concrete maps into abstract states, the mostly-concrete interpreter does not need semantics over the signedness domain: integer operations on the application state only occur in application code which is *not* executed mostly-concretely.

As the value of `m` in the abstract caller state is $[\texttt{"b"} \mapsto \texttt{"f"}, \texttt{"a"} \mapsto \texttt{"i"}]$, this value can be directly reused in the callee mostly-concrete state. Hence, mostly-concrete interpretation has no problem concretely evaluating this call to `dispatch` with `"b"` and `m` to determine that control should transfer back to the application by calling `f` with `p`.

At this final call back into the application, Concerto once again switches to abstract interpretation, transforming the mostly-concrete state into an abstract state. The value of `arg` in the mostly-concrete state is $\{+\}$, which becomes the value of `p` in the abstract interpreter. Using this abstract value, the abstract interpreter can prove the true branch of the conditional is never taken. Analysis of the call to `h` on line 27 proceeds similarly, again using mostly-concrete interpretation to precisely resolve the `dispatch` call. The final call-graph used during combined interpretation is shown in Fig. 3c.

The above process bears many similarities to partial evaluation [Futamura 1999; Mogensen 1995] where the configuration file is treated as a static input. A sufficiently powerful partial evaluator that supports metaprogramming (e.g., [Sullivan 2001]), could produce a residual program from Fig. 2 that a signedness analysis could verify. However, suppose `g` used dynamic input to choose between `"b"` and another procedure name as the argument to `dispatch`. In this scenario, the partial evaluator would *not* fully reduce the body of `dispatch`, making the signedness analysis imprecise. In contrast, under Concerto, provided the abstract interpretation faithfully tracks the two possible procedure names, the mostly-concrete interpreter can precisely resolve the `invoke` operation to the two potential callees.

2.2 Paper Outline

The remainder of this paper formalizes and elaborates on the process sketched above. Section 3 defines an intraprocedural language we will use and extend throughout this paper. Our formal

$$\begin{array}{rcl@{\qquad}rcl}
\ell & ::= & \ell_a \mid \ell_f & \mathit{prog} & ::= & (\mathit{fstmt}^{\ell} \mid \mathit{astmt}^{\ell})^{*} \\
\mathit{astmt}^{\ell} & ::= & \ell_a : \mathit{astmt} & \mathit{fstmt}^{\ell} & ::= & \ell_f : \mathit{fstmt} \\
\mathit{astmt} & ::= & \texttt{goto}\ \ell \mid x = \mathit{aexp} \mid \texttt{if}\ x \texttt{<=>} y\ \texttt{goto}\ \ell & \mathit{fstmt} & ::= & \texttt{goto}\ \ell \mid x = \mathit{fexp} \mid \texttt{if}\ x \texttt{<=>} y\ \texttt{goto}\ \ell \\
\mathit{aexp} & ::= & y \mid bc_a \mid \mathit{aop}(v_1, \ldots, v_n) & \mathit{fexpr} & ::= & x \mid bc_f \mid \mathit{fop}(v_1, \ldots, v_n)
\end{array}$$

Fig. 4. Intraprocedural grammar, parameterized by language-specific choices for bc_f, *fop*, bc_a, and *aop*.

language makes explicit the state separation hypothesis. We define a concrete semantics for this language against which we prove Concerto sound, and describe the expected definition of abstract interpreters.

Section 4 describes our main contribution: the combination of concrete and abstract interpretation. Section 4.1 demonstrates how a naïve combination of concrete and abstract interpretation yields a sound basis for combined interpretation, but one that is impractical due to the difficulty of concretizing abstractions of infinite sets of values, as was the case with $\{+\}$ above. Section 4.2 defines *mostly*-concrete interpretation which can handle abstractions of infinite sets of values. Section 4.3 shows how combining abstract and mostly-concrete interpretation yields a sound interpretation, and formalizes how mostly-concrete states are transformed into abstract states, and vice versa. Finally, Section 4.4 defines a set of sufficient conditions for Concerto to match or exceed the precision of an abstract interpretation.

Section 5 briefly discusses how we extend our formalism to procedures. Section 6 describes a particular iteration strategy that is natural in practice and Section 7 sketches how we ensure termination (while retaining soundness) under this iteration strategy using widening. We then discuss our prototype implementation (Section 8), and the results of initial case studies (Section 9). We close with a discussion of related work (Section 10) and future work (Section 11).

3 PRELIMINARY DEFINITIONS

3.1 Language Definition

Our core calculus is a simple imperative language with conditional/unconditional goto, variable assignments, and constants and primitive operations over a set of types. We formalize a *type-based* state separation by partitioning this set of types into application and framework types, and restricting all operations on framework types to framework code and similarly for application types.

The formal grammar is given in Fig. 4. We assume two disjoint families of types $\mathcal{A}$ (for application) and $\mathcal{F}$ (for framework) and, with a slight abuse of notation, will use $\mathcal{A}$ (respectively $\mathcal{F}$) as a metavariable to range over the types in $\mathcal{A}$ (respectively $\mathcal{F}$). Every variable is given a type drawn from one of these families. bc_f and *fop* range over base constants and primitive operations respectively for types in $\mathcal{F}$, and bc_a and *aop* do the same for $\mathcal{A}$. All operations in *fop* have type $(\mathcal{F} \times \ldots \times \mathcal{F}) \rightarrow \mathcal{F}$, and similarly for *aop* and $\mathcal{A}$. The <=> nonterminal ranges over comparison operators.

In our language, ℓ_f label framework statements, and similarly for ℓ_a and application statements. The label of a statement determines what operations may be performed by that statement: *fop*s and bc_f may only appear in code labeled with ℓ_f, and similarly for *aop*, bc_a and ℓ_a. Further, we require that comparisons in statements labeled with ℓ_a can only compare values of types in $\mathcal{A}$, and similarly for ℓ_f and $\mathcal{F}$. However, statements labeled ℓ_f may move values between variables with type $\mathcal{A}$, and vice versa for statements labeled ℓ_a. As such, framework code must treat values of type $\mathcal{A}$ ("application values") opaquely, threading them through to statements labeled ℓ_f. Similarly, statements labeled ℓ_a treat $\mathcal{F}$ values opaquely. This syntactic restriction models the state separation hypothesis described in Section 2. (In practice, this strict separation may be violated by, e.g., primitive types like `int`, library types, etc.; our implementation has special support for these shared types as described in Section 8.4.)

The full operational semantics (omitted for space reasons) are defined in terms of the following denotations and value domains, which we will use throughout the remainder of the paper. Let V_a be the set of all values with a type in $\mathcal{A}$ and similarly for V_f and $\mathcal{F}$. Every *aop* has a denotation $[\![aop]\!] : (V_a \times \ldots V_a) \to V_a$; we assume that *aop*s are deterministic. In contrast, some *fop* operations may produce values that depend on the value of some environment model $\mathcal{E}$, which models nondeterminism due to file contents, network requests, etc. Formally, each *fop* has a denotation $[\![fop]\!] : \mathcal{E} \times V_f \times \ldots \times V_f \to \wp(V_f \times \mathcal{E})$, where $\langle v', E' \rangle \in [\![fop]\!](E, v_1, \ldots, v_n)$ means executing *fop* in environment $E \in \mathcal{E}$ with arguments $v_1, \ldots, v_n$ produces a new environment model E' and result v'. To simplify presentation, we require that denotations are total functions; in practice, we assume that the denotations gracefully handle runtime type errors (e.g., by returning a sentinel error value, halting execution, etc.). Finally, we assume that $[\![\texttt{<=>}]\!]$ denotes into a binary relation over values of the appropriate type, $[\![\texttt{<}\neq\texttt{>}]\!]$ is the negation of $[\![\texttt{<=>}]\!]$, and $[\![bc_f]\!]$ and $[\![bc_a]\!]$ produce values in V_f and V_a respectively corresponding to the interpretation of those constants.

Although the result of *fop*s depend on the current environment E, in some cases we may have *a priori* information such that a seemingly nondeterministic *fop*, i.e., reading a file, is *effectively deterministic*. For example, in Section 2, we exploited domain specific information about runtime contents of the configuration file to precisely execute the initialization loop. We account for this knowledge by allowing hypotheses on the domain of $\mathcal{E}$. For example, if $\mathcal{E}$ models file-system contents, and we have *a priori* knowledge that a file f always has content c at runtime, we can restrict $\mathcal{E}$ to include only models where the file f has contents c.

Throughout the rest of this paper we assume that we are operating on some arbitrary program written in this language and that the relations *pred* and *succ* are defined with the obvious definitions and there is a map *prog* from labels to unlabeled statements.

Example 3.1 (Map Language). We can encode the Map language and state separation of Section 2 in this language framework as follows. Control-flow constructs (`if`/`while`) can be encoded using the `goto` representation defined in Fig. 4; we defer discussion of procedures to Section 5.

We take $\mathcal{A} = \{int\}$, i.e., the type of machine integers, with bc_a defined to be integer constants, *aop* to be the usual arithmetic operations, and V_a as machine integers. We next define $\mathcal{F} = \{File, Str, \mathcal{M}\}$, where $\mathcal{M}$ is the domain of maps from strings to strings, and instantiate *fop* with the following:

$$\texttt{open} : Str \to File \qquad \texttt{read} : File \to Str \qquad \texttt{set} : \mathcal{M} \times Str \times Str \to Str \qquad \texttt{get} : \mathcal{M} \times Str \to Str$$

We define bc_f as the set of literal string constants and $\texttt{empty} : \mathcal{M}$ which is an empty map. Finally, we take $V_f = STR \cup FileContents \cup (STR \to STR)$, where STR is the set of string values and $FileContents$ is a finite stream of STR values. Given the syntactic constraints on where *aop*s and *fop*s may appear, this instantiation encodes that the application may not manipulate the framework dispatch map `m`, nor may the framework manipulate integers received from the application.

Finally, the domain of environment models $\mathcal{E}$ is a map from strings to file contents, i.e., $STR \to FileContents$. We encode the information about the configuration file by requiring that: $\mathcal{E} = \{e \mid e \in STR \to FileContents \land e[\texttt{"config"}] = \langle\texttt{"b"}, \texttt{"f"}, \texttt{"a"}, \texttt{"i"}\rangle\}$ □

3.2 Concrete Properties

Concerto targets abstract interpretations where the approximated concrete property is the set of reaching concrete states that may occur during program execution. In the intraprocedural language described thus far, a concrete state is a valuation for the variables in the program plus an environment model. Formally, a concrete state is defined as: $S = \mathcal{E} \times (X \to V)$ where $V = V_a \cup V_f \cup \{\bot_V\}$, X is the domain of variables appearing in a program, $\mathcal{E}$ is type of environment models described in Section 3.1, and $\bot_V$ is a sentinel "uninitialized" value. We assume the type system is sound, i.e.,

$$F(r)[\ell^\circ \rightsquigarrow \ell^\bullet] = \bigsqcup_{\substack{p \in pred(\ell) \\ in \in r[p^\bullet \rightsquigarrow \ell^\circ]}} step^F(in, \ell) \sqcup \begin{cases} \bigsqcup_{e \in \iota_{\mathcal{E}}} step^F(\langle \iota_S, e\rangle, \ell) & \ell = s \\ \emptyset & o.w. \end{cases}$$

$$F(r)[p^\bullet \rightsquigarrow \ell^\circ] = \begin{cases} \{\langle s, E\rangle | \langle s, E\rangle \in r[p^\circ \rightsquigarrow p^\bullet] \wedge s[x][\![\texttt{<=>}]\!]s[y]\} & prog[p] = \texttt{if x <=> y goto } \ell \\ \{\langle s, E\rangle | \langle s, E\rangle \in r[p^\circ \rightsquigarrow p^\bullet] \wedge (s[x][\![\texttt{<}\neq\texttt{>}]\!]s[y])\} & prog[p] = \texttt{if x <=> y goto } \ell' \\ r[p^\circ \rightsquigarrow p^\bullet] & o.w. \end{cases}$$

$$step^F(\langle in, E\rangle, \ell) = \begin{cases} \{\langle in, E\rangle\} & prog[\ell] = \texttt{if} \ldots \vee \texttt{goto} \ldots \\ \{\langle in[x \mapsto in[y]], E\rangle\} & prog[\ell] = x = y \\ \{\langle in[x \mapsto [\![c]\!]], E\rangle\} & prog[\ell] = x = c \\ \{\langle in[x \mapsto [\![aop]\!](in[v_1], \ldots, in[v_n])], E\rangle\} & prog[\ell] = x = aop(v_1, \ldots, v_n) \\ \{\langle in[x \mapsto r], E'\rangle \mid \langle r, E'\rangle \in [\![fop]\!](E, in[v_1], \ldots, in[v_n])\} & prog[\ell] = x = fop(v_1, \ldots, v_n) \end{cases}$$

Fig. 5. Concrete semantic function. c is any constant of type $\mathcal{A}$ or $\mathcal{F}$ and $[\![c]\!]$ is its corresponding denotation.

for any concrete state s arising during execution: $type(x) \in \mathcal{F} \Rightarrow s[x] \in V_f$ and $type(x) \in \mathcal{A} \Rightarrow s[x] \in V_a$, and that all variables must be defined before use, i.e., a program never observes $\bot_V$.

Next, define a domain of *flow edges* $\mathcal{L}$ as: $\mathcal{L} = \{\ell^\circ \rightsquigarrow \ell^\bullet\} \cup \{p^\bullet \rightsquigarrow \ell^\circ \mid p \in pred(\ell)\}$, where ℓ° and $\ell^\bullet$ respectively denote the "entrance to" and "exit from" ℓ. Elements $\ell^\circ \rightsquigarrow \ell^\bullet$ of $\mathcal{L}$ correspond to the flow of program control through the statement labeled ℓ, whereas an element $p^\bullet \rightsquigarrow \ell^\circ$ corresponds to the flow from some predecessor p to ℓ. When convenient, we will abbreviate $\ell^\circ \rightsquigarrow \ell^\bullet$ as simply ℓ. We will use $\vec{\ell}$ to represent an arbitrary element of $\mathcal{L}$.

Our domain of concrete properties is $R = \mathcal{L} \to \wp(S)$, which forms a complete lattice, defined pointwise over edges; the powerset of states also forms a complete lattice with the usual definitions.

We now define the concrete semantic function, $F : R \to R$, the least fixed-point of which is a complete set of reaching states for a program. That is, for every $\vec{\ell} \in \mathcal{L}$, if $\langle s, E'\rangle \in (lfp\, F)[\vec{\ell}]$ then there is some execution that flows through edge $\vec{\ell}$ yielding $\langle s, E'\rangle$. The full definition of F is given in Fig. 5. We assume program execution begins at a single, distinguished label s. Program start is modeled with the second term in the definition of $F(r)[\ell^\circ \rightsquigarrow \ell^\bullet]$: $\iota_{\mathcal{E}}$ is a set of possible initial environments, and ι_S is a distinguished start state which maps all variables to $\bot_V$.

3.3 Abstract Properties

Concerto is designed to combine mostly-concrete execution with an abstract interpretation defined as follows. We assume that sets of reaching concrete states are over-approximated by the complete lattice $\widehat{S}$ with ordering and least upper bound operator $\sqsubseteq_{\widehat{S}}$ and $\sqcup_{\widehat{S}}$ respectively. (We will use $\sqsubseteq_D$ to indicate a partial order on a domain D.) The domain of abstract properties, $\widehat{R} = \mathcal{L} \to \widehat{S}$, also forms a complete lattice defined by the pointwise extension of $\sqsubseteq_{\widehat{S}}$ and $\sqcup_{\widehat{S}}$. We further assume that the domain $\widehat{R}$ forms a Galois connection with the domain of concrete properties R, defined by the abstraction and concretization functions α_A and γ_A. We use the standard notation $R \underset{\alpha_A}{\overset{\gamma_A}{\leftrightarrows}} \widehat{R}$ to denote this connection. The abstract semantics of the abstract interpretation are given by a monotone abstract semantic function $\widehat{F} : \widehat{R} \to \widehat{R}$. We assume that this function is a sound abstraction of F according to the above Galois connection, i.e., $\alpha_A \circ F \sqsubseteq_{R \to \widehat{R}} \widehat{F} \circ \alpha_A$,[3] whence by [Cousot and Cousot 1979b] we have that $\alpha_A(lfp\, F) \sqsubseteq_{\widehat{R}} lfp\, \widehat{F}$. In other words, the reaching states computed by the least fixed point of F are soundly over-approximated according to α_A by the least fixed point of $\widehat{F}$.

Thus, an abstract interpretation is defined by the 7-tuple $\langle \widehat{R}; \widehat{S}; \widehat{F}; \alpha_A; \gamma_A; \sqcup_{\widehat{S}}; \sqsubseteq_{\widehat{S}}\rangle$ where:

$$\widehat{R} = \mathcal{L} \to \widehat{S} \qquad\qquad R \underset{\alpha_A}{\overset{\gamma_A}{\leftrightarrows}} \widehat{R} \qquad\qquad \alpha_A \circ F \sqsubseteq_{R \to \widehat{R}} \widehat{F} \circ \alpha_A$$

[3] Equivalently, $\alpha_A \circ F \circ \gamma_A \sqsubseteq_{\widehat{R} \to \widehat{R}} \widehat{F}$ or $F \circ \gamma_A \sqsubseteq_{\widehat{R} \to R} \gamma_A \circ \widehat{F}$.

$$\widehat{F}(\widehat{r})[\ell^{\circ} \rightsquigarrow \ell^{\bullet}] = step^{+}(\bigsqcup_{p \in pred(\ell)} \widehat{r}[p], \ell) \sqcup \begin{cases} step^{+}(\bot, \ell) & \ell = s \\ \bot & o.w. \end{cases}$$

$$\widehat{F}(\widehat{r})[p^{\bullet} \rightsquigarrow \ell^{\circ}] = \begin{cases} \widehat{r}[p^{\circ} \rightsquigarrow p^{\bullet}][x \mapsto lop \cap \{0, +\}] & \text{if } prog[s] = \texttt{if } x \texttt{>=} \mathit{0} \texttt{ goto } \ell \wedge lop \cap \{0, +\} \neq \emptyset \\ \bot & \text{if } prog[s] = \texttt{if } x \texttt{<=} \mathit{0} \texttt{ goto } \ell' \wedge lop \cap \{-, 0\} = \emptyset \\ \dots & \\ \widehat{r}[p] & o.w. \end{cases}$$

$$step^{+}(\widehat{in}, \ell) = \begin{cases} \widehat{in}[x \mapsto \widehat{in}[y]] & prog[\ell] = x = y \\ \widehat{in}[x \mapsto \widehat{in}[y]\widehat{[\![aop]\!]}\widehat{in}[z]] & prog[\ell] = x = y \; aop \; z \\ \widehat{in}[x \mapsto \{sign(N)\}] & prog[\ell] = x = N \\ \widehat{in} & prog[\ell] = \texttt{goto} \dots \vee prog[\ell] = \texttt{if} \dots \\ \widehat{in}[x \mapsto \top_F] & prog[\ell] = x = bc_f \vee prog[\ell] = x = fop(v_1, \dots, v_n) \end{cases}$$

Fig. 6. The abstract semantics for our running signedness example. In the above, we use an infix notation for *aop* as we assume all integer operations are binary. In the $step^{+}$ function, N is an integer constant; in the definition of $\widehat{F}$, *lop* is $\widehat{r}[p^{\circ} \rightsquigarrow p^{\bullet}][x]$ and s is the distinguished start label.

Example 3.2 (Signedness Analysis). The abstract state for the intraprocedural signedness analysis discussed in Section 2 is $\widehat{S} = X_f \to \wp(V_f)^{\top} \times X_a \to \wp(\{-, 0, +\})$, and α_A is defined as:

$$\alpha_A(r)[\vec{\ell}] = \left\langle \lambda x : X_f.\{s[x] \mid \langle s, e \rangle \in r[\vec{\ell}] \wedge s[x] \neq \bot_V\},\ \lambda x : X_a.\{sign(s[x]) \mid \langle s, e \rangle \in r[\vec{\ell}]\} \right\rangle$$

In the above definitions, X_f is the set of program variables with type in $\mathcal{F}$, X_a are those with types in $\mathcal{A}$, and *sign* returns one of +, −, or 0 depending on the sign of its argument. We omit the definition of γ_A as it can be derived from the definition of α_A [Cousot and Cousot 1979b]. The domain $\wp(V_f)^{\top}$ is the powerset domain of concrete values, extended with a special $\top_F$ value, which represents any possible value of type $\mathcal{F}$. We omit the definitions of $\sqsubseteq_{\widehat{S}}$ and $\sqcup_{\widehat{S}}$ as they are standard.

The abstract semantic function $\widehat{F}$ is defined in Fig. 6. We have included only the comparison rules necessary to verify the program in Fig. 2. We omit the definitions for $\widehat{[\![+]\!]}$, $\widehat{[\![-]\!]}$, etc. however $\widehat{[\![+]\!]}$ is defined such that $\{0, +\}\widehat{[\![+]\!]}\{+\} = \{+\}$, which again is sufficient to verify the example in Fig. 2. □

4 COMBINED INTERPRETATION

Given the language, semantic functions, and Galois connection defined in Section 3, we can define an initial, naïve attempt at combined interpretation. Intuitively, this strawman combination, which we call Concerto$_0$, analyzes framework code by applying the concrete semantic function F at framework statements and applying the abstract semantic function $\widehat{F}$ at application statements. Concerto$_0$ translates between abstract and concrete states using abstraction and concretization functions. This approach is sound (as proved in the following Section 4.1) but ultimately infeasible to implement as it requires materializing infinite sets of states and values. To overcome this limitation, we extend concrete interpretation to *mostly-concrete* interpretation. Mostly-concrete interpretation avoids materializing infinite sets by using *finite* abstractions of sets of possible values. We then define the combination of mostly-concrete and abstract interpretation used by Concerto and prove it sound. This combination must translate between different state representations. Unlike Concerto$_0$, we do not use explicit abstraction or concretization functions. Instead we formalize *domain transformers* which soundly translate between state domains but are weaker than a Galois connection. We close by proving when Concerto matches or exceeds the precision of plain abstract interpretation.

4.1 Naïve Combination

To motivate the need for mostly-concrete interpretation, we elaborate on the strawman Concerto$_0$ and enumerate why it is impractical as a basis for combined analysis.

Concerto$_0$ executes framework code (statements with label ℓ_f) concretely and abstractly interprets application code (statements with label ℓ_a). Our initial attempt at combined interpretation therefore operates over a combined semantic domain that represents reaching states in the framework with sets of concrete states, and reaching states in the application with the abstract state domain $\widehat{S}$. However, the semantic function F, which models concrete execution, operates over the fully-concrete domain R. Thus, the combined domain is injected into the fully-concrete domain by applying a concretization function to the abstract states of the combined domain. Symmetrically, to abstractly execute application code with the abstract semantic function $\widehat{F}$, the combined domain is injected into the abstract domain by applying an abstraction function to the reaching concrete state component. After injection and applying both semantic functions, Concerto$_0$ combines the concrete results at framework statements and abstract results at application statements.

As the combined interpretation uses the highly precise concrete semantics for framework statements (and therefore *fop*s), Concerto$_0$ can, at least in principle, precisely analyze framework code. The concrete interpreter may also use the hypotheses on runtime environments to gain further precision. For example, if the framework parses a configuration file as in the example of Section 2, the concrete interpreter may simply open and parse the configuration file directly. However, not all *fop*s will be analysis-time deterministic, leading to an explosion in reaching concrete states. Further, Concerto$_0$ relies on an explicit concretization function which cannot be implemented in practice.

We formalize the informal description above as follows. First, we partition the space of $\mathcal{L}$ into two sets, $\mathcal{L}_A = \{\ell_a^\circ \rightsquigarrow \ell_a^\bullet\} \cup \{\ell_a^\bullet \rightsquigarrow \ell^\circ \mid \ell_a \in \mathit{pred}(\ell)\}$ and $\mathcal{L}_F = \{\ell_f^\circ \rightsquigarrow \ell_f^\bullet\} \cup \{\ell_f^\bullet \rightsquigarrow \ell^\circ \mid \ell_f \in \mathit{pred}(\ell)\}$. $\mathcal{L}_A$ are flow edges originating in ℓ_a-labeled statements, and symmetrically for $\mathcal{L}_F$ and ℓ_f. We define this combined domain $\overline{R}_0$ and an abstraction function $\alpha_0 : R \to \overline{R}_0$ as:

$$\overline{R}_0 = \mathcal{L}_F \to S \times \mathcal{L}_A \to \widehat{S} \qquad\qquad \alpha_0(r) = \langle \lambda\vec{\ell} : \mathcal{L}_F.r[\vec{\ell}],\ \lambda\vec{\ell} : \mathcal{L}_A.\alpha_A(r)[\vec{\ell}]\rangle$$

$\overline{R}_0$ is a complete lattice with the standard component-wise definitions of least upper bound and ordering. As α_0 is a monotone, complete join morphism there is some γ_0 such that R and $\overline{R}_0$ form a Galois connection. We further assume that the abstract and concrete states form a Galois connection $S \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \widehat{S}$ and that $\alpha_A = \dot{\alpha_S}$ and $\gamma_A = \dot{\gamma_S}$, where $\dot{f}$ denotes the pointwise extension of f. Using α_S and γ_S, the injection functions described above are defined as:

$$\mathit{inj}_R(\langle m, \widehat{m}\rangle)[\vec{\ell}] = \begin{cases} m[\vec{\ell}] & \vec{\ell} \in \mathcal{L}_F \\ \gamma_S(\widehat{m}[\vec{\ell}]) & o.w. \end{cases} \qquad\qquad \mathit{inj}_{\widehat{R}}(\langle m, \widehat{m}\rangle)[\vec{\ell}] = \begin{cases} \alpha_S(m[\vec{\ell}]) & \vec{\ell} \in \mathcal{L}_F \\ \widehat{m}[\vec{\ell}] & o.w. \end{cases}$$

We can now define Concerto$_0$'s combined interpretation function $C_0 : \overline{R}_0 \to \overline{R}_0$ as:

$$C_0(X) = \left\langle F(\mathit{inj}_R(X))|_{\mathcal{L}_F},\ \widehat{F}(\mathit{inj}_{\widehat{R}}(X))|_{\mathcal{L}_A} \right\rangle \tag{1}$$

Theorem 1. *C_0 is sound, i.e., $\alpha_0 \circ F \sqsubseteq_{R \to \overline{R}_0} C_0 \circ \alpha_A$.*

Proof. By the assumed soundness of $\widehat{F}$, that $\mathit{inj}_{\widehat{R}} \circ \alpha_0 = \alpha_A$ and that $\mathit{inj}_R \circ \alpha_0$ is extensive. □

Theorem 1 establishes C_0 is sound, but that doesn't mean C_0 is a good idea. In fact, it is not. Using C_0 as the basis for combined interpretation is impractical for the following reasons:

(1) *Infinite Sets* In many cases, concretizing an abstract state will yield an infinite number of concrete states. For example, concretizing the abstract state $[m \mapsto [$`"b"` $\mapsto$ `"f"`, `"a"` $\mapsto$ `"i"`$], p \mapsto \{+\}]$ at the call to `dispatch` in Section 2 yields the following set of concrete states:

$$\bigcup_{e \in \mathcal{E}} \{\langle e, [m \mapsto [\texttt{"b"} \mapsto \texttt{"f"}, \texttt{"a"} \mapsto \texttt{"i"}], p \mapsto n]\rangle \mid n > 0\}$$

The concretization operation cannot be implemented in a meaningful way, as materializing this infinite set is clearly impossible.

(2) *Nondeterminism* Not all framework operations will be deterministic, even with *a priori* knowledge about the program's runtime environment. For example, the exact user input entered is impossible to know at analysis time. Instead, combined interpretation based on C_0 would have to enumerate over all possible values produced in all environments which is impractical from an implementation perspective.

(3) *Exponential Explosion in States* Nondeterministic *fop*s whose denotations return multiple possible results will cause exponential explosions in the number of reaching states. For example, if n states reach an *fop* that produces m unique results, the concrete semantics will generate $n \cdot m$ result states. This problem is similar to the exponential state explosion common in symbolic execution.

Thus, instead of combining abstract and concrete interpretation, Concerto combines abstract and *mostly*-concrete interpretation, which addresses the above limitations. Section 4.2 describes the semantics of mostly-concrete interpretation, Section 4.3 describes how to combine abstract and mostly-concrete interpretation soundly, and finally Section 4.4 gives our precision result for combined interpretation.

4.2 Mostly-Concrete Interpretation

Mostly-concrete interpretation introduces the following extensions to concrete interpretation:

Extension 1: The mostly-concrete interpreter supports runtime values that are finite abstractions of (potentially infinite) sets of values. For example, the abstract value $\{+\}$ from Section 2 is a finite abstraction of an infinite set of numbers. Thus, when converting from an abstract to mostly-concrete state, Concerto does *not* need to materialize an infinite (or unmanageably large) set of concrete values; the abstract interpretation need only provide these abstractions.

Extension 2: When an *fop* would yield an infinite or unmanageably large set of values, the mostly-concrete interpreter may instead use a special "unknown" value that represents "any possible value." The mostly-concrete semantics extend the concrete semantics to soundly handle this value.

Extension 3: The mostly-concrete semantic domain is a map from flow edges to a *single* mostly-concrete state, which itself maps program variables to the abstractions of multiple possible values mentioned in Extension 1 above. Thus, our mostly-concrete domain cannot track relationships between program variables, i.e., it is *non*-relational.

Except for these extensions, the mostly-concrete semantics mirror the concrete semantics (hence the name). Provided all *fop*s are deterministic and environment agnostic,[4] given a deterministic input state, the results of mostly-concrete and concrete interpretation on framework code will coincide.

To represent multiple possible values for variables of type $\mathcal{F}$, the mostly-concrete domain uses powersets of values in V_f extended with a special "unknown" value. For $\mathcal{A}$, the abstract interpretation provides an abstraction domain $\widehat{A}$ that satisfies the properties described in Section 4.3. Informally, the $\widehat{A}$ domain must be non-relational and path-insensitive. However, the abstract value domain used internally by the AI has no such restriction.

Formally, the domain of reaching mostly-concrete states $\widetilde{R}$ is:

$$\widetilde{R} = \mathcal{L} \to \widetilde{S} \qquad\qquad \widetilde{S} = (X_f \to \wp(V_f)^\top) \times (X_a \to \widehat{A})$$

As in Example 3.2, X_f and X_a are the sets of program variables with types in $\mathcal{F}$ and $\mathcal{A}$ respectively. $\wp(V_f)^\top$ is the powerset domain of concrete values in $\mathcal{F}$, extended with $\top_F$, which is the "unknown"

[4] We say an *fop* is environment agnostic if it does not depend on the environment model, i.e., $\forall e_1, e_2 \in \mathcal{E}, v_1, \ldots, v_n \in V_f.\{v \mid \langle v, \_\rangle \in [\![fop]\!](e_1, v_1, \ldots, v_n)\} = \{v \mid \langle v, \_\rangle \in [\![fop]\!](e_2, v_1, \ldots, v_n)\}$

value described in Extension 2. The abstractions for variables with types in $\mathcal{A}$ in the mostly-concrete interpreter are drawn from the domain $\widehat{A}$ provided by the AI; $\widehat{A}$ is assumed to form a complete lattice. $\widehat{A}$ may or may not be used internally by the abstract interpreter. The domain of $\widetilde{S}$ and $\widetilde{R}$ form a complete lattice equipped with the standard pointwise join and ordering operators, as well as top and bottom values. We further assume that the analysis defines a monotone complete join-morphism $\alpha_v : \wp(V_a) \to \widehat{A}$ that abstracts a set of concrete values of type $\mathcal{A}$ to a value in $\widehat{A}$.

Example 4.1 (Signedness Analysis)**.** For the signedness analysis, $\widehat{A}$ is the same domain as used in the abstract interpretation, $\wp(\{-, 0, +\})$. α_v is defined as: $\alpha_v(I) = \{sign(i) \mid i \in I\}$ where *sign* is defined as in Example 3.2. The join and ordering operators are set union and inclusion respectively. In the example from Section 2, Concerto used the abstract value $\{+\}$ for the value of `arg` in `dispatch` which abstracted the set $\{n | n > 0\}$. □

Example 4.2 (Pentagons)**.** Suppose instead the abstract interpretation of Section 2 had used a relational domain like Pentagons [Logozzo and Fähndrich 2008], which is the interval domain complemented with symbolic upper bounds. Then, $\widehat{A} = Intv$, where *Intv* is the plain interval domain [Cousot and Cousot 1977]. α_v is defined as $\alpha_v(I) = [lb\, I, ub\, I]$ (or $\perp$ if $I = \emptyset$), where *ub* and *lb* return the upper (resp. lower) bound of a set, or ∞ (resp. $-\infty$) if no such bound exists in $\mathbb{Z}$. The join and ordering operators are the standard interval union and inclusion operators. Unlike the above example, we *cannot* reuse the pentagon domain for $\widehat{A}$ for reasons discussed in Section 4.3. □

We now define an abstraction function α_F as follows:

$$\alpha_F(r)[\ell] = \left\langle \lambda x : X_f.\mathcal{V}(r[\ell], x),\ \ \lambda x : X_a.\alpha_v(\mathcal{V}(r[\ell], x))\right\rangle$$

Where $\mathcal{V}$ is the reaching value set for a set of states and variable, defined as: $\mathcal{V}(S, x) = \{s[x] \mid \langle s, \_\rangle \in S \wedge s[x] \neq \perp_V\}$. α_F approximates a variable x of type $\mathcal{F}$ with exactly the set of (initialized) values that reach x. Variables of type $\mathcal{A}$ are approximated by applying α_v to the set of reaching values. The approximation for a variable may not depend on the value of other variables, nor the location at which the variable is being approximated.

As α_v is a complete join morphism, α_F is also a complete join-morphism, and thus by [Cousot and Cousot 1979b], there exists some γ_F such that $R \underset{\alpha_F}{\overset{\gamma_F}{\leftrightarrows}} \widetilde{R}$.

In Fig. 7, we define the mostly-concrete semantic function $I_\top : \widetilde{R} \to \widetilde{R}$. The structure of $I_\top$ closely mirrors that of the concrete semantic function F. Operations and comparisons on values from types in $\mathcal{F}$ use the same operations as in F, but lifted to powersets of values. If one of the operands to a comparison is unknown (i.e., $\top_F$), then both branches are taken. The lifted version of $\widetilde{[\![fop]\!]}$ in Fig. 7 assumes that if any argument of an *fop* is unknown (again, $\top_F$) then the result is also unknown. Otherwise, the lifted version computes the result of applying the *fop* to every possible valuation of arguments in any environment. This can yield large (or even infinite sets) but the semantics of Fig. 7 do not abstract these sets to the $\top_F$ value, which simplifies our formal presentation. In practice, our implementation falls back on $\top_F$ for infinite to unmanageably large sets of values (Section 7).

Like concrete interpretation, mostly-concrete interpretation can exploit application-specific knowledge and precisely model *fop*s that would otherwise be treated as nondeterministic. As described in Section 3.1, we model effectively deterministic *fop*s by introducing hypotheses on the domain of program environments $\mathcal{E}$. Thus, despite taking the union over $\mathcal{E}$, the definition of $\widetilde{[\![fop]\!]}$ uses only environment models consistent with application-specific information.

Unlike *fop*s, operations on values of types from $\mathcal{A}$ are modeled with imprecise, albeit sound, semantics. Mostly-concrete interpretation can support arbitrary abstractions of $\mathcal{A}$ values precisely because it makes no attempt to interpret *aop*s and therefore does not need to “understand” the

$$I_\top(\widetilde{r})[\ell^\circ \rightsquigarrow \ell^\bullet] = \mathit{step}^\top\Big(\bigsqcup_{p \in \mathit{pred}(\ell)} \widetilde{r}[p^\bullet \rightsquigarrow \ell^\circ], \ell\Big) \sqcup \begin{cases} \mathit{step}^\top(\bot_{\widetilde{S}}, \ell) & \ell = s \\ \bot_{\widetilde{S}} & o.w. \end{cases} \qquad I_\top(\widetilde{r})[p^\bullet \rightsquigarrow \ell^\circ] = \begin{cases} \widetilde{r}[p] & \widetilde{\mathcal{FT}}(\widetilde{r}, p^\bullet \rightsquigarrow \ell^\circ) \\ \bot & o.w. \end{cases}$$

$$\widetilde{\mathcal{FT}}(\widetilde{r}, p^\bullet \rightsquigarrow \ell^\circ) = \begin{cases} \widetilde{s}[x]\widetilde{[\![\texttt{<=>}]\!]}\widetilde{s}[y] & p \in \ell_f \wedge \mathit{prog}[p] = \texttt{if } x\texttt{<=>}y \texttt{ goto } \ell & (2) \\ \widetilde{s}[x]\widetilde{[\![\texttt{<}\neq\texttt{>}]\!]}\widetilde{s}[y] & p \in \ell_f \wedge \mathit{prog}[p] = \texttt{if } x\texttt{<=>}y \texttt{ goto } \ell' & (3) \\ \mathit{true} & \text{o.w.} & (4) \end{cases}$$

Where $\widetilde{s} = \widetilde{r}[p^\circ \rightsquigarrow p^\bullet]$ and $\widetilde{[\![R]\!]}$ is the lifting of R defined by:

$$\widetilde{v}\widetilde{[\![R]\!]}\top_F \qquad \top_F\widetilde{[\![R]\!]}\widetilde{v} \qquad v \in \widetilde{v} \wedge v' \in \widetilde{v}' \wedge v[\![R]\!]v' \Rightarrow \widetilde{v}\widetilde{[\![R]\!]}\widetilde{v}'$$

$$\mathit{step}^\top(\widetilde{in}, \ell) = \begin{cases} \widetilde{in} & \mathit{prog}[\ell] = \texttt{if} \ldots \vee \texttt{goto}\ldots & (5) \\ \widetilde{in}[x \mapsto \widetilde{in}[y]] & \mathit{prog}[\ell] = x = y & (6) \\ \widetilde{in}[x \mapsto \{[\![bc_f]\!]\}] & \mathit{prog}[\ell] = x = bc_f & (7) \\ \widetilde{in}[x \mapsto \widetilde{[\![fop]\!]}(\widetilde{in}[v_1], \ldots, \widetilde{in}[v_n])] & \mathit{prog}[\ell] = x = \mathit{fop}(v_1, \ldots, v_n) & (8) \\ \widetilde{in}[x \mapsto \top_{\widehat{A}}] & \mathit{prog}[\ell] = x = \mathit{aexp} \wedge \mathit{type}(x) = A & (9) \end{cases}$$

$$\widetilde{[\![fop]\!]}(\widetilde{f_1}, \ldots, \widetilde{f_n}) = \begin{cases} \top_F & \widetilde{f_1} = \top_F \vee \ldots \vee \widetilde{f_n} = \top_F & (10) \\ \displaystyle\bigcup_{\substack{f_1 \in \widetilde{f_1}, \ldots, f_n \in \widetilde{f_n} \\ E \in \mathcal{E}}} \{f' \mid \langle f', \_\rangle \in [\![fop]\!](E, f_1, \ldots, f_n)\} & \text{o.w.} & (11) \end{cases}$$

Fig. 7. Mostly-concrete semantic function. In $I_\top$, s is again the distinguished program start label.

$$\overline{R} = (\mathcal{L}_F \to \widetilde{S}) \times (\mathcal{L}_A \to \widehat{S}) \qquad \alpha_C(r) = \left\langle \lambda\vec{\ell} : \mathcal{L}_F.\alpha_F(r)[\vec{\ell}],\ \lambda\vec{\ell} : \mathcal{L}_A.\alpha_A(r)[\vec{\ell}] \right\rangle$$

$$\widetilde{\tau} : \widehat{S} \to \widetilde{S} \text{ such that: } \forall\vec{\ell}, r : R.\alpha_F(r)[\vec{\ell}] \sqsubseteq_{\widetilde{R}} \widetilde{\tau}(\alpha_A(r)[\vec{\ell}]) \qquad \widetilde{inj} : \overline{R} \to \widetilde{R} \text{ where: } \widetilde{inj}\big(\langle\widetilde{m}, \widehat{m}\rangle\big)[\vec{\ell}] = \begin{cases} \widetilde{m}[\vec{\ell}] & \vec{\ell} \in \mathcal{L}_F \\ \widetilde{\tau}(\widehat{m}[\vec{\ell}]) & \vec{\ell} \in \mathcal{L}_A \end{cases}$$

$$\widehat{\tau} : \widetilde{S} \to \widehat{S} \text{ such that: } \forall\vec{\ell}, r : R.\alpha_A(r)[\vec{\ell}] \sqsubseteq_{\widehat{S}} \widehat{\tau}(\alpha_F(r)[\vec{\ell}]) \qquad \widehat{inj} : \overline{R} \to \widehat{R} \text{ where: } \widehat{inj}\big(\langle\widetilde{m}, \widehat{m}\rangle\big)[\vec{\ell}] = \begin{cases} \widehat{m}[\vec{\ell}] & \vec{\ell} \in \mathcal{L}_A \\ \widehat{\tau}(\widetilde{m}[\vec{\ell}]) & \vec{\ell} \in \mathcal{L}_F \end{cases}$$

Fig. 8. The combined mostly-concrete and abstract domain definitions, along with the domain transformers and derived injection functions.

abstraction domain $\widehat{A}$. However, Concerto does not suffer a precision loss from these coarse semantics; due to the state separation hypothesis, all *aops*, bc_a and comparisons over $\mathcal{A}$ occur in statements labeled ℓ_a, which are modeled in Concerto with abstract interpretation. In other words, the mostly-concrete semantics for operations over $\mathcal{A}$ values are imprecise, but never actually executed in the mostly-concrete interpreter.

Theorem 2. *$\alpha_F \circ F \sqsubseteq_{R \to \widetilde{R}} I_\top \circ \alpha_F$, i.e., $I_\top$ is a sound over-approximation of F.*

Proof Sketch. By case analysis on the definitions of step^F and $\mathit{step}^\top$. The full proofs are available in Appendix A.1. □

4.3 Combined Abstract and Mostly-Concrete Interpretation

We now show how to combine the mostly-concrete and abstract interpreters. The approach broadly mirrors the strawman approach from Section 4.1. Specifically, combined interpretation operates over a combined domain $\overline{R}$. Like its strawman counterpart $\overline{R}_0$, $\overline{R}$ represents reaching states in the application with abstract state $\widehat{S}$, but uses mostly-concrete states $\widetilde{S}$ for the framework instead of $\wp(S)$. Concerto's combined interpretation also injects the combined state representation into the "native" formats expected by the abstract and mostly-concrete interpreters. However, instead of using abstraction and concretization functions as in Section 4.1, we use *domain transformers* to soundly translate between state representations without requiring one of the abstract or mostly-concrete state representation to be more precise than the other.

The combined analysis domain $\overline{R}$ is defined in Fig. 8, along with an abstraction function α_C. As α_A and α_F are complete join morphisms, α_C is itself a complete join morphism, and thus there exists some γ_C such that R and $\overline{R}$ form a Galois connection. The monotone functions $\widetilde{\tau}$ and $\widehat{\tau}$ in Fig. 8 are the domain transformers described above: $\widetilde{\tau}$ transforms a state from the abstract interpreter into a mostly-concrete state, and $\widehat{\tau}$ performs a transformation in the opposite direction. They are both functions provided by the analysis that must fulfill the following conditions:

$$\forall\vec{\ell}, r : R.\alpha_A(r)[\vec{\ell}] \sqsubseteq_{\widehat{S}} \widehat{\tau}(\alpha_F(r)[\vec{\ell}]) \qquad (12) \qquad\qquad \forall\vec{\ell}, r : R.\alpha_F(r)[\vec{\ell}] \sqsubseteq_{\widetilde{S}} \widetilde{\tau}(\alpha_A(r)[\vec{\ell}]) \qquad (13)$$

Intuitively, conditions (12) and (13) state that the transformers must be consistent with the target domain's abstraction function. As a consequence, (13) implies that any relational information present in $\widehat{R}$ must be discarded when moving to the non-relational domain $\widetilde{R}$. The $\widehat{A}$ values produced by α_F are the result of a non-relational abstraction function α_v, and by the inequality of Eq. (13), the result of $\widetilde{\tau}$ can do no better. Despite this restriction on relational abstractions in the mostly-concrete domain, the above requirements on the the domain transformers do not provide information about the relative precision of the two domains. In fact, as mentioned above and illustrated below, it may not necessarily be the case that one of the domains is more precise than the other.

Example 4.3 (Trivial Transformers)**.** Consider the domain of Example 3.2 and the definition of α_v from Example 4.1. The two state representations are equal ($\widehat{S} = \widetilde{S}$), and $\widetilde{\tau} = \widehat{\tau} = id$. In other words, the two domains have the same expressive power. □

Example 4.4 (Relational Domain)**.** Suppose instead of representing integers with of the signedness domain used in Example 3.2, we used the Pentagon domain of Example 4.2 with the corresponding α_v and $\widehat{A} = Intv$. Then the abstract domain is $X_a \rightarrow Intv \times X_a \rightarrow \wp(X_a) \times X_f \rightarrow \wp(V_f)^\top$, where the first two components are respectively the interval environment and strict upper bounds of integers described in [Logozzo and Fähndrich 2008]. This abstract domain has used the $\wp(V_f)^\top$ representation for framework variables. As with the embedding of $\widehat{A}$ in the mostly-concrete interpreter, this embedding of mostly-concrete values into the abstract domain is feasible due to the state separation hypothesis: i.e., *fop*s will not appear in code analyzed by the abstract interpreter.

The domain transformers may be defined as:

$$\widetilde{\tau}(\langle b, s, m\rangle) = \langle b, m\rangle \qquad\qquad \widehat{\tau}(\langle b, m\rangle) = \langle b, \lambda x : X_a.\emptyset, m\rangle$$

That is, $\widetilde{\tau}$ discards the relational information from $\widehat{S}$ when moving to $\widetilde{S}$, and $\widehat{\tau}$ uses the top element of the strict upper bound domain in the output (as the input mostly-concrete state does not have any relational information). In this example, the abstract domain is more precise, i.e. $\widehat{S} \underset{\widetilde{\tau}}{\overset{\widehat{\tau}}{\leftrightarrows}} \widetilde{S}$. □

Example 4.5 (Trivial Abstract Domain)**.** Consider a domain $X_a \rightarrow \wp(\{-, 0, +\}) \times X_f \rightarrow \mathbf{1}$, where $\mathbf{1}$ is unary domain whose single element $\mathtt{tt}$ represents "any possible value", i.e., the analysis does not try to reason about maps, strings, or I/O. Then $\widetilde{\tau}(\langle m, z\rangle) = \langle m, \lambda x : X_f.\top_F\rangle$ and $\widehat{\tau}(\langle m, t\rangle) = \langle m, \lambda x : X_f.\mathtt{tt}\rangle$, and $\widetilde{S} \underset{\widehat{\tau}}{\overset{\widetilde{\tau}}{\leftrightarrows}} \widehat{S}$, i.e., the mostly-concrete domain is more precise than the abstract domain. □

Example 4.6 (Mixed Expressiveness)**.** Finally, consider a combination of Examples 4.4 and 4.5 where integers (i.e., variables in X_a) are modeled in the abstract domain with pentagons, $\widehat{A} = Intv$, but framework types (maps and strings) are modeled with the highly imprecise domain $\mathbf{1}$. Then $\widetilde{\tau}(\langle b, s, z\rangle) = \langle b, \lambda x : X_f.\top_F\rangle$ and $\widehat{\tau}(\langle b, m\rangle) = \langle b, \lambda x : X_a.\emptyset, \lambda x : X_f.\mathtt{tt}\rangle$. $\widetilde{\tau}$ and $\widehat{\tau}$ do not form a

$$LB(r)[\ell^\circ \rightsquigarrow \ell^\bullet] = step^{LB}(\sqcup_{p\in pred(\ell)} r[p^\bullet \rightsquigarrow \ell^\circ], \ell) \sqcup \begin{cases} step^{LB}(\bot_{\widetilde{S}}, s) & \ell = s \\ \bot_{\widetilde{S}} & o.w. \end{cases}$$

$$LB(r)[\ell_f^\bullet \rightsquigarrow \ell^\circ] = r[\ell_f^\circ \rightsquigarrow \ell_f^\bullet] \qquad\qquad LB(r)[\ell_a^\bullet \rightsquigarrow \ell^\circ] = \bot$$

$$step^{LB}(\widetilde{in}, \ell) = \begin{cases} \widetilde{in} & prog[\ell] = \texttt{if} \ldots \vee prog[\ell] = \texttt{goto}\ \ell & (15) \\ \widetilde{in}[x \mapsto \widetilde{in}[y]] & prog[\ell] = x = y & (16) \\ \widetilde{in}[x \mapsto \bot_{\widehat{A}}] & prog[\ell] = x = bc_a \vee prog[\ell] = x = aop(v_1, \ldots, v_n) & (17) \\ \widetilde{in}[x \mapsto \top_F] & prog[\ell] = x = fop(v_1, \ldots, v_n) \vee prog[\ell] = x = bc_f & (18) \end{cases}$$

Fig. 9. Semantic function defining a lower-bound on the precision of the abstract interpretation.

Galois connection: the abstract domain is more precise for integers, but the mostly-concrete domain more precisely represents maps and strings. □

We are ready to define the combined interpretation function $C : \overline{R} \to \overline{R}$ as follows:

$$C(X) = \left\langle I_\top \circ \widetilde{inj}(X)|_{\mathcal{L}_F},\ \widehat{F} \circ \widehat{inj}(X)|_{\mathcal{L}_A} \right\rangle \tag{14}$$

This definition closely mirrors Eq. (1). The injection functions $\widetilde{inj}$ and $\widehat{inj}$ (defined in Fig. 8) take the place of inj_R and $inj_{\widehat{R}}$ and translate the combined domain into $\widetilde{R}$ and $\widehat{R}$ by using the domain transformers $\widetilde{\tau}$ and $\widehat{\tau}$ respectively. The fully-concrete semantic function F has also been replaced with the mostly-concrete semantic function $I_\top$.

We can state the soundness of C according to α_C:

Theorem 3. $\alpha_C \circ F \sqsubseteq_{R \to \overline{R}} C \circ \alpha_C$

Proof Sketch. By Theorem 2, the assumed soundness of $\widehat{F}$, and from the fact that $\alpha_F \sqsubseteq \widetilde{inj} \circ \alpha_C$ and that $\alpha_A \sqsubseteq \widehat{inj} \circ \alpha_C$. □

As C is a monotone function on a complete lattice it has a least fixed point [Tarski 1955]. From Theorem 3 and [Cousot and Cousot 1979b] we then have that: $\alpha_C(lfp\, F) \sqsubseteq lfp\, C$.

4.4 Conditions for Increased Precision

C is sound, but may not necessarily be more precise than the original function $\widehat{F}$. We now discuss a set of sufficient conditions for when C is at least as (if not more) precise as $\widehat{F}$.

First, we define a function $LB : \widetilde{R} \to \widetilde{R}$ as shown in Fig. 9. Intuitively, LB provides a lower-bound on the precision of the abstract semantic function; showing that $I_\top$ can "do better" than this lower bound will imply that $I_\top$ provides improved precision compared to $\widehat{F}$ on framework code. LB specifies an imprecise lower bound for modeling framework operations and comparisons, but provides no lower bound on the precision for application operations or comparisons. However, as LB is non-relational, the definition implies that $\widehat{F}$ must also be non-relational.

C is more precise (as defined below) if the following conditions hold:

$$\widehat{\tau} \circ \widetilde{\tau} = id \qquad LB \circ \dot{\widetilde{\tau}} \sqsubseteq_{\widehat{R} \to \widetilde{R}} \dot{\widetilde{\tau}} \circ \widehat{F} \qquad \forall U \subseteq \widehat{S}.\widetilde{\tau}(\sqcup U) = \sqcup_{\widehat{s} \in U} \widetilde{\tau}(\widehat{s}) \tag{19}$$

That is, if no precision is lost by moving to the domain $\widetilde{S}$ and then back to $\widehat{S}$, if LB is a lower bound on the precision of $\widehat{F}$, and if $\widetilde{\tau}$ is a complete join morphism. In the above, $\dot{\widetilde{\tau}}$ denotes the pointwise extension of $\widetilde{\tau}$.

As C and $\widehat{F}$ operate over different domains, we first introduce a function $\widehat{proj} : \widehat{R} \to \overline{R}$ to *project* the $\widehat{R}$ domain into the combined domain $\overline{R}$:

$$\widehat{proj}(\widehat{s}) = \langle \lambda\vec{\ell} : \mathcal{L}_F.\widetilde{\tau}(\widehat{s}[\vec{\ell}]),\ \lambda\vec{\ell} : \mathcal{L}_A.\widehat{s}[\vec{\ell}] \rangle$$

$\widehat{proj}$ uses $\widetilde{\tau}$ to translate abstract to mostly-concrete states, reversing the $\widehat{inj}$ operation.

If the above conditions hold, we can prove:

Theorem 4. $\widehat{inj}(\mathit{lfp}\,C) \sqsubseteq_{\widehat{R}} \mathit{lfp}\,\widehat{F}$

Proof Sketch. From the assumptions in Eq. (19) and by case analysis on $step^{\top}$ and $step^{LB}$, it can be shown that $C \circ \widehat{proj} \sqsubseteq_{\widehat{R}\to\overline{R}} \widehat{proj} \circ \widehat{F}$ (Lemma 6 in the appendix), whence it follows by straightforward transfinite induction (Lemma 7 in the appendix) that $\mathit{lfp}\,C \sqsubseteq_{\overline{R}} \widehat{proj}(\mathit{lfp}\,\widehat{F})$. As $\widehat{inj}$ is monotone, we have: $\widehat{inj}(\mathit{lfp}\,C) \sqsubseteq_{\widehat{R}} \widehat{inj} \circ \widehat{proj}(\mathit{lfp}\,\widehat{F})$, whence we have $\widehat{inj}(\mathit{lfp}\,C) \sqsubseteq_{\widehat{R}} \mathit{lfp}\,\widehat{F}$, as $\widehat{inj} \circ \widehat{proj} = id$ by Eq. (19). □

Theorem 4 states that the approximation of reaching states computed by the combined interpretation function C is at least as precise as that computed by the abstract interpreter. The inequality is *not* strict: whether Concerto matches or exceeds the precision of plain abstract interpretation depends on the program and the abstract semantics and domain. As a trivial example, on a program with only application statements (i.e., with labels drawn from $\mathcal{L}_A$) Concerto will necessarily do no better than plain abstract interpretation.

Example 4.7 (Signedness Analysis)**.** The signedness example of Section 2 satisfies the above conditions, and thus combined interpretation with Concerto is at least as precise as plain abstract interpretation. As both $\widetilde{\tau}$ and $\widehat{\tau}$ are the identity function, parts 1 and 3 of Eq. (19) are trivially satisfied. The abstract semantic function in Fig. 6 uses very coarse approximations of *fop* operations, and therefore part 2 of Eq. (19) is also satisfied. □

5 PROCEDURES

Our formalism so far does not support procedures, but they are essential. Section 5.1 sketches the addition of procedures to our formal language and extensions to the concrete and mostly-concrete semantics. Section 5.2 then demonstrates that by restricting control transfers between the framework and application to procedures calls and returns, Concerto does not need AI developers to provide the full domain transformers ($\widetilde{\tau}$ and $\widehat{\tau}$) introduced in Section 4.3 since it suffices to transform only parameter and return values across procedure boundaries. We delay until Section 8 how to extend our support for procedures to objects and methods in the style of Java.

5.1 Interprocedural Semantics

We assume a program now has the following form, where f ranges over procedure names:

$$\begin{array}{rcl@{\qquad}rcl} prog & ::= & D* & astmt & ::= & \ldots \mid \texttt{return}\ x \mid x = f(y) \\ D & ::= & \texttt{proc}\ f(p)\{\ (fstmt^{\ell} \mid astmt^{\ell})^{*}\} & fstmt & ::= & \ldots \mid \texttt{return}\ x \mid x = f(y) \end{array}$$

We assume that procedure names are distinct, that there is a distinguished main procedure, and that the distinguished program start label s is the first statement of this procedure.

For a procedure call statement labeled with ℓ, we assume there are two pseudo-labels: ℓ_c and ℓ_r, corresponding to the point in program execution immediately before invoking the function and immediately after the called function returns. We extend the *succ* and *pred* relations to include these pseudo-labels. For a call-site labeled ℓ, if control may flow from label ℓ' to ℓ, $(\ell', \ell_c) \in succ$, and similarly for (ℓ_r, ℓ') and control-flow from ℓ to ℓ'. In addition, the entry point label of the callee at ℓ is a successor of ℓ_c and the predecessor of ℓ_r is the return site label in the callee.

We extend the state definition to track the runtime stack: $R = \mathcal{L} \to \wp(S^{*} \times \ell^{*} \times \mathcal{E})$, where S^{*} is a sequence of states S as defined in Section 3.2, and ℓ^{*} is a sequence of statement labels. Intuitively, these two components represent the runtime stack and the return locations for active procedure invocations respectively. We also assume that the domain of variables X includes a special return slot ρ.

$$F(r)[\ell_r^\circ \rightsquigarrow \ell_r^\bullet] = \bigsqcup_{p \in pred(\ell_r)} \{\langle s \circ s_r[x \mapsto s_c[\rho]], \mathcal{R}, E\rangle \mid \langle s \circ s_r \circ s_c, \mathcal{R}, E\rangle \in r[p^\bullet \rightsquigarrow \ell_r^\circ] \wedge \langle s \circ s_r, \mathcal{R}, \_\rangle \in r[\ell_c^\circ \rightsquigarrow \ell_c^\bullet]\}$$

$$F(r)[p^\bullet \rightsquigarrow \ell_r^\circ] = \{\langle s, \mathcal{R}, E\rangle \mid \langle s, \mathcal{R} \circ \ell, E\rangle \in r[p]\}$$

$$F(r)[\ell_c^\bullet \rightsquigarrow \ell'^\circ] = \{\langle s \circ s_r \circ \iota_S[p \mapsto s_r[y]], \mathcal{R} \circ \ell, E\rangle \mid \langle s \circ s_r, \mathcal{R}, E\rangle \in r[\ell_c]\}$$

$$F(r)[\ell_c^\circ \rightsquigarrow \ell_c^\bullet] = \bigsqcup_{p \in pred(\ell_c)} r[p^\bullet \rightsquigarrow \ell_c^\circ]$$

$$I_\top(\widetilde{r})[\ell_r^\circ \rightsquigarrow \ell_r^\bullet] = \bigsqcup_{p \in pred(\ell_r)} \widetilde{r}[\ell_c][x \mapsto \widetilde{r}[p^\bullet \rightsquigarrow \ell_r^\circ][\rho]]$$

$$I_\top(\widetilde{r})[p^\bullet \rightsquigarrow \ell_r^\circ] = \widetilde{r}[p]$$

$$I_\top(\widetilde{r})[\ell_c^\bullet \rightsquigarrow \ell'^\circ] = [p \mapsto \widetilde{r}[\ell_c][y]]$$

$$I_\top(\widetilde{r})[\ell_c^\circ \rightsquigarrow \ell_c^\bullet] = \bigsqcup_{p \in pred(\ell_c)} \widetilde{r}[p^\bullet \rightsquigarrow \ell_c^\circ]$$

$$step^\top(\widetilde{in}, \ell) = \begin{cases} \widetilde{in}[\rho \mapsto \widetilde{in}[x]] & prog[\ell] = \texttt{return}\ x \\ \textit{as before} & o.w. \end{cases}$$

$$step^F(in, \ell) = \begin{cases} in[\rho \mapsto in[x]] & prog[\ell] = \texttt{return}\ x \\ \textit{as before} & o.w. \end{cases}$$

Fig. 10. Extensions to the semantic functions F and $I_\top$ to support procedures. In the above definitions, p is the name of the parameter of the called procedures, y is the variable passed as the argument, and x is the variable in the caller to which the result of the function is defined. ι_S is a state where all variables are bound to $\bot_V$.

We extend the definitions of F and $I_\top$ given in Sections 3.2 and 4.2 respectively as shown in Fig. 10 to handle these new statement forms and flow edges. We omit the updated definitions of F and $I_\top$ at the distinguished start label, although they are the obvious extensions to the terms in Figs. 5 and 7. We use ρ to store the return value of functions with return type $\mathcal{F}$ and $\mathcal{A}$: we shall assume that there are two different versions of ρ of the appropriate type. Finally, $\mathcal{V}$ is now defined as: $\mathcal{V}(\mathcal{S}, x) = \{s[x] \mid \langle s_0 \circ s, \_, \_\rangle \in \mathcal{S} \wedge s[x] \neq \bot_V\}$.

We have proved (see Appendix B) that this updated definition of $I_\top$ is sound with respect to the updated definition of F. If the abstract interpretation is also sound with respect to this updated definition, and the $\widetilde{\tau}$ and $\widehat{\tau}$ functions still fulfill the conditions in Section 4 then all the results of the previous section still hold with no modifications to our formalisms or proofs.

5.2 Interprocedural Domain Transformers

We have so far treated procedure call and return as orthogonal to control-flow transfer between the application and framework, i.e., procedure bodies could be a mix ℓ_f and ℓ_a statements. In practice, we require all transfers between the application and framework occur at procedure boundaries. In other words, each procedure may only contain ℓ_a statements or ℓ_f statements. This restriction is entirely reasonable: real frameworks encapsulate their functionality in methods/classes/modules/etc.: source programs would not mix framework and application code in the same procedure.

This (non-)restriction also significantly simplifies how Concerto transfers values between abstract and mostly-concrete interpretation. Recall that Concerto applies the domain transformers at transitions between the application and framework. However, the above syntactic restriction implies that Concerto needs the domain transformers only at procedure entry and exit. Further, at procedure entry, the mostly-concrete interpreter can access only the program state reachable via the parameters. Thus, instead of using $\widetilde{\tau}$ at application-to-framework calls, the AI may directly provide mostly-concrete arguments to the mostly-concrete component, which then binds the arguments in a empty local state. Similarly, we assume that when given mostly-concrete arguments the abstract interpreter can construct a sound abstract procedure entry state. (This assumption is possible because our language is statically scoped and has no global variables, heaps are discussed in Section 8.) As a consequence, instead of using $\widehat{\tau}$ at framework-to-application calls, the mostly-concrete interpreter may provide mostly-concrete arguments to the AI, which binds them in an abstract empty state, transforming the mostly-concrete values into a "native" abstract representation as necessary.

Concerto uses a similar process for return flows. The mostly-concrete semantics use the exit state of a called procedure only to extract the procedure's return value. Instead of using $\widetilde{\tau}$ at return flows from the application to framework, the AI simply provides a mostly-concrete representation of the return slot ρ. We likewise assume that the AI is interested only in the value of ρ in callee exit states. Thus, at framework-to-application return flows, the mostly-concrete interpreter may provide only the mostly-concrete return value, which the AI transforms into a native representation as necessary.

The direct exchange of values sketched above obviates the need for implementations of $\widetilde{\tau}$ and $\widehat{\tau}$, but to retain soundness, the values exchanged must be consistent with those produced by some $\widetilde{\tau}$ and $\widehat{\tau}$ that satisfy the definitions given in Section 4.3. For example, consider an application-to-framework call where the abstract entry state is $\widehat{s}$. The mostly-concrete argument provided to the mostly-concrete interpreter must therefore be $\widetilde{\tau}(\widehat{s})[p]$ for some $\widetilde{\tau}$, and where p is the parameter of the called procedure. Thus, any valid domain transformers $\widetilde{\tau}$ and $\widehat{\tau}$ provide correctness specifications for the exchange of values. However, a trivial way to ensure soundness is to generate complete definitions for some $\widetilde{\tau}$ and $\widehat{\tau}$, and hand-simplify the results of applying the transformers at procedure call and return. For example, to correctly generate mostly-concrete return values at application-to-framework return flows, it is sufficient to use a simplification of $\lambda\widehat{s}.\widetilde{\tau}(\widehat{s})[\rho]$.

Context Sensitivity. The above discussion does not treat context-sensitivity in the abstract interpretation. In our implementation we have the AI provide functions to compute the callee context at framework-to-application calls. Further the mostly-concrete semantics in Fig. 10 are context-*insensitive*, which is not very precise in practice. In practice, we unroll the call-graph up to recursive cycles in the mostly-concrete interpreter, effectively giving unlimited context-sensitivity. We discuss these two techniques in Section 8.3. We have not formalized our approach to context-sensitivity, although adapting our proofs and formalism is a straightforward, albeit tedious, extension.

6 ITERATION STRATEGY

We now briefly describe the iteration strategy used by Concerto. Our implementation runs mostly-concrete and abstract interpreters in parallel until they converge to a local fixed-point on their respective partitions of the program (framework and application code respectively). The results are then exchanged between the two interpreters (under the syntactic restriction of Section 5.2, this exchange is performed at procedure boundaries as described above) and the process repeats until the overall process converges to a fixed-point. We refer to this process as *subfixpoint iteration*.

The subfixpoint iteration scheme sketched above is very similar to chaotic asynchronous iteration with memory [Cousot 1977], with one key difference. Before starting iteration in the mostly-concrete component, subfixpoint iteration discards results at framework statements computed in previous rounds of subfixpoint iteration. In other words, after exchanging information with the abstract interpretation component, the mostly-concrete component iterates a "fresh" mostly-concrete interpreter, beginning with only information received from the abstract interpreter. We have shown in Appendix C that this process converges to the least fixed point of C. Our proof depends on the following property of subfixpoint iteration (Lemma 21 in the appendix): any information discarded between runs of the mostly-concrete interpreter can be soundly recovered with enough iterations in the next run of the mostly-concrete interpreter.

This iteration strategy justifies analyzing application-to-framework calls by spawning a fresh mostly-concrete interpreter seeded with mostly-concrete arguments provided by the AI that flow into the framework from the application. We describe this process in more detail in Section 8.

7 WIDENING AND FINITIZATION

Two significant challenges remain to a realizable implementation. First, although we have proved that subfixpoint iteration converges to the least fixed point of C, it may not do so in finite time; the domain $\overline{R}$ does not possess an ascending chain condition that will ensure convergence in finite steps. Second, we have not yet guaranteed that mostly-concrete interpretation manipulates only finite sets of values. To address the first issue, we apply widening [Cousot and Cousot 1977] during iteration. We address the second issue by forbidding infinite sets of values, and describe how the mostly-concrete interpreter uses $\top_F$ in practice to avoid materializing infinite sets.

Widening. Following the vocabulary of [Bourdoncle 1993], we require two *widening point sets* $\mathcal{W}_A$ and $\mathcal{W}_F$ for application and framework statements, respectively. A widening point set is a set of statement labels such that, if during iteration the states at all widening points stabilize, then the overall iteration stabilizes in a finite number of steps. We further require that if the states at all widening points in two iteration sequences stabilize to the same set of values, then the two sequences stabilize to the same result. We leave the choice of $\mathcal{W}_A$ up to the abstract interpretation, although we expect most interpreters will use a variation on the strategy described by [Bourdoncle 1993]. In our mostly-concrete interpreter, we use the headers of unbounded loops and the entry point of a representative method selected from recursive cycles (including sub-cycles).

We assume that the analysis provides widening operators $\nabla_{\widehat{A}}$ for values of type $\widehat{A}$ and $\widehat{\nabla}$ for abstract states $\widehat{S}$. From $\nabla_{\widehat{A}}$, we derive a widening operator for mostly-concrete states:

$$\langle m_f, m_a\rangle \widetilde{\nabla} \langle m'_f, m'_a\rangle = \left\langle \left(\lambda x : X_f . \begin{cases} m_f & m'_f[x] \sqsubseteq m_f[x] \\ \top_F & o.w. \end{cases}\right), \ \left(\lambda x : X_a . m_a[x] \nabla_{\widehat{A}} m'_a[x]\right)\right\rangle$$

Given the above assumptions and definitions, we ensure termination as follows. We again iterate the abstract and mostly-concrete interpreters in parallel, except we instrument the abstract semantic function $\widehat{F}$ and $I_\top$ to apply widening operations at the locations in the widening point sets $\mathcal{W}_A$ and $\mathcal{W}_F$ respectively. We have proved (Appendix D) that these individual iterations terminate in a finite number of steps. After the two interpreters stabilize, they exchange results and the process repeats; with the mostly-concrete interpreter again discarding previously computed information as described above. This process stabilizes in a finite number of steps to an over-approximation of $\mathit{lfp}\, C$ (see Appendix D.1).

Precision. Section 4.4 gave conditions for when $\mathit{lfp}\, C$ will be at least as precise as $\mathit{lfp}\, \widehat{F}$. Whether this precision result also translates to the widened subfixpoint iteration presented above will depend on the choice of widening operators. As widening operators are not necessarily monotone, the instrumented $\widehat{F}$ and $I_\top$ functions are not necessarily monotone either. Without monotonicity, reasoning about the relative precision of subfixpoint iteration with widening is difficult. This result is not surprising; as noted in [Cousot and Cousot 1992a], when using widening the order of iteration can have a significant impact on the precision of the final result.

Finitization. Finally, to ensure Concerto manipulates only finite sets, we require that the AI does not provide infinite arguments or return value representations to the mostly-concrete interpreter. We also extend the definition of $I_\top$ to finitize the result of all *fop* operations. Whenever applying *fop* to two arguments would produce an infinite set of values (or an otherwise impractically large set, e.g., all 32-bit machine integers) the modified semantic function abstracts this set with $\top_F$. This finitization is sound, while avoiding materializing infinite sets in our implementation.

8 EXTENSIONS FOR A REALISTIC PROTOTYPE

We have implemented a prototype combined interpreter for a subset of Java.[5] Our subset includes 1) interfaces and dynamic dispatch, 2) reflection, 3) dynamically sized arrays, and 4) (potentially) unbounded loops. We include two primitives to simulate I/O. The first, `read()`, reads an integer from a deterministic stream. This primitive models reading from a deterministic configuration file. The second primitive, `nondet()` reads from a nondeterministic stream, which simulates, e.g., packets received from the network or user input. While falling considerably short of the full complexity of

[5]Our implementation prototype is open-source and is available at https://github.com/uwplse/concerto.

Java, we can use these language features to effectively simulate some of the most difficult to analyze code idioms we have encountered in real-world framework implementations (see Section 9).

Our prototype takes as input an abstract interpretation implementation which exposes basic operations required by Concerto. We introduce these operations incrementally as we extend our basic procedural language to support objects, methods, etc.

8.1 Objects and the Heap

We first consider only class-based objects with fields, deferring methods to Section 8.2 and primitives, interfaces, and libraries to Section 8.4. We assume each concrete class belongs either to the framework or the application, taking $\mathcal{F}$ to be the framework classes and $\mathcal{A}$ the application classes. We also extend our concrete state to include a concrete heap: $S = \mathcal{E} \times (X \rightarrow V) \times H$. Object allocation and field manipulation are defined via *aop*s and *fop*s that additionally side-effect the heap (we omit a full formalization for space reasons). This formulation implies that framework code may not directly manipulate the object fields of application classes and vice versa. However, as argued in Section 2, real-world applications and frameworks almost exclusively communicate via functional interfaces.

By classifying each class as either framework or application, we can effectively partition the program's runtime heap H into H_a containing application objects and H_f containing framework objects, i.e., $S = \mathcal{E} \times (X \rightarrow V) \times H_a \times H_f$. We can then use different abstractions for H_a and H_f.

In the mostly-concrete interpreter, object operations on framework classes manipulate mostly-concrete heaps of type $\widetilde{H_f}$.[6] $\widetilde{H_f}$ is equipped with join, ordering, and widening operators. The mostly-concrete interpreter does not use its own abstraction for H_a, using instead the one provided by the abstract interpreter (see below). Similarly, in the abstract interpreter, object operations on application classes operate on an abstract representation of H_a, i.e., the application heap component.[7] Concerto does *not* make any assumptions on the *internal* heap representation used by the abstract interpretation. However, the abstract interpretation must provide two functions: $\texttt{projectH} : \widehat{S} \rightarrow \widehat{H_a}$ and $\texttt{injectH} : \widehat{H_a} \rightarrow \widehat{S} \rightarrow \widehat{S}$ such that $\forall \widehat{s}. \widehat{s} \sqsubseteq_{\widehat{S}} \texttt{injectH}\ (\texttt{projectH}\ \widehat{s})\ \widehat{s}$ for some type $\widehat{H_a}$ defined by the analysis. The AI must provide widening, join, and ordering operations for $\widehat{H_a}$.

To tie these two heap representations together, we extend the mostly-concrete state representation to include mostly-concrete and abstract heaps: $\widetilde{S} = (X_f \rightarrow \wp(V_f)^\top) \times (X_a \rightarrow \widehat{A}) \times \widetilde{H_f} \times \widehat{H_a}$. V_f is the domain of mostly-concrete heap locations, and $\widehat{A}$ is an abstract representation of objects of type $\mathcal{A}$. During execution, the mostly-concrete interpreter updates the mostly-concrete heap and threads the abstract heap representation through unchanged. Concerto requires that the abstract interpreter also threads the mostly-concrete heap through its interpretation. To ensure the concrete heap is correctly handled, the abstract interpretation must operate over an instrumented state representation, $\widehat{S} \times H_f$. We provide APIs for the AI to manipulate this instrumented representation.

The approach described so far does not allow for framework objects to store references to application objects and vice versa. To relax this restriction, we require the abstract interpretation meets some additional conditions. First, the abstract heap must represent fields and variables with type $\mathcal{F}$ with the domain $\wp(V_f)^\top$. Next, when performing a write of $v' : \wp(V_f)^\top$ to a field/variable of type $\mathcal{F}$ with the existing value $v : \wp(V_f)^\top$, the value v'' of the field/variable after the write must satisfy the constraint $v'' \sqsubseteq v' \sqcup v$, i.e., the new value is bounded above by the result from a weak update. This requirement ensures that the abstract interpretation never produces mostly-concrete

[6] We do not use a fully concrete heap to handle object allocations in unbounded loops. Our widening operator detects such cases and introduces mostly-concrete *summary objects* where appropriate.

[7] Technically, the abstract interpretation may also operate on an abstraction of H_f. However, all object operations that mutate H_f are modeled by the mostly-concrete interpreter, so in practice the abstract interpreter only uses an abstraction of H_a.

object locations "out of thin air" that may not have yet been allocated in the mostly-concrete heap component. A similar concern exists when storing values of type $\mathcal{A}$ into the concrete heap: any abstractions stored into the concrete heap must remain valid, and updates via aliasing should be propagated to these values. A sufficient condition is for the abstract interpreter to internally use an abstract heap $\widehat{Loc} \rightharpoonup \widehat{O}$ where $\widehat{Loc}$ is a finite domain of abstract locations and $\widehat{O}$ are abstract objects, and to take $\widehat{A} = \wp(\widehat{Loc})$ and $\widehat{H_a} = \widehat{Loc} \rightharpoonup \widehat{O}$.

8.2 Methods and Domain Transformers

We require that methods in application classes contain only application code, and similarly for framework classes. As described in Section 5.2, this (non-)restriction ensures that values change representation only at method boundaries.

For framework-to-application calls, the abstract interpreter must expose a method `interpret` that analyzes a method m in context C (see below), with mostly-concrete arguments $a_1, a_2, \ldots, a_n$, and abstract receiver $\widehat{r}$. When Concerto encounters a method call in the mostly-concrete interpreter with a base pointer $\widehat{r}$ of type $\widehat{A}$ (i.e., static type $\mathcal{A}$), it yields into the abstract interpreter by passing the mostly-concrete arguments, abstract receiver, and a computed context C to `interpret`. `interpret` is responsible for constructing an initial abstract state for m and then performing abstract interpretation over the method body. When analysis of m is complete, `interpret` returns a mostly-concrete representation of the return value. This process mirrors the one described in Section 5.2. However, Concerto additionally instruments the above process to inject the caller's mostly-concrete and abstract heaps into the callee abstract state, and similarly extract the abstract and mostly-concrete heaps from the abstract exit state.

Concerto also provides an API for the AI to yield into the mostly-concrete interpreter when it encounters a call back into the framework. The AI calls this API method with a mostly-concrete receiver and arguments as well as the abstract caller state. The mostly-concrete interpreter extracts the two heaps from this caller state, binds the argument and receiver values, and begins executing the called method. When execution of the method completes, Concerto injects the resulting mostly-concrete and abstract heaps into the provided abstract caller state, and returns this updated state and a mostly-concrete return value to the AI.

8.3 Context-Sensitivity and Mostly-Concrete Interpretation

Concerto supports context-sensitive analyses. Our implementation is polymorphic over the type of contexts, leaving the representation entirely to the client abstract interpretation. When computing the context for an application method call, Concerto passes information about the current state, call site, and call stack to a `mkContext` method exposed by the AI. `mkContext` is responsible for computing the analysis context C for the method call and returning it to Concerto, which passes the computed context to the `interpret` function as described above.

At application-to-framework calls, Concerto spawns a fresh mostly-concrete interpreter and runs it until the called method returns. Where possible, this interpreter unrolls all statically bounded loops and unfolds the call-graph (effectively giving unlimited context-sensitivity and some path-sensitivity). However, due to nondeterministic program inputs or imprecision in the abstract interpreter, mostly-concrete interpretation may encounter nondeterministic conditionals, loops, and unbounded recursive cycles.

To handle nondeterministic choice, Concerto could fork two interpreters and continue execution down each path in parallel as in Klee [Cadar et al. 2008] or Java PathFinder [Brat et al. 2000]. While sound, this approach would encounter the exponential explosion of paths common in symbolic execution. Instead, Concerto forks two interpreters at nondeterministic branches and executes both branches in parallel up to the conditional control-flow join point. At the control-flow join point, the two interpreter's states are joined and execution continues along a single thread of execution.

To ensure termination in the presence of nondeterministic loops or unbounded recursion, the mostly-concrete interpreter is instrumented to detect potentially infinite loop unrolling or call-graph unfolding and then falls back on over-approximation using widening.

8.4 Allowing State Separation Violations

The state separation hypothesis applies to the language presented thus far; the choice of whether a variable is modeled by an abstract value or mostly-concrete value could be made based on types, and operations on abstract values can only occur in the abstract interpreter and similarly for mostly-concrete values. However, the state separation hypothesis is violated if we add primitive types, interfaces, and common library types such as `Hashtable` to our supported subset of Java; e.g., frameworks may interrogate or modify an integer produced by application code. Although we could restrict the use of primitives and libraries to only application or framework code, and further require that all implementers of an interface must be either application or framework classes, such a restriction would be unrealistic. We therefore describe how to handle these features as well as direct mutation of application objects by the framework and vice versa.

8.4.1 Primitive Types and Operations. Our limited subset of Java supports only integers with basic arithmetic operations and comparisons. Integer values are represented in the mostly-concrete interpreter with a sum type: $\wp(\mathbb{N})^\top + \widehat{A}$. When an integer abstraction of type $\widehat{A}$ flows into a framework method we automatically lift it into the sum type. We use $\widehat{A}$ as the abstract representation of objects and integers: in practice we expect that internally $\widehat{A}$ is a union of objects and integers abstractions.

When executing an arithmetic or comparison operation, CONCERTO checks if both operands are of type $\wp(\mathbb{N})^\top$. If so, then the interpreter executes the concrete arithmetic operation lifted to the $\wp(\mathbb{N})^\top$ domain. If one or both of the operands are of type $\widehat{A}$, then CONCERTO uses a `lift` $: \wp(\mathbb{N})^\top \to \widehat{A}$ method exported by the abstract interpreter to convert the powerset representation into an $\widehat{A}$ representation. After conversion, CONCERTO calls methods exposed by the abstract interpretation that perform the primitive operations on elements of $\widehat{A}$. Finally, in cases such as array indexing, CONCERTO may need to transform $\wp(\mathbb{N})^\top + \widehat{A}$ into $\wp(\mathbb{N})^\top$. The abstract interpretation must also expose a function `lower` $: \widehat{A} \to \wp(\mathbb{N})^\top$. A sound choice for this function is to simply return $\top_F$.

At calls from the framework into the application, CONCERTO passes the sum representation directly to the abstract interpretation which may lift this sum type into a native representation. In practice, abstract interpretations gain precision by using the sum representation for integers, and lifting to a native representation on demand for arithmetic and comparison operations.

8.4.2 Interfaces and Library Types. An interface I may have implementers in the application and the framework. Thus, given a variable/field of type I, it may be unknown whether that variable/field contains an instance of a framework or application class. We resolve this ambiguity by requiring that the abstract interpretation and mostly-concrete interpreter use a combined object representation $\wp(V_f) \times \widehat{A}$ for values with an interface type. Intuitively, $\langle v_f, \widehat{a} \rangle : \wp(V_f)^\top \times \widehat{A}$ represents *either* a framework object that is abstracted by v_f *or* an application object abstracted by $\widehat{a}$. In the case that one of the components is the least element in its respective lattice then the interpretation of combined value is the interpretation of the non-bottom component. In principle, we could have used this product representation for primitive types, but found that in practice the sum type representation is easier to use.

When the mostly-concrete interpreter encounters a method call on an interface, it splits the receiver into its two components, and then performs a concrete method call on the concrete component while simultaneously performing the abstract method call by yielding into the abstract interpreter. The results from both method calls are then merged using the appropriate join operations and execution continues. Concerto exports an API that performs the symmetric operation for interface calls encountered in application code by the abstract interpreter.

Both framework and application code may use the same standard libraries, breaking our simplifying assumption that the types used in the application and framework are disjoint. For example, Java types like `ArrayList`, `HashMap`, etc. are ubiquitous. Concerto supports these *library types* using the same product representation used for interfaces. Unlike interfaces, the choice of whether to model library objects using a mostly-concrete or abstract representation is not based on static type information, but on the allocation site. For example, an `ArrayList` allocated in the application will be modeled using abstract values, whereas an `ArrayList` allocated in the framework will be modeled mostly-concretely. This approach is possible because we assume that library types, like interfaces, only export object-oriented interfaces and Concerto does not need to support framework code that directly modifies the internal state of a library objects allocated in the application and vice versa.

8.4.3 Direct Field Access. We have so far assumed that the framework never directly accesses or mutates application object fields and vice-versa. As argued in Section 2, we expect this assumption holds for the vast majority framework-based applications. However, our approach can still be applied in the cases where the assumption does not hold, albeit with some precision penalty. We describe how to support framework code that reads and writes application object fields; the approach for the application code directly accessing framework fields is symmetric.

We first consider the case where framework code reads an application object field. Recall that we model operations on application fields as *aop*s and that we use an extremely coarse model for *aop*s in our mostly-concrete semantics. Thus, we can soundly model reads of application object fields as simply returning the maximal element from the appropriate lattice. For example, the mostly-concrete interpreter may use $\top_{\widehat{A}}$ to model the value read from a field of application type. Any future operations on this read value will necessarily be imprecise; the exact extent of this imprecision will depend on how the read value is used by the program.

Our approach for handling direct mutations of application fields is broadly similar. As field writes are modeled as *aop*s that side effect the abstract heap, mutations are coarsely modeled by simply havocing the abstract heap, i.e. the interpretation of a field write returns $\top_{\widehat{H_a}}$, where $\widehat{H_a}$ is the domain of abstract heaps. This coarse approach will cause greater imprecision compared to the field read case above, but we contend that direct field mutations are exceedingly rare in practice. As frameworks and applications are developed independently from one another, the framework implementation cannot guarantee any application-specific object invariants are preserved by a field mutation. A similar argument applies for direct mutations of framework field by the application code. Thus, we expect any imprecision introduced by this coarse modeling to be limited for real-world applications.

9 EVALUATION

To evaluate the feasibility and benefits of our combined analysis approach, we implemented a small "web application framework" called Yawn (Your Analysis' Worst Nightmare) in the subset of Java supported by our prototype implementation. Yawn implements an accept loop which parses requests received on our language's nondeterministic IO stream and routes these requests to application defined handlers. Yawn contains several difficult-to-analyze features found in real-world frameworks, including dependency injection, an embedded Lisp interpreter, and indirect flow. The dependency injection component and the Lisp interpreter heavily used reflection, and the run-time behavior of all three features is determined by the contents of a configuration file.

Table 1. Summary of abstract interpretations. **CS** is the context-sensitivity of the analysis if applicable. **PS** indicates if the analysis is path-sensitive.

Name	**CS?**	**Heap**	**Domain**	**Relational?**	**PS?**
Pta	No	Type-based	Reaching Types	No	No
IFlow	Caller Method	Type-based	Access Paths/Reaching Types	No	No
Abc	Call site 1-CFA	Abstract Location	Pentagons/Abstract Locations	Yes	Yes

Table 2. Number of reports issued and execution times of the interpreters with (**Conc.**) and without (**Std. AI**) Concerto. t/o indicates a timeout. For a discussion of the Pta results, see the main text.

Analysis	**Time (Conc.)**	**Time (Std. AI)**	**Reports (Conc.)**	**Reports (Std. AI)**
Pta	4.7 s	1282.7 s	—	—
Abc	8.8 s	t/o	0	2
IFlow	4.6 s	t/o	3	7

We implemented a simple application using the Yawn framework. The application's primary functionality is implemented as a collection of request handlers which perform simple mathematical operations (e.g., summing two integers) on request parameters. The application uses in-memory state and a simulated database layer implemented as standalone modules. These handlers and modules are constructed and wired together using Yawn's dependency injection mechanism. Yawn also includes a filtering mechanism to preprocess requests. Our application applies a filter that uses Yawn's embedded Lisp interpreter to run a filtering program specified in the application's configuration file.

Next, we implemented three abstract interpreters that use different abstract domains, heap representations, and context sensitivity. These analyses are summarized in Table 1. Pta performs VTA-style [Sundaresan et al. 2000] call-graph construction using a type-based heap where abstract addresses are sets of type names. IFlow is an information flow integrity analysis [Denning 1976] to find flows from untrusted sources to sensitive sinks. It uses the caller method as the context when analyzing a callee. For the heap abstraction, IFlow reuses the type-based heap from the Pta interpreter, and extends the reaching type domain with k-limited access paths [Deutsch 1994; Jones and Muchnick 1979] that track which heap locations are tainted. Finally, the most complex (and expensive) analysis is Abc, an array bounds checker. Abc uses call site 1-CFA for contexts, and uses an abstract heap that maps abstract locations to abstract objects. An abstract location is pair consisting of an allocation site and the context in which the allocation occurred. Object values are abstract by powersets of abstract locations. Integers are abstracted with an approximation of the reduced product [Cousot and Cousot 1979b] of the Interval domain *Intv* and inequalities between access paths, giving a weakly relational Pentagon domain [Logozzo and Fähndrich 2008]. As the choice of $\widehat{A}$ must be non-relational, the abstract representation of integers in $\widehat{A}$ is simply *Intv*. Abc also propagates inequalities induced by comparison operators making it partially path sensitive.

Finally, we ran each interpreter over the application twice: once using Concerto and once with standard abstract interpretation. Each run had a one hour time budget, and we measured the total time of each run. After each run, we collected call-graph information (Pta) or any alarms reported (Abc & IFlow). In the event of a timeout, we collected any information computed up to that point.

For every analysis, Concerto vastly outperformed plain abstract interpretation as shown in Table 2. Under plain abstract interpretation, Abc and IFlow timed out while Pta took approximately 275× longer than combined interpretation. The Abc and IFlow timeouts were caused by enormous strongly connected components due to sound but imprecise modeling of Yawn's use of reflection and indirection. Even with widening, propagating information through these cycles overwhelmed the abstract interpreters. Pta also encountered large strongly connected components, but the lack of context-sensitivity and simplicity of the abstract domains mitigated the performance impacts.

Further, the quality of analysis results was significantly worse with plain interpretation compared to Concerto. To evaluate the precision of Abc and IFlow, we classified the alarms reported as either true or false positives. Combined interpretation correctly found all 3 information leaks in our test application and also successfully verified that the application was free of out-of-bounds array accesses. In contrast, plain abstract interpretation reported 7 leaks and and 2 out-of-bounds accesses, respectively. For the array bounds checker, all these reports were false positives, and all but 3 were false positives for the information flow analysis. Additionally, as these results were collected after timeouts, they represent a lower bound on the imprecision of Abc and IFlow.

Pta does not find bugs, but produces a call-graph for a downstream analysis; we include it in our experiments to demonstrate the impact of combined interpretation on resolving reflective invocations in framework code. As a representative example, under plain interpretation Pta resolved the reflective allocations in the dependency injection facility to 38 possible types, compared to just 15 types under combined interpretation. Similarly, within the Lisp interpreter, plain interpretation resolved the reflective invocations to as many as 30 and no fewer than 8 callees, whereas combined interpretation resolved every invocation to a single callee.

10 RELATED WORK

There has been considerable work on improving analysis precision for framework-based applications by using the information in configuration files, particularly for Android applications [Arzt et al. 2014; Blackshear et al. 2015]. The Frameworks for Frameworks system by Sridharan et al. [2011] generalizes these efforts, providing a framework for writing framework models. Unfortunately, manually providing a framework model is an enormous effort that is rarely reusable across different frameworks. In contrast, once the concrete semantics for a language are specified, combined interpretation can in principle be used with any framework without further effort.

Several static analyses have resolved calls that use reflection [Barros et al. 2015; Li et al. 2015; Smaragdakis et al. 2015]. To succeed, these techniques rely on programs using reflection APIs with constant strings and type-casting the results in stylized ways. These patterns rarely occur in framework implementations. In addition, these techniques are tailored to specific APIs (i.e., Java's reflection API) and adapting them to other domains requires non-trivial work by the analysis designer. Moreover, frameworks are difficult to analyze for reasons beyond their use of reflection.

Many efforts have combined dynamic and static analysis, yielding "blended" or "hybrid" analyses. Some hybrid analyses use information recorded during dynamic executions of the program to improve the static-analysis precision [Csallner et al. 2008; Dufour et al. 2007; Grech et al. 2017; Ren and Foster 2016; Wei and Ryder 2013]. For example, TamiFlex by Bodden et al. [2011] instruments an application as it runs a representative workload to record the callees of reflective calls. These approaches generally suffer from unsoundness; even representative workloads rarely exercise all possible execution paths. Symmetrically, other researchers have run a dynamic analysis seeded with information produced by a static analysis [Balzarotti et al. 2008; Chugh et al. 2009; Lam et al. 2008]. As with the above approaches, any dynamic analysis will almost certainly be unsound.

Other researchers have explored other approaches to combining analyses. For example, [Fink et al. 2008] runs a series of increasingly precise analyses to prune false positives left by earlier analyses. Work on tunable analyses of JavaScript [Ko et al. 2015] uses the results of a pre-analysis to restrict the abstract states explored by an abstract interpretation. [Ferrara 2014] explores combining a value and heap analysis to improve precision in abstract interpretation. Their approach embeds information produced by the heap analysis into the domain of the value analysis. This embedding closely mirrors how Concerto embeds abstract values into the mostly-concrete state and vice versa. Other researchers have combined different execution strategies for different portions of code [Avgerinos et al. 2014; Chipounov et al. 2011]. For example, [Chipounov et al. 2011] switch to *fully* concrete execution on portions of the system under test. Their approach requires concretizing symbolic values and then checking that doing so does not prune any feasible paths. Concretization is performed lazily, which parallels how Concerto threads abstract values through mostly-concrete interpretation. However, due to the state separation hypothesis, Concerto can avoid *all* concretization.

Our combined interpretation bears similarity to concolic testing [Godefroid et al. 2005; Sen and Agha 2006; Sen et al. 2005]. Concolic testing performs symbolic and concrete execution in parallel, but falls back on concrete values in the symbolic interpreter when it encounters an expression outside of the logic of the underlying theorem prover. This approach is similar to how Concerto uses concrete execution to precisely reason about difficult-to-analyze code. Similarly, as noted in Section 2, our technique is similar to partial evaluation [Futamura 1999; Mogensen 1995]. However, partial evaluation is typically used for optimization [Brown and Palsberg 2017; Jones et al. 1993], and the full resolution of reflection is usually an orthogonal concern. We are unaware of work trying to use partial evaluation to handle difficult-to-analyze framework code for sound program analysis, and we believe Concerto is less brittle than a partial evaluation approach to this problem.

Finally, many researchers have improved analysis precision by combining abstract domains [Brauer et al. 2010; Cousot et al. 2006; Fähndrich and Logozzo 2010; Ferrara 2010; Laviron and Logozzo 2009; Logozzo and Fähndrich 2008; Toubhans et al. 2013; Zanioli et al. 2012], via the reduced product [Cousot and Cousot 1979b], reduced tensor product [Nielson 1985], etc. Astree [Cousot et al. 2005] in particular is an industry tool that computes an approximate reduced product by propagating information through a tree of abstract domains [Cousot et al. 2006]. Our approach could be formalized as a degenerate case of this framework, where the exchange of information between the abstract and mostly-concrete interpreters takes place at the transition points via the communication channels described in [Cousot et al. 2006]. However, while the reduced product domain found in Astree and other abstract interpretations typically exchange information about the same program point between multiple domains, under Concerto each program statement is analyzed by either mostly-concrete or abstract interpretation. Further, our subfixpoint iteration strategy is significantly different from the one described in [Cousot et al. 2006].

11 CONCLUSIONS AND FUTURE WORK

We presented Concerto, a framework for soundly combining concrete and abstract interpretation. Concerto targets framework-based applications that use difficult-to-analyze reflection, metaprogramming, and abstractions. This combination is possible because framework-based applications in practice satisfy a state separation hypothesis which Concerto exploits to opaquely embed abstract values into a mostly-concrete interpreter. Our combination supports any abstract interpreter that satisfies a modest set of conditions, and yields significant improvements in initial experiments with a research prototype. Further generalizing our framework, in particular supporting alternative definitions of soundness (e.g., via concretization function [Cousot and Cousot 1992b]) is one area of future work. We also plan to extend our initial research prototype to support the full Java language and real-world frameworks.

ACKNOWLEDGMENTS

We thank James Wilcox, Ben Hardekopf, Bill Harris, James Bornholt, and Jared Roesch for their comments on early drafts of this work. Thanks also to Pavel Panchekha for his insight on our proofs. Finally, we thank the anonymous reviewers, in particular our shepherd, for their comments and feedback on this work. This material is based upon work supported in part by DARPA under agreement number FA8750-16-2-0032.

REFERENCES

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI*.

Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing symbolic execution with veritesting. In *ICSE*.

Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2008. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Symposium on Security and Privacy*.

Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Michael D Ernst, et al. 2015. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *ASE*.

Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. 2015. Droidel: A general approach to android framework modeling. In *SOAP*.

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*.

François Bourdoncle. 1993. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and their Applications*.

Guillaume Brat, Klaus Havelund, SeungJoon Park, and Willem Visser. 2000. Java PathFinder-second generation of a Java model checker. In *Workshop on Advances in Verification*.

Jörg Brauer, Thomas Noll, and Bastian Schlich. 2010. Interval analysis of microcontroller code using abstract interpretation of hardware and software. In *Workshop on Software & Compilers for Embedded Systems*.

Matt Brown and Jens Palsberg. 2017. Jones-optimal partial evaluation by specialization-safe normalization. *Proc. ACM Program. Lang.* 2, POPL (2017), 14.

Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*.

Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *ASPLOS*.

Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. 2009. Staged Information Flow for Javascript. In *PLDI*.

Patrick Cousot. 1977. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. *Res. rep. RR* 88 (1977).

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*.

Patrick Cousot and Radhia Cousot. 1979a. Constructive versions of Tarski's fixed point theorems. *Pacific journal of Mathematics* 82, 1 (1979).

Patrick Cousot and Radhia Cousot. 1979b. Systematic design of program analysis frameworks. In *POPL*. ACM.

Patrick Cousot and Radhia Cousot. 1992a. Abstract interpretation and application to logic programs. *The Journal of Logic Programming* 13, 2-3 (1992), 103–179.

Patrick Cousot and Radhia Cousot. 1992b. Abstract interpretation frameworks. *Journal of logic and computation* 2, 4 (1992).

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTRÉE analyzer. In *ESOP*.

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2006. Combination of abstractions in the ASTRÉE static analyzer. In *Annual Asian Computing Science Conference*.

Christoph Csallner, Yannis Smaragdakis, and Tao Xie. 2008. DSD-Crasher: A hybrid analysis tool for bug finding. *TOSEM* 17, 2 (2008), 8.

Dorothy E Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (1976).

Alain Deutsch. 1994. Interprocedural May-alias Analysis for Pointers: Beyond K-limiting. In *PLDI*.

Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2007. Blended analysis for performance understanding of framework-based applications. In *ISSTA*.

Manuel Fähndrich and Francesco Logozzo. 2010. Static contract checking with abstract interpretation. In *Formal Verification of Object-Oriented Software*.

Pietro Ferrara. 2010. Static type analysis of pattern matching by abstract interpretation. In *Formal Techniques for Distributed Systems*. Springer, 186–200.

Pietro Ferrara. 2014. Generic combination of heap and value analyses in abstract interpretation. In *VMCAI*.

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *TOSEM* 17, 2 (2008).

Martin Fowler. 2004. Inversion of control containers and the dependency injection pattern. (2004).

Yoshihiko Futamura. 1999. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *PLDI*.

Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2017. Heaps Don't Lie: Countering Unsoundness with Heap Snapshots. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 68 (2017).

Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Peter Sestoft.

Neil D Jones and Steven S Muchnick. 1979. Flow analysis and optimization of LISP-like structures. In *POPL*.

John B Kam and Jeffrey D Ullman. 1977. Monotone data flow analysis frameworks. *Acta Informatica* 7, 3 (1977), 305–317.

Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. 2015. Practically Tunable Static Analysis Framework for Large-Scale JavaScript Applications. In *ASE*.

Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley. 2008. Securing web applications with static and dynamic information flow tracking. In *Partial Evaluation and Semantics-based Program Manipulation*.

Vincent Laviron and Francesco Logozzo. 2009. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *VMCAI*.

Yue Li, Tian Tan, and Jingling Xue. 2015. Effective soundness-guided reflection analysis. In *SAS*.

Francesco Logozzo and Manuel Fähndrich. 2008. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *Symposium on Applied Computing*.

Torben Mogensen. 1995. Self-applicable online partial evaluation of the pure lambda calculus. In *Symposium on Partial Evaluation and Semantics-based Program Manipulation*.

Flemming Nielson. 1985. Tensor products generalize the relational data flow analysis method. In *4th Hungarian Computer Science Conference*. 211–225.

Brianna M. Ren and Jeffrey S. Foster. 2016. Just-in-time Static Type Checking for Dynamic Languages. In *PLDI*.

Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*.

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *ESEC/FSE*.

Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More sound static handling of Java reflection. In *ASPLAS*.

Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. 2011. F4F: Taint Analysis of Framework-based Web Applications. In *OOPSLA*.

Gregory T Sullivan. 2001. Dynamic partial evaluation. In *Programs as Data Objects*. 238–256.

Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical Virtual Method Call Resolution for Java. In *OOPSLA*.

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics* 5, 2 (1955).

Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. 2013. Reduced product combination of abstract domains for shapes. In *VMCAI*.

Shiyi Wei and Barbara G Ryder. 2013. Practical blended taint analysis for JavaScript. In *ISSTA*.

Matteo Zanioli, Pietro Ferrara, and Agostino Cortesi. 2012. SAILS: Static Analysis of Information Leakage with Sample. In *Symposium on Applied Computing*.

A PROOFS FOR SECTION 4

A.1 Soundness of $I_\top$

We now prove that $\alpha_F \circ F \sqsubseteq I_\top \circ \alpha_F$. We first note the following fact about α_F that we will exploit during our proofs:

$$\forall \vec{\ell}, \vec{\ell}', r, r', x, y.\mathcal{V}(r[\vec{\ell}], x) \subseteq \mathcal{V}(r'[\vec{\ell}], y) \Rightarrow \alpha_F(r)[\vec{\ell}][x] \sqsubseteq \alpha_F(r')[\vec{\ell}'][y] \tag{20}$$

$$\alpha_v(\emptyset) = \bot_{\widehat{A}} \tag{21}$$

To begin, we will ignore the "initial state" term from the definition of F, and first prove $I_\top$ sound with respect to:

$$F_0(r)[\vec{\ell}] = \begin{cases} \bigcup_{\substack{p \in pred(\ell) \\ in \in r[p^\bullet \rightsquigarrow \ell^\circ]}} step^F(in, \ell) & \vec{\ell} = s^\circ \rightsquigarrow s^\bullet \\ F(r)[\vec{\ell}] & o.w. \end{cases}$$

which is the original concrete semantic function without the intialization term. We first prove that the inequality holds for some arbitrary r and $\ell^\circ \rightsquigarrow \ell^\bullet$. We first need the following lemmas.

Lemma 1. *$\forall \vec{\ell}, r, \widetilde{s}.\alpha_F([\vec{\ell} \mapsto \mathcal{FL}])[\vec{\ell}] \sqsubseteq \widetilde{s} \Rightarrow \alpha_F \circ F_0(r)[\vec{\ell}] \sqsubseteq \widetilde{s}$ where $\mathcal{FL} = F_0(r)[\vec{\ell}]$ and $[\vec{\ell} \mapsto s] : R$ is shorthand for $\bot_R[\vec{\ell} \mapsto s]$.*

Proof. For every x, $\mathcal{V}(F_0(r)[\vec{\ell}], x) = \mathcal{V}([\vec{\ell} \mapsto \mathcal{FL}][\vec{\ell}], x)$, and thus by (20), $\alpha_F([\vec{\ell} \mapsto \mathcal{FL}])[\vec{\ell}][x] = \alpha_F(F_0(r))[\vec{\ell}][x]$. By transitivity, we have $\alpha_F \circ F_0(r)[\vec{\ell}] \sqsubseteq \widetilde{s}$. □

Intuitively, Lemma 1 ensures that to establish pointwise inequality, it suffices to consider the abstracted state for each $\vec{\ell}$ individually.

Lemma 2.

$$\begin{aligned} &\forall r, \ell, p \in pred(\ell), \widetilde{s}. \\ &\quad \left(\forall \langle in, E\rangle \in r[p^\bullet \rightsquigarrow \ell^\circ].\alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell] \sqsubseteq \widetilde{s}\right) \\ &\quad \Rightarrow \alpha_F([\ell \mapsto \mathcal{FL}])[\ell] \sqsubseteq \widetilde{s} \end{aligned}$$

where $\mathcal{FL}$ is defined as in Lemma 1.

Proof. First, we show that $\bot_{\widetilde{R}}[\ell \mapsto \widetilde{s}]$ is an upper bound for the set $\{\alpha_F(m) \mid m \in \mathcal{M}\}$, where $\mathcal{M} = \{[\ell \mapsto step^F(\langle in, E\rangle, \ell)] \mid p \in pred(\ell) \wedge \langle in, E\rangle \in r[p^\bullet \rightsquigarrow \ell^\circ]\}$. Let m be some element of $\mathcal{M}$. By assumption, $\alpha_F(m)[\ell] \sqsubseteq \widetilde{s}$, so it remains to show that $\forall \ell'.\alpha_F(m)[\ell'] \sqsubseteq \bot_{\widetilde{R}}[\ell \mapsto \widetilde{s}][\ell'] = \bot_{\widetilde{S}}$. This follows from the fact that for any x, $\alpha_F(\bot_R)[\ell'][x] = \bot_{\widetilde{V}}$ (where $\bot_{\widetilde{V}}$ is $\bot_{\widehat{A}}$ or $\emptyset$, depending on the type of x) and by (20) and $\mathcal{V}(\bot_R[\ell'], x) = \emptyset = \mathcal{V}(m[\ell'], x)$, whence we have that $\alpha_F(m)[\ell'][x] = \bot_{\widetilde{V}}$.

Next, observe that $[\ell \mapsto \mathcal{FL}] = [\ell \mapsto \bigcup_{\substack{p \in pred(\ell) \\ \langle in, E\rangle \in r[p^\bullet \rightsquigarrow \ell^\circ]}} step^F(\langle in, E\rangle, \ell)]$ which is equivalent to $\bigsqcup_{\substack{p \in pred(\ell) \\ \langle in, E\rangle \in r[p^\bullet \rightsquigarrow \ell^\circ]}} [\ell \mapsto step^F(\langle in, E\rangle, \ell)]$, i.e., $\bigsqcup \mathcal{M}$. As α_F preserves least upper bounds, and from the fact that $\bot_{\widetilde{R}}[\ell \mapsto \widetilde{s}]$ is an upper bound of the set $\{\alpha_F(m) \mid m \in \mathcal{M}\}$, we have that: $\alpha_F(\bigsqcup_{\substack{p \in pred(\ell) \\ \langle in, E\rangle \in r[p^\bullet \rightsquigarrow \ell^\circ]}} [\ell \mapsto step^F(\langle in, E\rangle, \ell)]) \sqsubseteq \bot_{\widetilde{R}}[\ell \mapsto \widetilde{s}]$, from which it is immediate that $\alpha_F([\ell \mapsto \mathcal{FL}])[\ell] \sqsubseteq \widetilde{s}$. □

Lemma 2 implies that to show an abstract state $\widetilde{s}$ over-approximates the result of stepping all incoming concrete states, it suffices to show that $\widetilde{s}$ over-approximates stepping each individual incoming state. We now show that for any ℓ and r, $I_\top(\alpha_F(r))[\ell^\circ \rightsquigarrow \ell^\bullet]$ is such an $\widetilde{s}$.

Lemma 3. $\forall r, \ell, in, p \in pred(\ell).\langle in, E\rangle \in r[p^\bullet \rightsquigarrow \ell^\circ]$:

(1)

$$\forall x, y. \mathcal{V}(step^F(\langle in, E\rangle, \ell), x) \subseteq \mathcal{V}(\{\langle in, E\rangle\}, y)$$
$$\Rightarrow \alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell][x] \sqsubseteq \Big(\bigsqcup_{p' \in pred(\ell)} \alpha_F(r)[p'^{\bullet} \rightsquigarrow \ell^{\circ}]\Big)[y]$$

(2) $\alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell] \sqsubseteq I_{\top}(\alpha_F(r))[\ell]$

Proof.

(1) $\langle in, E\rangle \in r[p^{\bullet} \rightsquigarrow \ell^{\circ}]$ implies that $\mathcal{V}(\{\langle in, E\rangle\}, y) \subseteq \mathcal{V}(r[p^{\bullet} \rightsquigarrow \ell^{\circ}], y)$, whence by transitivity and (20) we then have:

$$\alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell][x] \sqsubseteq \alpha_F(r)[p^{\bullet} \rightsquigarrow \ell^{\circ}][y]$$

Transivity and least upper bounds gives

$$\alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell][x] \sqsubseteq \alpha_F(r)[p^{\bullet} \rightsquigarrow \ell^{\circ}][y] \sqsubseteq \bigsqcup_{p' \in pred(\ell)} \alpha_F(r)[p'^{\bullet} \rightsquigarrow \ell^{\circ}][y]$$

(2) By the pointwise definition of the domains, it suffices to show that:

$$\forall x. \alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell][x] \sqsubseteq step^{\top}(\widetilde{in}, \ell)[x]$$
$$\sqsubseteq I_{\top}(\alpha_F(r))[\ell][x]$$

where $\widetilde{in} = \bigsqcup_{p' \in pred(\ell)} \alpha_F(r)[p'^{\bullet} \rightsquigarrow \ell^{\circ}]$. As an immediate consequence of part 1 above, we can conclude that if all states produced by $step^F$ retain the original value of x, and that $step^{\top}$ likewise does not change the value of x from $\widetilde{in}$, the inequality holds at x.

We proceed by the branches of $step^{\top}$:

Case (5): $step^{\top}$ simply returns $\widetilde{in}$, and $step^F(\langle in, E\rangle, \ell)$ must return $\{\langle in, E\rangle\}$ and by the above argument, the inequality holds for all x.

Case (6): We must only establish the inequality for the left-hand side of the assignment. As $\mathcal{V}(step^F(\langle in, E\rangle, \ell), lhs) = \mathcal{V}(\{\langle in, E\rangle\}, rhs)$, we have by part (1) above:

$$\alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell][lhs] \sqsubseteq \widetilde{in}[rhs] = step^{\top}(\widetilde{in}, \ell)[lhs]$$

Case (7): As above, we need only establish that the relationship holds for the lhs. By definition of $step^F$, $\mathcal{V}(step^F(\langle in, E\rangle, \ell), lhs) = \mathcal{V}(\{\langle in[lhs \mapsto [\![bc_f]\!]], E\rangle\}, lhs) = \{[\![bc_f]\!]\}$. By the definition of α_F:

$$\alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell][lhs] = \{[\![bc_f]\!]\}$$
$$= \widetilde{in}[lhs \mapsto \{[\![bc_f]\!]\}][lhs]$$
$$= step^{\top}(\widetilde{in}, \ell)[lhs]$$

Case (8): Except for the lhs, every variable in each state produced by $step^F$ retains its original value in in, and thus we must only show that the inequality holds for lhs. Define the set $O = \{r \mid \langle r, E'\rangle \in [\![fop]\!](E, in[v_1], \ldots, in[v_n]\}$ where $v_1, \ldots, v_n$ are the variable arguments to the fop. By the definition of α_F, $\alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell][lhs] = O$. Consider now the value of $step^{\top}(\widetilde{in}, \ell)[lhs]$. If it is $\top_F$, then the result trivially again holds. Otherwise, we can again conclude that $step^{\top}(\widetilde{in}, \ell)[lhs] = \widetilde{[\![fop]\!]}(\widetilde{in}[v_1], \ldots, \widetilde{in}[v_n]) \neq \top_F$, whence by the the definition of $\widetilde{[\![fop]\!]}$, we may further conclude that $\widetilde{in}[v_1], \ldots, \widetilde{in}[v_n] \neq \top_F$. By the definition of least upper bounds and from the definition of α_F, we must then have: $in[v_1] \in \widetilde{in}[v_1] \wedge \ldots \wedge in[v_n] \in \widetilde{in}[v_n]$. As $E \in \mathcal{E}$, from the definition of $\widetilde{[\![fop]\!]}$ we have $O \subseteq \widetilde{[\![fop]\!]}(\widetilde{in}[v_1], \ldots, \widetilde{in}[v_n]) = step^{\top}(\widetilde{in}, \ell)[lhs]$ as required.

Case (9): By definition of α_F, $\alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell][lhs] = \widehat{v} : \widehat{A} \sqsubseteq \top_{\widehat{A}} = step^{\top}(\widetilde{in}, \ell)[lhs]$.

□

Let us now prove soundness of $I_\top$ w.r.t F_0 for $\ell^\circ \rightsquigarrow \ell^\bullet$:

Lemma 4. $\forall \ell, r.\alpha_F \circ F_0(r)[\ell^\circ \rightsquigarrow \ell^\bullet] \sqsubseteq I_\top \circ \alpha_F(r)[\ell^\circ \rightsquigarrow \ell^\bullet]$

Proof. Immediate from Lemmas 1 and 2, and Lemma 3 part 2. □

Lemma 5. $\forall p^\bullet \rightsquigarrow \ell^\circ, r.\alpha_F \circ F_0(r)[p^\bullet \rightsquigarrow \ell^\circ] \sqsubseteq I_\top \circ \alpha_F(r)[p^\bullet \rightsquigarrow \ell^\circ]$

Proof. As $\alpha_F(\bot_R) = \bot_{\widetilde{R}}$, if $F_0(r)[p^\bullet \rightsquigarrow \ell^\circ] = \emptyset$ the inequality trivially holds. Otherwise, by definition, $F_0(r)[p^\bullet \rightsquigarrow \ell^\circ] \subseteq r[p]$, and thus $\alpha_F(F_0(r))[p^\bullet \rightsquigarrow \ell^\circ] \sqsubseteq \alpha_F(r)[p]$ (where p is shorthand for $p^\circ \rightsquigarrow p^\bullet$). If $\widetilde{\mathcal{FT}}(\alpha_F(r), p^\bullet \rightsquigarrow \ell^\circ)$ is true, then $I_\top(\alpha_F(r))[p^\bullet \rightsquigarrow \ell^\circ] = \alpha_F(r)[p]$ as required. It therefore suffices to show that $F_0(r)[p^\bullet \rightsquigarrow \ell^\circ] \neq \emptyset \Rightarrow \widetilde{\mathcal{FT}}(\alpha_F(r), p^\bullet \rightsquigarrow \ell^\circ)$. We proceed by cases on the form of $prog[p]$. If $p \in \ell_f$ and $prog[p]$ is conditional over x and y with target ℓ, then there exists some state $s \in r[p]$ such that $s[x]\llbracket\texttt{<=>}\rrbracket s[y]$. If either $\alpha_F(r)[p][x] = \top_F$ or $\alpha_F(r)[p][y] = \top_F$, then by definition $\alpha_F(r)[p][x]\widetilde{\llbracket\texttt{<=>}\rrbracket}\alpha_F(r)[p][y]$ is trivially true. Otherwise, by the definition of α_F, $s[x] \in \alpha_F(r)[p][x]$ and similarly for $s[y]$ and $\alpha_F(r)[p][y]$, thus $\alpha_F(r)[p][x]\widetilde{\llbracket\texttt{<=>}\rrbracket}\alpha_F(r)[p][y] \Rightarrow \widetilde{\mathcal{FT}}(\alpha_F(r), p^\bullet \rightsquigarrow \ell^\circ)$. A similar argument holds for when $p \in \ell_f$, and $prog[p]$ is a condition with fallthrough target ℓ. Finally, for any other statement, $\widetilde{\mathcal{FT}}$ is trivially true. □

We can now prove our main result:

Proof of Theorem 2. For $\vec{\ell} \neq s^\circ \rightsquigarrow s^\bullet$, $\alpha_F \circ F_0$ is equivalent to $\alpha_F \circ F$, whence the result holds from Lemmas 4 and 5. It remains to show that the proof holds at $\vec{\ell} = s^\circ \rightsquigarrow s^\bullet$ (which we will abbreviate in the following as s).

We must show that, for some arbitrary r, $\alpha_F(F(r))[s] \sqsubseteq I_\top(\alpha_F(r))[s]$. By reasoning similar to Lemma 1, it suffices to show:

$$\alpha_F([s \mapsto \bigcup_{\substack{p \in pred(\ell) \\ in \in r[p^\bullet \rightsquigarrow \ell^\circ]}} step^F(in, \ell)] \sqcup [s \mapsto \bigcup_{e \in \iota_{\mathcal{E}}} step^F(\langle \iota_S, e\rangle, \ell)]) \sqsubseteq I_\top(\alpha_F(r))$$

As α_F is a complete join morphism, this is equivalent to showing that:

$$\alpha_F([s \mapsto F_0(r)[\vec{\ell}]]) \sqcup \alpha_F([s \mapsto \bigcup_{e \in \iota_{\mathcal{E}}} step^F(\langle \iota_S, e\rangle, \ell)]) \sqsubseteq I_\top(\alpha_F(r))$$

By Lemmas 2 and 3, we have the necessary bound for the first term of the join. It therefore remains to show that:

$$\alpha_F([s \mapsto \bigcup_{e \in \iota_{\mathcal{E}}} step^F(\langle \iota_S, e\rangle, \ell)]) \sqsubseteq I_\top(\alpha_F(r))$$

for which it suffices to to show that:

$$\alpha_F([s \mapsto \bigcup_{e \in \iota_{\mathcal{E}}} step^F(\langle \iota_S, e\rangle, \ell)])[s] \sqsubseteq step^\top(\bot_{\widetilde{S}}, \ell) \sqsubseteq I_\top(\alpha_F(r))[s]$$

This result holds as a special case of the reasoning from the proof of Lemma 3 part 2.

We conclude by noting that we were careful to show that soundness of $I_\top$ is preserved when any $\widetilde{\llbracket fop \rrbracket}$ returns $\top_F$ or if the abstraction process returns $\top_F$, which justifies the soundness of our finitization approach. □

A.2 Soundness of Combined Interpretation

Proof of Theorem 3. We proceed by cases on the components of $\overline{R}$, for some input argument X.

Case $\vec{\ell} \in \mathcal{L}_A$: We must show that $\alpha_C \circ F(X)[\vec{\ell}] = \alpha_A(F(X))[\vec{\ell}] \sqsubseteq C(\alpha_C(X))[\vec{\ell}] = \widehat{F} \circ \widehat{inj}(\alpha_C(X))[\vec{\ell}] = \widehat{F}(\widehat{inj}(\alpha_C(X)))[\vec{\ell}]$. By assumption, we have that $\alpha_A(F(X))[\vec{\ell}] \sqsubseteq \widehat{F}(\alpha_A(X))[\vec{\ell}]$, so it suffices to show that $\alpha_A(X) \sqsubseteq \widehat{inj}(\alpha_C(X))$. For labels in $\mathcal{L}_A$, the inequality is immediate. For labels in $\mathcal{L}_F$, we have $\alpha_A(X)[\vec{\ell}] \sqsubseteq \widehat{\tau}(\alpha_F(X)[\vec{\ell}]) = \widehat{inj}(\alpha_C(X))[\vec{\ell}]$ from assumption (12).

Case $\vec{\ell} \in \mathcal{L}_F$: As above, by the soundness of $I_\top$, we have that $\alpha_F(F(X))[\vec{\ell}] \sqsubseteq I_\top(\alpha_F(X))[\vec{\ell}]$, so it suffices to show that $\alpha_F(X) \sqsubseteq \widetilde{inj}(\alpha_C(X))$. As in the above case, the inequality immediately holds at labels in $\mathcal{L}_F$. Otherwise we have $\forall \vec{\ell'} \in \mathcal{L}_A.\alpha_F(X)[\vec{\ell'}] \sqsubseteq \widetilde{\tau}(\alpha_A(X)) = \widetilde{inj}(\alpha_C(X))[\vec{\ell'}]$ by (13).

□

A.3 Increased Precision

LEMMA 6. $C \circ \widehat{proj} \sqsubseteq_{\widehat{R} \to \widehat{R}} \widehat{proj} \circ \widehat{F}$

PROOF. Suffices to show that, for some $\widehat{r}$, and for all $\vec{\ell}$, $C \circ \widehat{proj}(\widehat{r})[\vec{\ell}] \sqsubseteq \widehat{proj} \circ \widehat{F}(\widehat{r})[\vec{\ell}]$. Before proceeding we observe that: $\widetilde{inj} \circ \widehat{proj} = \dot{\widetilde{\tau}}$ and $\widehat{inj} \circ \widehat{proj} = id$.

By cases on whether $\vec{\ell} \in \mathcal{L}_F$ or $\vec{\ell} \in \mathcal{L}_A$:

Case $\vec{\ell} \in \mathcal{L}_A$: Then $C \circ \widehat{proj}(\widehat{r})[\vec{\ell}] = \widehat{F} \circ \widehat{inj} \circ \widehat{proj}(\widehat{r})[\vec{\ell}]$. Because $\widehat{inj} \circ \widehat{proj} = id$, we must show $\widehat{F}(\widehat{r})[\vec{\ell}] \sqsubseteq \widehat{proj} \circ \widehat{F}(\widehat{r})[\vec{\ell}] = \widehat{F}(\widehat{r})[\vec{\ell}]$ which holds trivially as $\widehat{proj}$ is the identity at $\mathcal{L}_A$.

Case $\vec{\ell} \in \mathcal{L}_F$: Then we must show that $I_\top \circ \widetilde{inj} \circ \widehat{proj}(\widehat{r})[\vec{\ell}] = I_\top(\dot{\widetilde{\tau}}(\widehat{r}))[\vec{\ell}] \sqsubseteq \widehat{proj} \circ \widehat{F}(\widehat{r})[\vec{\ell}] = \widetilde{\tau}(\widehat{F}(\widehat{r})[\vec{\ell}]) = \dot{\widetilde{\tau}}(\widehat{F}(\widehat{r}))[\vec{\ell}]$. By assumption, $LB \circ \dot{\widetilde{\tau}} \sqsubseteq \dot{\widetilde{\tau}} \circ \widehat{F}$, so it suffices to show that $I_\top \circ \dot{\widetilde{\tau}}(\widehat{r})[\vec{\ell}] \sqsubseteq LB \circ \dot{\widetilde{\tau}}(r)[\vec{\ell}]$. By cases on the form of $\vec{\ell} \in \mathcal{L}_F$:

Subcase $\vec{\ell} = \ell_f^\circ \rightsquigarrow \ell_f^\bullet$: From the definitions of $I_\top$ and LB, we must show that:

$$step^\top(\widetilde{s}, \ell_f) \sqsubseteq step^{LB}(\widetilde{s}, \ell_f)$$

where $\widetilde{s} = \bigsqcup_{p \in pred(\ell_f)} \dot{\widetilde{\tau}}(\widehat{r})[p^\bullet \rightsquigarrow \ell_f^\circ]$. For branches (15) and (16) in $step^{LB}$, the corresponding branch (5) and (6) in $step^\top$ clearly yield identical results. As branch (18) returns $\widetilde{s}[x \mapsto \top_F]$, and the corresponding branches (7) and (8) of $step^\top$ produce states of the form: $\widetilde{s}[x \mapsto v]$ (where v is some member of $\wp(V_f)^\top$), the inequality trivially holds. Finally, we do not have to consider branch (17) of $step^{LB}$ nor branch (9) of $step^F$ because the syntactic constraints of the language rule such cases out for a label of the form ℓ_f.

Finally, if $\ell = s$, we must additionally show that $step^\top(\bot_{\widetilde{S}}, s) \sqsubseteq step^{LB}(\bot_{\widetilde{S}}, s)$. This follows by the same reasoning as above.

Subcase $\vec{\ell} = \ell_f^\bullet \rightsquigarrow \ell^\circ$: By definition, $I_\top(\dot{\widetilde{\tau}}(\widehat{r}))[\ell_f^\bullet \rightsquigarrow \ell^\circ]$ either returns $\dot{\widetilde{\tau}}(\widehat{r})[\ell_f^\circ \rightsquigarrow \ell_f^\bullet]$ or $\bot$. As $LB(\dot{\widetilde{\tau}}(\widehat{r}))[\ell_f^\bullet \rightsquigarrow \ell^\circ] = \dot{\widetilde{\tau}}(\widehat{r})[\ell_f^\circ \rightsquigarrow \ell_f^\bullet]$, the inequality trivially holds.

□

LEMMA 7. *Assume for two complete lattices T and $\widehat{T}$, a complete-join morphism $\mu : \widehat{T} \to T$, and two functions $F : T \to T$, $\widehat{F} : \widehat{T} \to \widehat{T}$ we have that $F \circ \mu \sqsubseteq_{\widehat{T} \to T} \mu \circ \widehat{F}$. Assume then we have two Ord termed sequences defined via transfinite recursion as:*

$$\begin{array}{rcl} F^0 & = & \bot \\ F^{\delta+1} & = & F(F^\delta) \\ F^\lambda & = & \bigsqcup_{\beta<\lambda} F^\beta \end{array} \qquad\qquad \begin{array}{rcl} \widehat{F}^0 & = & \bot \\ \widehat{F}^{\delta+1} & = & \widehat{F}(\widehat{F}^\delta) \\ \widehat{F}^\lambda & = & \bigsqcup_{\beta<\lambda} \widehat{F}^\beta \end{array}$$

Then for any $\delta \in Ord$, $F^\delta \sqsubseteq \mu(\widehat{F}^\delta)$

PROOF. By transfinite induction.

Case $\delta = 0$: Then $F^0 = \bot_T = \mu(\bot_{\widehat{T}}) = \mu(\widehat{F}^0)$, where the equality between $\bot$ terms comes from the fact that μ is a complete join-morphism.

Case $\delta + 1$: Assume $F^{\delta} \sqsubseteq \mu(\widehat{F}^{\delta})$. By the monotonicity of F, we have that $F^{\delta+1} = F(F^{\delta}) \sqsubseteq F(\mu(\widehat{F}^{\delta}))$, so it suffices to show that $F(\mu(\widehat{F}^{\delta})) \sqsubseteq \mu(\widehat{F}^{\delta+1})$. Further, by definition $\widehat{F}^{\delta+1} = \widehat{F}(\widehat{F}^{\delta})$, thus we must show that $F \circ \mu(\widehat{F}^{\delta}) \sqsubseteq \mu \circ \widehat{F}(\widehat{F}^{\delta})$, which holds by assumption.

Case $\delta = \lambda$: By the inductive hypothesis, $\forall \beta < \lambda . F^{\beta} \sqsubseteq \mu(\widehat{F}^{\beta})$, thus

$$F^{\lambda} = \bigsqcup_{\beta<\lambda} F^{\beta} \sqsubseteq \bigsqcup_{\beta<\lambda} \mu(\widehat{F}^{\beta}) = \mu(\bigsqcup_{\beta<\lambda} \widehat{F}^{\beta}) = \mu(\widehat{F}^{\lambda})$$

Where the second-to-last equality follow from μ being a complete join morphism.

□

Proof of Theorem 4. From the definition of $\mathit{lfp}\, C$ and $\mathit{lfp}\, \widehat{F}$ as the limit of the ordinal termed sequences defined as in Lemma 7, Lemma 6, that $\widehat{\mathit{proj}}$ is a complete join morphism from part 3 of assumption Eq. (19), Lemma 7 gives us that $\mathit{lfp}\, C \sqsubseteq \widehat{\mathit{proj}}(\mathit{lfp}\, \widehat{F})$. As $\widehat{\mathit{inj}}$ is monotone, we have: $\widehat{\mathit{inj}}(\mathit{lfp}\, C) \sqsubseteq \widehat{\mathit{inj}} \circ \widehat{\mathit{proj}}(\mathit{lfp}\, \widehat{F})$, whence we have $\widehat{\mathit{inj}}(\mathit{lfp}\, C) \sqsubseteq \mathit{lfp}\, \widehat{F}$ as $\widehat{\mathit{inj}} \circ \widehat{\mathit{proj}} = \mathit{id}$ as observed in the proof of Lemma 6. □

B PROOFS FOR SECTION 5

We note that Eq. (20) still applies under the updated definition of $\mathcal{V}$.

Updated Proof for Theorem 2. We begin by noting that for the intraprocedural fragment of the language, the proofs of soundness given Appendix A.1 generalize naturally to the new definition of states. As an informal argument as to why: note that the definition of $\mathcal{V}$ given above only considers the values of variables in the stack frame of the currently executing method. From the definition of α_F, this in turn implies that the values in the abstracted state are exclusively determined by the concrete values in the active state frame. Finally, as the intraprocedural fragment of the language manipulates only the active stack frame, the arguments made in Appendix A.1 translate naturally the extended state definition. Finally, although we have extended the intraprocedural fragment of the language with a `return` statement, the semantics of this statement given by $\mathit{step}^{\top}$ and step^{F} are simply a special case of an assignment statement, and the argument given in the proof of Lemma 3 easily applies to this statement as well.

We therefore only concern ourselves with establishing soundness with respect to the newly defined interprocedural fragment. By Lemma 1, it suffices to establish soundness for each new type of edge individually. Without loss of generality, we proceed to prove soundness for some arbitrary instance of each type of edge and for some r.

Case $\ell_c^{\circ} \rightsquigarrow \ell_c^{\bullet}$: By reasoning similar to Appendix A.1, it suffices to show that

$$\alpha_F([\ell_c \mapsto \bigsqcup_{p \in pred(\ell_c)} r[p^{\bullet} \rightsquigarrow \ell_c^{\circ}]])[\ell_c] \sqsubseteq I_{\top}(\alpha_F(r))[\ell_c]$$
$$= \bigsqcup_{p \in pred(\ell_c)} \alpha_F(r)[p^{\bullet} \rightsquigarrow \ell_c^{\circ}]$$

From the definition of the domain R, we have:

$$\alpha_F([\ell_c \mapsto \bigsqcup_{p \in pred(\ell_c)} r[p^{\bullet} \rightsquigarrow \ell_c^{\circ}]]) = \alpha_F(\bigsqcup_{\substack{p \in pred(\ell_c) \\ in \in r[p^{\bullet} \rightsquigarrow \ell_c^{\circ}]}} [\ell_c \mapsto \{in\}])$$
$$= \bigsqcup_{\substack{p \in pred(\ell_c) \\ in \in r[p^{\bullet} \rightsquigarrow \ell_c^{\circ}]}} \alpha_F([\ell_c \mapsto \{in\}])$$

By the definition of least upper bounds, it suffices to show that, for some arbitrary $p' \in pred(\ell_c), in \in r[p'^\bullet \rightsquigarrow \ell_c^\circ]$:

$$\alpha_F([\ell_c \mapsto \{in\}])[\ell_c] \sqsubseteq \bigsqcup_{p \in pred(\ell_c)} \alpha_F(r)[p^\bullet \rightsquigarrow \ell_c^\circ]$$

We show that $\alpha_F([\ell_c \mapsto \{in\}])[\ell_c] \sqsubseteq \alpha_F(r)[p'^\bullet \rightsquigarrow \ell_c^\circ]$, whence by the definition of least upper bounds and transitivity we will have the desired result.
For some arbitrary variable v, from the fact that $in \in r[p'^\bullet \rightsquigarrow \ell_c^\circ]$, it is immediate that:

$$\mathcal{V}([\ell_c \mapsto \{in\}][\ell_c], v) = \mathcal{V}(\{in\}, v) \subseteq \mathcal{V}(r[p'^\bullet \rightsquigarrow \ell_c^\circ], v)$$

whence by Eq. (20) we have that $\alpha_F([\ell_c \mapsto \{in\}])[\ell_c][v] \sqsubseteq \alpha_F(r)[p'^\bullet \rightsquigarrow \ell_c^\circ][v]$, as required.

Case $p^\bullet \rightsquigarrow \ell_r^\circ$: Trivial, by the definition of α_F and reasoning similar to the above.

Case $\ell_c^\bullet \rightsquigarrow \ell'^\circ$: We first abbreviate $\ell_c^\bullet \rightsquigarrow \ell'^\circ$ as $\vec{c}$ and (as usual) $\ell_c^\circ \rightsquigarrow \ell_c^\bullet$ as ℓ_c. By reasoning similar to that in case $\ell_c^\circ \rightsquigarrow \ell_c^\bullet$, it suffices to show that:

$$\bigsqcup_{\langle s \circ s_r, \mathcal{R}, E\rangle \in r[\ell_c]} \alpha_F([\vec{c} \mapsto \{\langle s \circ s_r \circ [p \mapsto s_r[y]], \mathcal{R} \circ \ell, E\rangle\}])[\vec{c}] \sqsubseteq [p \mapsto \alpha_F(r)[\ell_c][y]]$$

for which it suffies to show, for some arbitrary $\langle s \circ s_r, \mathcal{R}, E\rangle \in r[\ell_c]$ that:

$$\alpha_F([\vec{c} \mapsto \mathcal{K}])[\vec{c}] \sqsubseteq [p \mapsto \alpha_F(r)[\ell_c][y]]$$

where $\mathcal{K} = \{\langle s \circ s_r \circ [p \mapsto s_r[y]], \mathcal{R} \circ \ell, E\rangle\}$. From the new definition on $\mathcal{V}$, for any variable $v \neq p$, it is immediate that $\mathcal{V}(\mathcal{K}, v) = \emptyset$, whence we have that:

$$\begin{aligned}\alpha_F([\vec{c} \mapsto \mathcal{K}])[\vec{c}][v] &= \alpha_F(\bot_R)[\vec{c}][v] \\ &= \bot_{\widetilde{R}}[\vec{c}][v] = \Big[p \mapsto \alpha_F(r)[\ell_c][y]\Big][v]\end{aligned}$$

It therefore remains to show that

$$\alpha_F([\vec{c} \mapsto \mathcal{K}])[\vec{c}][p] \sqsubseteq \Big[p \mapsto \alpha_F(r)[\ell_c][y]\Big][p] = \alpha_F(r)[\ell_c][y]$$

From the definition of $\mathcal{K}$, we have that

$$\mathcal{V}(\mathcal{K}, p) = \{s_r[y]\} \subseteq \mathcal{V}(r[\ell_c], y)$$

whence by Eq. (20) we have the desired result.

Case $\ell_r^\circ \rightsquigarrow \ell_r^\bullet$: We will prove that, for some some arbitrary $r, p \in pred(\ell_r)$, $\langle s \circ s_r \circ s_c, \mathcal{R}, E\rangle \in r[p^\bullet \rightsquigarrow \ell_r^\circ]$ where $\langle s \circ s_r, \mathcal{R}, E\rangle \in r[\ell_c^\circ \rightsquigarrow \ell_c^\bullet]$ that:

$$\alpha_F([\ell_r \mapsto \mathcal{J}])[\ell_r] \sqsubseteq \alpha_F(r)[\ell_c]\Big[x \mapsto \alpha_F(r)[p^\bullet \rightsquigarrow \ell_r^\circ][\rho]\Big]$$

where $\mathcal{J} = \{\langle s \circ s_r[x \mapsto s_c[\rho]], \mathcal{R}, E\rangle\}$
For any variable $v \neq x$, we have that $\mathcal{V}(\mathcal{J}, v) \subseteq \mathcal{V}(r[\ell_c], v)$, whence the inequality holds by Eq. (20). For the variable x, we have that $\mathcal{V}(\mathcal{J}, x) \subseteq \mathcal{V}(r[p^\bullet \rightsquigarrow \ell_r^\circ], \rho)$, whence the inequality again holds by assumption on α_F.

□

C FORMALISMS AND PROOFS FOR SUBFIXPOINT ITERATION

Before defining subfixpoint iteration, we introduce the following notation. Let $m : T \to U$ be a map, where U is a complete lattice. For $S \subseteq T$, define:

$$m\|_S = \lambda x. \begin{cases} m[x] & x \in S \\ \bot_U & o.w. \end{cases}$$

We will abuse notation and for an element $\langle \widehat{m}, \widetilde{m}\rangle : \overline{R}$, define $\langle \widehat{m}, \widetilde{m}\rangle|_{\mathcal{L}_A} = \widehat{m}$ and $\langle \widehat{m}, \widetilde{m}\rangle|_{\mathcal{L}_F} = \widetilde{m}$.

We define the subfixpoint iteration process as a function $M : \overline{R} \to \overline{R}$:

$$M(X) = \left\langle \bigsqcup_{i<\omega} U^i(\widehat{inj}(X))|_{\mathcal{L}_A}, \bigsqcup_{i<\omega} T(\widetilde{inj}(X)\|_{\mathcal{L}_A})^i(\bot)|_{\mathcal{L}_F} \right\rangle \qquad U(X) = X\|_{\mathcal{L}_F} \sqcup \widehat{F}(X)\|_{\mathcal{L}_A}$$

$$T(m)(X) = m\|_{\mathcal{L}_A} \sqcup I_\top(X)\|_{\mathcal{L}_F}$$

In the above definitions, the U and T play the role of iterating the two interpreters to fixpoint.[8] In the definition of U, after abstract semantic function $\widehat{F}$ is applied to the input argument, the results from this application in $\mathcal{L}_F$ are discarded in favor of the values in the input argument X. Thus, while iterating $U^i(\widehat{inj}(X))$, the states at flow edges $\mathcal{L}_F$ are effectively fixed to $\widehat{inj}(X)\|_{\mathcal{L}_F}$, while the states at flow edges $\mathcal{L}_A$ evolve through repeated application of $\widehat{F}$. The definition of T is similar; except the state information in application code is fixed while the state information in the framework evolves via application of $I_\top$. Unlike the $\sqcup_{i<\omega}U^i(\widehat{inj}(X))$ term, $\sqcup_{i<\omega}T(\widetilde{inj}(X)\|_{\mathcal{L}_A})^i(\bot)$ iterates from bottom, discarding information computed in previous rounds of iteration as described in Section 6.

We first prove that M is monotone.

Lemma 8.

(1) $\forall m, m', X, X' : \widetilde{R}.m \sqsubseteq m' \wedge X \sqsubseteq X' \Rightarrow T(m)(X) \sqsubseteq T(m')(X')$
(2) $\forall m, m', X, X' : \widetilde{R}.m \sqsubseteq m' \wedge X \sqsubseteq X' \Rightarrow \sqcup_{i<\omega}T(m)^i(X) \sqsubseteq \sqcup_{i<\omega}T(m')^i(X')$
(3) $\forall X, X' : \widehat{R}.X \sqsubseteq X' \Rightarrow U(X) \sqsubseteq U(X')$
(4) $\forall X, X' : \widehat{R}.X \sqsubseteq X' \Rightarrow \sqcup_{i<\omega}U^i(X) \sqsubseteq \sqcup_{i<\omega}U^i(X')$

Proof.

(1) By cases on whether $\vec{\ell} \in \mathcal{L}_A$ or $\vec{\ell} \in \mathcal{L}_F$:
Case $\vec{\ell} \in \mathcal{L}_F$: $T(m)(X)[\vec{\ell}] = I_\top(X)[\vec{\ell}] \sqsubseteq I_\top(X')[\vec{\ell}] = T(m')(X')[\vec{\ell}]$ by the monotonicity of $I_\top$ (by Lemma 18, proved below).
Case $\vec{\ell} \in \mathcal{L}_A$: $T(m)(X)[\vec{\ell}] = m[\vec{\ell}] \sqsubseteq m'[\vec{\ell}] = T(m')(X')[\vec{\ell}]$ by assumption.
(2) By induction, using the monotonicity of T shown above it can be shown that $\forall i < \omega . T(m)^i(X) \sqsubseteq T(m')^i(X')$, whence the result follows from the definition of least upper bounds.
(3) Immediate from the fact that $\widehat{F}$ is monotone.
(4) By similar proof to (2) above.

□

Theorem 5. *M is monotone.*

Proof. Immediate from parts 2 and 4 of Lemma 8 and the monotonicity of $\widetilde{\tau}$ and $\widehat{\tau}$. □

We next define two ordinal termed sequences $\mathfrak{C}^{\delta \in \mu}$ and $\mathfrak{M}^{\delta \in \mu}$, where μ is an ordinal defined as in [Cousot and Cousot 1979a], via transfinite recursion as:

$$\begin{array}{rclrcl} \mathfrak{C}^0 & = & \bot_{\overline{R}} & \mathfrak{M}^0 & = & \bot_{\overline{R}} \\ \mathfrak{C}^{\delta+1} & = & C(\mathfrak{C}^\delta) & \mathfrak{M}^{\delta+1} & = & M(\mathfrak{M}^\delta) \\ \mathfrak{C}^\lambda & = & \bigsqcup_{\beta<\lambda} \mathfrak{C}^\beta & \mathfrak{M}^\lambda & = & \bigsqcup_{\beta<\lambda} \mathfrak{M}^\beta \end{array}$$

As both M and C are monotone functions over complete lattices, then by Corollary 3.3 of [Cousot and Cousot 1979a], the limit of $\mathfrak{C}^{\delta \in \mu}$, $\mathfrak{C}^\epsilon$, exists and is the least fixpoint of C and the limit of $\mathfrak{M}^{\delta \in \mu}$ also exists and is a least fixed point of M.

Before proceeding, we first show how to (almost) instantiate the asynchronous chaotic iteration with memory strategy defined by [Cousot 1977] to give an equivalent definition of subfixpoint

[8]We have not justified here that the term $\sqcup_{i<\omega}U^i(\widehat{inj}(X))$ is a fixpoint of U, nor that a fixed point of this function exists. However, we chose this definition for the parallel to the widening definitions presented below; which we prove converges to a fixed-point. If $\widehat{F}$ is ω-upper-continuous, then the U term converges to a fixpoint, as proved in Appendix C.5.

iteration. (In the following, for continuity of notation, we will use F as the monotone function over a lattice $(L^n)^m \to L^n$, and *not* the concrete semantic function of Section 3.2.)

Assume some isomorphism ξ between $\{0, \ldots, |\mathcal{L}|\}$ and $\mathcal{L}$. We can define an indexed representation of an element of $\overline{R}$, where X_i corresponds to the mostly-concrete or abstract state at $\xi(i)$. We take $m = 3$, and for any i such that $\xi(i) \in \mathcal{L}_A$, define:

$$\begin{aligned} F_i(I, E, D) &= \widehat{F}(\widehat{inj}(\eta(I, E)))[\xi(i)] \\ \eta(I, E) &= \left\langle \lambda\vec{\ell}_a : \mathcal{L}_A . I_{\xi^{-1}(\vec{\ell}_a)},\ \lambda\vec{\ell}_f : \mathcal{L}_F : E_{\xi^{-1}(\vec{\ell}_f)} \right\rangle \end{aligned}$$

η transforms the indexed representation into the product-of-maps representation expected by $\widehat{inj}$. When constructing the abstract state map it uses the values from I, and those from E to construct the mostly-concrete map.

For i such that $\xi(i) \in \mathcal{L}_F$, define:

$$\begin{aligned} F_i(I, E, D) &= I_{\top}(\widetilde{inj}(\chi(D)))[\xi(i)] \\ \chi(D) &= \left\langle \lambda\vec{\ell}_a : \mathcal{L}_A . D_{\xi^{-1}(\vec{\ell}_a)},\ \lambda\vec{\ell}_f : \mathcal{L}_F . D_{\xi^{-1}(\vec{\ell}_f)} \right\rangle \end{aligned}$$

Like η, χ transforms the indexed representation into a map representation, using D for states in $\mathcal{L}_F$ and $\mathcal{L}_A$. Clearly, $F \circ \sigma = C$.

We will now define the ordinal termed sequences S of indices that specify the values of I, E, and D. When indexing S, we take $I \equiv 1$, $E \equiv 2$, and $D \equiv 3$. Then:

$$\begin{aligned} (S_I^{\delta+1})_i &= \delta \\ (S_E^{\delta+1})_i &= \lambda \text{ (where } \lambda \text{ is the largest limit ordinal} \leq \delta) \\ (S_D^{\delta+1})_i &= \begin{cases} S_E^{\delta+1} & \xi(i) \in \mathcal{L}_A \\ 0 & \xi(i) \in \mathcal{L}_F \land \delta = \lambda \text{ (where } \lambda \text{ is a limit ordinal)} \\ \delta & o.w. \end{cases} \end{aligned}$$

We do not define the value of S_j at the limit ordinals as they are not used during iteration, we assume they take some value consistent with the requirements given in [Cousot 1977].

These definitions imply that I always holds the values computed in the last round of iteration, E holds the values computed at the most recent limit ordinal, and D holds the values of at the most recent limit ordinal for states at $\mathcal{L}_A$, $\bot_{\widetilde{S}}$ for $\mathcal{L}_F$ at the immediate successor to some limit ordinal, and the value from the previous iteration otherwise (we assume that $X_i^0 = \bot$ for all i).

It should be clear that the value computed at $M(\bot_{\overline{R}})[\vec{\ell}]$ corresponds to the values computed at $X_{\xi^{-1}(\vec{\ell})}$ at ω. However, our limiting behavior at all other limit ordinals is different: M takes the limits of new sequences, whereas at the limit ordinals the sequence defined in [Cousot 1977] take the limit over the entire preceding sequence of values. In addition, the definition of S_D violates requirement 3.1.(d). As a result, although we have that the limit of $\mathfrak{M}^\delta$ exists and is a least fixed point of M, it is not immediate that this is equal to $\mathit{lfp}\, C$.

We therefore directly prove our desired result:

THEOREM 6. $\mathit{lfp}\, M = \mathit{lfp}\, C$

C.1 Technical Lemmas

Before proving Theorem 6 we need the following technical lemmas.

LEMMA 9. *The function $T(m)(X)$ is ω-upper-continuous in its second argument.*

Proof.

$$\begin{aligned}\bigsqcup_n T(m)(x_n) &= \bigsqcup_n I_\top(x_n)\|_{\mathcal{L}_F} \sqcup m\|_{\mathcal{L}_A} \\ &= m\|_{\mathcal{L}_A} \sqcup \bigsqcup_n I_\top(x_n)\|_{\mathcal{L}_F} \\ &= m\|_{\mathcal{L}_A} \sqcup I_\top(\bigsqcup_n x_n)\|_{\mathcal{L}_F} \text{ (by the } \omega\text{-upper-continuity of } I_\top \text{, see Appendix C.1.1)} \\ &= T(m)(\bigsqcup_n x_n)\end{aligned}$$

□

Theorem 7. *The limit of $T(m)^{(i\in\mathbb{N})}(\bot)$ exists and is a fixpoint of $T(m)$ for any m.*

Proof. Follows immediately from Lemma 9 and Kleene's fixpoint theorem. □

Let us recall the following facts from [Cousot and Cousot 1979a]:

(1) The least upper bound of a (potentially infinite) set S of post-fixed points of a monotone function F is itself a post-fixed point of F

(2) For a monotone function $g : L \to L$ on a complete lattice L, the ordinal termed sequence defined as:

$$\begin{aligned}\mathfrak{G}^0 &= \bot_L \\ \mathfrak{G}^{\delta+1} &= g(\mathfrak{G}^\delta) \\ \mathfrak{G}^\lambda &= \bigsqcup_{\alpha<\lambda} \mathfrak{G}^\alpha\end{aligned}$$

is increasing

Next define the family of sequences $\mathfrak{T}_m^\delta, \delta \in Ord$ via transfinite recursion as:

$$\begin{aligned}\mathfrak{T}_m^0 &= \bot_{\widetilde{R}} \\ \mathfrak{T}_m^{\delta+1} &= T(m)(\mathfrak{T}_m^\delta) \\ \mathfrak{T}_m^\lambda &= \bigsqcup_{\alpha<\lambda} \mathfrak{T}_m^\alpha\end{aligned}$$

By the monotonicity of T we have that all such sequences are increasing, and by Theorem 7 the limit of all sequences exist and is $\mathfrak{T}_m^\omega$.

Lemma 10. $\forall m, \delta \in Ord.\mathfrak{T}_m^\delta \sqsubseteq \mathfrak{T}_m^\omega$

Proof. Immediate from the definition of $\mathfrak{T}_m^\delta$ and by the fact that $\mathfrak{T}_m^\omega$ is the least fixpoint of $T(m)$. □

C.1.1 Continuity of $I_\top$. We now demonstrate the ω-upper-continuity of $I_\top$ on increasing chains of $\widetilde{R}$.

Lemma 11. *For any increasing chain $x_i : \wp(V_f)^\top, v \in \bigsqcup_i x_i \Rightarrow \exists k.v \in x_k$*

Proof. First, observe that by the definition of least upper bound, if $(\bigsqcup_i x_i) \neq \top_F$, then $\forall i, x_i, \neq \top_F$. Thus, by the definition of $\bigsqcup$ in $\wp(V_f)^\top$ as a union over the sets x_i, $v \in \bigsqcup_i x_i$ implies there is some set in the chain x_i, that contains v. We let j be the smallest such x_j containing x_j. □

Corollary 11.1. *For an increasing chain of products $c_i \equiv \langle x_1, \dots, x_n\rangle_i : \wp(V_f)^\top \times \dots \wp(V_f)^\top$, if there exists some $\langle v_1, \dots, v_n\rangle$ such that $\forall j.v_j \in \pi_j(\bigsqcup_i c_i)$, then there exists some k such that $\forall j.v_j \in \pi_j(c_k)$.*

PROOF. By pointwise application of Lemma 11, for each j there must be some k_j such that $v_j \in \pi_j(c_k)$. Let m be the max over these k_j. As the chain c_i is increasing, if $v_j \in \pi_j(c_{k_j})$, then necessarily $v_j \in \pi_j(c_m)$. □

LEMMA 12. *$\widetilde{[\![fop]\!]}$ is ω-upper-continuous for all increasing chains $arg_i : \wp(V_f)^\top \times \ldots \times \wp(V_f)^\top$*

PROOF. Consider the case where, for some j and k, $\pi_j(arg_k) = \top_F$. Then $\widetilde{[\![fop]\!]}(arg_k) = \top_F$, and further $\pi_j(\bigsqcup_i arg_i) = \top_F$ whence $\widetilde{[\![fop]\!]}(\bigsqcup_i arg_i) = \top_F$.

Consider now the case where the chain contains no $\top_F$ values. Then, by the definition of $[\![fop]\!]$ and $\widetilde{[\![fop]\!]}$, the sequence $\widetilde{[\![fop]\!]}(arg_i)$ will not contain $\top_F$, and thus $\bigsqcup_i \widetilde{[\![fop]\!]}(arg_i)$ will not equal $\top_F$, and similarly $\widetilde{[\![fop]\!]}(\bigsqcup_i arg_i) \neq \top_F$.

It therefore suffices to show that $v \in \bigsqcup_i \widetilde{[\![fop]\!]}(arg_i) \Leftrightarrow v \in \widetilde{[\![fop]\!]}(\bigsqcup_i arg_i)$.

Case $\Rightarrow$: By the definition of lub and $\widetilde{[\![fop]\!]}$, $\exists k.v \in \widetilde{[\![fop]\!]}(arg_k)$. But then $\forall j.\pi_j(arg_k) \sqsubseteq \pi_j(\bigsqcup_i arg_i)$ and thus by definition of $\widetilde{[\![fop]\!]}$, $v \in \widetilde{[\![fop]\!]}(\bigsqcup_i arg_i)$.

Case $\Leftarrow$: If $v \in \widetilde{[\![fop]\!]}(\bigsqcup_i arg_i)$, then there must exist some set of values v_j and some $E \in \mathcal{E}$ such that $\forall j.v_j \in \pi_j(\bigsqcup_i arg_i)$ and $[\![fop]\!](E, v_0, \ldots, v_n) = \langle v, \_\rangle$. By Corollary 11.1, there must be some k, such that $\forall j.v_j \in \pi_j(arg_k)$, whence $v \in \widetilde{[\![fop]\!]}(arg_k) \sqsubseteq \bigsqcup_i \widetilde{[\![fop]\!]}(arg_i)$.

□

LEMMA 13. *$\widetilde{[\![fop]\!]}$ is pointwise monotone.*

PROOF. Consider the case of some $\widetilde{[\![fop]\!]}(v_0, \ldots, v_n)$ where one of $v_i = \top_F$. Then, $\widetilde{[\![fop]\!]}(v_0, \ldots, v_n) = \top_F$, and for any v'_j, $v_j \sqsubseteq v'_j \Rightarrow v'_j = \top_F$, whence $\widetilde{[\![fop]\!]}(v'_0, \ldots, v'_n) = \top_F$, for any values v'_i pointwise greater than v_i. Next, consider the case of some arguments $v_i \sqsubseteq v'_i$, where $\forall j.v_j \neq \top_F$. Then $\widetilde{[\![fop]\!]}(v_0, \ldots, v_n) \neq \top_F$. If any of v'_i is $\top_F$, then monotonicity holds trivially. Finally, consider when, for each j, $v_j \subseteq v'_j$. Then by the definition of $\widetilde{[\![fop]\!]}$, it is immediate that $\widetilde{[\![fop]\!]}(v_0, \ldots, v_n) \subseteq \widetilde{[\![fop]\!]}(v'_0, \ldots, v'_n)$. □

LEMMA 14. *If $s_i : \widetilde{S}$ is an increasing sequence, and if for such sequences $\bigsqcup_i f(s_i) = f(\bigsqcup_i s_i)$, then: $\forall x.Q = \bigsqcup_i s_i[x \mapsto f(s_i)] = (\bigsqcup s_i)[x \mapsto f(\bigsqcup_i s_i)] = R$.*

PROOF. $Q[x] = \bigsqcup_i f(s_i) = f(\bigsqcup_i s_i) = R[x]$ and $Q[y \neq x] = \bigsqcup_i s_i[y] = (\bigsqcup_i s_i)[y] = R[y]$. □

LEMMA 15. *For an increasing sequence $s_i : \widetilde{S}, \forall \ell.\ \bigsqcup_i step^\top(s_i, \ell) = step^\top(\bigsqcup_i s_i, \ell)$.*

PROOF. By cases on the branch taken in $step^\top$:

Branch (5): Trivial

Branches (6, 7, 9): Follows from the Lemma 14 and that $\lambda\_.\top_{\widehat{A}}$, $\lambda\_.\{[\![bc_f]\!]\}$, and $\lambda s.s[y]$ are trivially continuous on increasing sequences.

Branch (8): From Lemma 14 and Lemma 12.

□

LEMMA 16. *$step^\top$ is monotone.*

PROOF. Immediate from the monotonicity of $\widetilde{[\![fop]\!]}$ (Lemma 13), and the definition of $step^\top$. □

LEMMA 17.

(1) $\forall \widetilde{r}, \widetilde{r}', p^\bullet \rightsquigarrow \ell^\circ.\widetilde{\mathcal{FT}}(\widetilde{r}, p^\bullet \rightsquigarrow \ell^\circ) \land \widetilde{r} \sqsubseteq \widetilde{r}' \Rightarrow \widetilde{\mathcal{FT}}(\widetilde{r}', p^\bullet \rightsquigarrow \ell^\circ)$

(2) *For an increasing sequence* r_i, $\forall p^\bullet \rightsquigarrow \ell^\circ.\widetilde{\mathcal{FT}}(\bigsqcup_i r_i, p^\bullet \rightsquigarrow \ell^\circ) \Leftrightarrow \exists j.\widetilde{\mathcal{FT}}(r_j, p^\bullet \rightsquigarrow \ell^\circ)$

Proof.

(1) Let $\widetilde{s} = \widetilde{r}[p]$ and $\widetilde{s}' = \widetilde{r}'[p]$. Consider by cases why $\widetilde{\mathcal{FT}}$ returned true:

Branch 2 Then $\widetilde{s}[x]\widetilde{[\![\texttt{<=>}]\!]}\widetilde{s}[y]$, thus either $\widetilde{s}[x] = \top_F, \widetilde{s}[y] = \top_F$, or $v \in \widetilde{s}[x]$ and $v' \in \widetilde{s}[y]$ such that $v[\![\texttt{<=>}]\!]v'$. In the former two cases, $\widetilde{s}'[x]$ or $\widetilde{s}'[y]$ must also $\top_F$, and thus $\widetilde{s}'[x]\widetilde{[\![\texttt{<=>}]\!]}\widetilde{s}'[y]$. Consider the final case. Then by definition of $\sqsubseteq$, $\widetilde{s}'[x] = \top_F$ or $\widetilde{s}[x] \subseteq \widetilde{s}'[x]$ and similarly for $\widetilde{s}[y]$ and $\widetilde{s}'[y]$. In the case where either $\widetilde{s}'[x]$ or $\widetilde{s}'[y]$ is $\top_F$, then $\widetilde{s}'[x]\widetilde{[\![\texttt{<=>}]\!]}\widetilde{s}'[y]$ trivially returns true. Otherwise, we have $v \in \widetilde{s}'[x]$ and similarly for v' and $\widetilde{s}'[y]$, and thus by definition $\widetilde{s}'[x]\widetilde{[\![\texttt{<=>}]\!]}\widetilde{s}'[y]$ holds.

Branch 3 By similar reasoning to the above, but on the definition of $\widetilde{[\![\texttt{<≠>}]\!]}$.

Branch 4 $\widetilde{\mathcal{FT}}$ must always return true.

(2) For the forward impliciation, consider by cases why $\widetilde{\mathcal{FT}}$ returned true:

Branch 2 By similar reasoning to the above, and without loss of generality, let us consider the case where $(\bigsqcup_i r_i)[p][x] = \top_F$. Then, there must exist some j such that $r_j[p][x] = \top_F$, and thus $\widetilde{\mathcal{FT}}(r_j, p^\bullet \rightsquigarrow \ell^\circ)$ trivially returns true. The case when $(\bigsqcup_i r_i)[p][y] = \top_F$ follows from similar reasoning. Finally, let us consider the case where $v \in (\bigsqcup_i r_i)[p][x]$, $v' \in (\bigsqcup_i r_i)[p][y]$, and $v[\![\texttt{<=>}]\!]v'$. By Corollary 11.1, there must exists some j, such that $v \in r_j[p][x]$ and $v' \in r_j[p][y]$, whence $r_j[p][x]\widetilde{[\![\texttt{<=>}]\!]}r_j[p][y]$ and thus $\widetilde{\mathcal{FT}}(r_j, p^\bullet \rightsquigarrow \ell^\circ)$ must return true.

Branch 3 By similar reasoning to the above.

Branch 4 $\forall j.\widetilde{\mathcal{FT}}(r_j, p, \ell)$ must be true.

The backward impliciation follows immediately from part one of the lemma and from the fact that $\forall j.r_j \sqsubseteq \bigsqcup_i r_i$.

□

Lemma 18. *$I_\top$ is monotone*

Proof. Consider some $\widetilde{r} \sqsubseteq \widetilde{r}'$, and some arbitrary $\ell^\circ \rightsquigarrow \ell^\bullet$:

$$\begin{aligned} I_\top(\widetilde{r})[\ell^\circ \rightsquigarrow \ell^\bullet] &= step^\top(\bigsqcup_{p \in pred(\ell)} \widetilde{r}[p^\bullet \rightsquigarrow \ell^\circ], \ell) \\ &\sqsubseteq step^\top(\bigsqcup_{p \in pred(\ell)} \widetilde{r}'[p^\bullet \rightsquigarrow \ell^\circ], \ell) \text{ (Lemma 16)} \\ &= I_\top(\widetilde{r}')[\ell] \end{aligned}$$

We have ignored the initialization terms, as it is constant and by the monotonicity of least upper bounds as $step^\top$ is monotone monotonicity is preserved.

Next, consider some $p^\bullet \rightsquigarrow \ell^\circ$, and whether $\widetilde{\mathcal{FT}}(\widetilde{r}, p^\bullet \rightsquigarrow \ell^\circ)$ is true. If so, then Lemma 17 implies that $\widetilde{\mathcal{FT}}(\widetilde{r}', p^\bullet \rightsquigarrow \ell^\circ)$ must also be true, which from the fact that $\widetilde{r}[p^\circ \rightsquigarrow p^\bullet] \sqsubseteq \widetilde{r}'[p^\circ \rightsquigarrow p^\bullet]$, gives us that $I_\top(\widetilde{r})[p^\bullet \rightsquigarrow \ell^\circ] = I_\top(\widetilde{r}')[p^\bullet \rightsquigarrow \ell^\circ]$. Otherwise, $I_\top(\widetilde{r})[p^\bullet \rightsquigarrow \ell^\circ] = \bot \sqsubseteq I_\top(\widetilde{r}')[p^\bullet \rightsquigarrow \ell^\circ]$. □

We can now prove the continuity of $I_\top$.

Theorem 8. *For an increasing sequence $r_i : \widetilde{R}$, $I_\top$ is continuous.*

Proof. We first note that the initialization term of $I_\top$ at the start label s is a constant term and can be easily factored out.

It suffices to consider an arbitrary $\ell^\circ \rightsquigarrow \ell^\bullet$ and $p^\bullet \rightsquigarrow \ell^\circ$.

$$\begin{aligned}\bigsqcup_i I_\top(r_i)[\ell^\circ \rightsquigarrow \ell^\bullet] &= \bigsqcup_i step^\top(\bigsqcup_{p\in pred(\ell)} r_i[p^\bullet \rightsquigarrow \ell^\circ], \ell)\\ &= step^\top(\bigsqcup_i \bigsqcup_{p\in pred(\ell)} r_i[p^\bullet \rightsquigarrow \ell^\circ], \ell)\\ &= step^\top(\bigsqcup_{p\in pred(\ell)} (\bigsqcup_i r_i)[p^\bullet \rightsquigarrow \ell^\circ], \ell)\\ &= I_\top(\bigsqcup_i r_i)[\ell]\end{aligned}$$

The second equality follows from $\sqcup_{p\in pred(\ell)} r_i[p^\bullet \rightsquigarrow \ell^\circ]$ being and increasing sequence and Lemma 15.

Next, let us consider some $p^\bullet \rightsquigarrow \ell^\circ$. Consider the case where $\widetilde{\mathcal{FT}}(\bigsqcup_i r_i, p^\bullet \rightsquigarrow \ell^\circ)$ is false, thus $I_\top(\bigsqcup_i r_i)[p^\bullet \rightsquigarrow \ell^\circ] = \bot$. By Lemma 17 (2), we have that $\forall j.\widetilde{\mathcal{FT}}(r_j, p^\bullet \rightsquigarrow \ell^\circ)$ is also false, and thus $\bigsqcup_i I_\top(r_i)[p^\bullet \rightsquigarrow \ell^\circ] = \bigsqcup_i \bot = I_\top(\bigsqcup_i r_i)[p^\bullet \rightsquigarrow \ell^\circ]$.

Finally, consider the case where $\widetilde{\mathcal{FT}}(\bigsqcup_i r_i, p^\bullet \rightsquigarrow \ell^\circ)$ is true, whence $I_\top(\bigsqcup_i r_i,)[p^\bullet \rightsquigarrow \ell^\circ] = \bigsqcup_i r_i[p^\circ \rightsquigarrow p^\bullet]$. By Lemma 17, there exists some j such that $\forall k \geq j.\widetilde{\mathcal{FT}}(r_k, p^\bullet \rightsquigarrow \ell^\circ)$ is true. As $I_\top$ is monotone, we have that $I_\top(r_i)$ is an increasing sequence, and thus

$$\begin{aligned}\Big[\bigsqcup_i I_\top(r_i)\Big][p^\bullet \rightsquigarrow \ell^\circ] &= \Big[\bigsqcup_{k\geq j} I_\top(r_k)\Big][p^\bullet \rightsquigarrow \ell^\circ] \text{ ($I_\top(r_i)$ is increasing)}\\ &= \bigsqcup_{k\geq j} r_k[p^\circ \rightsquigarrow p^\bullet] \text{ (As $\forall k \geq j.\widetilde{\mathcal{FT}}(r_k, p^\bullet \rightsquigarrow \ell^\circ)$ must be true)}\\ &= \bigsqcup_i r_i[p^\circ \rightsquigarrow p^\bullet] \text{ (r_i is increasing)}\\ &= I_\top(\bigsqcup_i r_i)[p^\bullet \rightsquigarrow \ell^\circ]\end{aligned}$$

□

C.2 $\mathfrak{C}^\epsilon \sqsubseteq \mathfrak{M}^\epsilon$

We first need the following lemmas.

Lemma 19. *$\forall \vec{\ell} \in \mathcal{L}_A, X \sqsubseteq X' \in \overline{R}.\widehat{F} \circ \widehat{inj}(X)[\vec{\ell}] \sqsubseteq \bigsqcup_{i<\omega} U^i(\widehat{inj}(X'))[\vec{\ell}]$*

Proof.

$$\widehat{F}(\widehat{inj}(X))[\vec{\ell}] \sqsubseteq \widehat{F}(\widehat{inj}(X'))[\vec{\ell}] = U(\widehat{inj}(X')[\vec{\ell}] \sqsubseteq \bigsqcup_{i<\omega} U^i(\widehat{inj}(X'))[\vec{\ell}]$$

□

Lemma 20. *$\forall m, \ell \in \mathcal{L}_A, \delta \in Ord.\delta > 0 \Rightarrow \mathfrak{T}^\delta_m[\ell] = m[\ell]$*

Proof. By an easy transfinite induction argument, at a successor ordinal $\delta + 1$: $\mathfrak{T}^{\delta+1}_m[\ell] = T(m)(\mathfrak{T}^\delta_m)[\ell] = m[\ell]$

At a limit ordinal, we have that $\mathfrak{T}^\lambda_m[\ell] = \bigsqcup_{\alpha<\lambda} \mathfrak{T}^\alpha_m[\ell] = \bot \sqcup \bigsqcup_{\alpha<\lambda} \mathfrak{T}^{\alpha+1}_m[\ell] = \bigsqcup_{\alpha<\lambda} \mathfrak{T}^{\alpha+1}_m[\ell] = \bigsqcup_{\alpha<\lambda} m[\ell] = m[\ell]$ □

Lemma 21. *$\forall \delta \in \mu, m.\mathfrak{C}^\delta|_{\mathcal{L}_A} \sqsubseteq m|_{\mathcal{L}_A} \Rightarrow \mathfrak{C}^{\delta+1}|_{\mathcal{L}_F} \sqsubseteq \mathfrak{T}^{\delta+2}_G|_{\mathcal{L}_F}$ where $G = \widetilde{inj}(m)\|_{\mathcal{L}_A}$*

Proof. By transfinite induction on δ:

Case $\delta = 0$: It suffices to show that for some $\ell \in \mathcal{L}_F$, $\mathfrak{C}^1[\ell] = C(\bot_{\overline{R}})[\ell] = I_\top(\widetilde{inj}(\bot_{\overline{R}}))[\vec{\ell}] \sqsubseteq \mathfrak{T}^2_G[\ell] = I_\top(\mathfrak{T}^1_G)[\ell]$. This follows from the monotonicity of $I_\top$ if we show that $\widetilde{inj}(\bot_{\overline{R}}) \sqsubseteq \mathfrak{T}^1_G$. At points in $\mathcal{L}_F$ this is immediate, as $\widetilde{inj}(\bot_{\overline{R}})[\vec{\ell}' \in \mathcal{L}_F] = \bot_{\widetilde{S}} \sqsubseteq \mathfrak{T}^1_G[\vec{\ell}']$. At some point $\vec{\ell}' \in \mathcal{L}_A$, observe that $\widetilde{inj}(\bot_{\overline{R}})[\vec{\ell}'] = \widetilde{\tau}(\bot_{\widehat{S}})$, and by Lemma 20 we have that $\mathfrak{T}^1_G[\vec{\ell}'] = G[\vec{\ell}'] = \widetilde{inj}(m)\|_{\mathcal{L}_A}[\vec{\ell}'] = \widetilde{\tau}(m[\vec{\ell}'])$, whence the result holds from the monotonicity of $\widetilde{\tau}$.

Case $\delta + 1$: It suffices to show that for some $\ell \in \mathcal{L}_F$ we have $\mathfrak{C}^{\delta+2}[\ell] = I_\top(\widetilde{inj}(\mathfrak{C}^{\delta+1}))[\ell] \sqsubseteq \mathfrak{T}^{\delta+3}_m[\ell] = I_\top(\mathfrak{T}^{\delta+2}_m)[\ell]$, which holds if we can prove that $\widetilde{inj}(\mathfrak{C}^{\delta+1}) \sqsubseteq \mathfrak{T}^{\delta+2}_G$.

As the sequence $\mathfrak{C}$ is increasing, we have that $\mathfrak{C}^\delta \sqsubseteq \mathfrak{C}^{\delta+1}$, whence by transitivity we have that $\mathfrak{C}^\delta|_{\mathcal{L}_A} \sqsubseteq m|_{\mathcal{L}_A}$, thus by the inductive hypothesis, we have $\mathfrak{C}^{\delta+1}|_{\mathcal{L}_F} \sqsubseteq \mathfrak{T}^{\delta+2}_G|_{\mathcal{L}_F}$.

To establish the required inequality, we proceed by subcases on the partition of some arbitrary $\vec{\ell}'$:

Subcase $\vec{\ell}' \in \mathcal{L}_F$: $\widetilde{inj}(\mathfrak{C}^{\delta+1})[\vec{\ell}'] = \mathfrak{C}^{\delta+1}[\vec{\ell}'] \sqsubseteq \mathfrak{T}^{\delta+2}_G|_{\mathcal{L}_F}[\vec{\ell}'] = \mathfrak{T}^{\delta+2}_G[\vec{\ell}']$, where the inequality holds from the application of the inductive hypothesis.

Subcase $\vec{\ell}' \in \mathcal{L}_A$: $\widetilde{inj}(\mathfrak{C}^{\delta+1})[\vec{\ell}'] = \widetilde{\tau}(\mathfrak{C}^{\delta+1}[\vec{\ell}']) \sqsubseteq \widetilde{\tau}(m[\vec{\ell}']) = \widetilde{inj}(m)[\vec{\ell}'] = G[\vec{\ell}'] = \mathfrak{T}^{\delta+2}_G[\vec{\ell}']$ where the final equality follows from Lemma 20, and the inequality follows from the assumption $\mathfrak{C}^{\delta+1}|_{\mathcal{L}_A} \sqsubseteq m|_{\mathcal{L}_A}$ and the monotonicity of $\widetilde{\tau}$.

Case $\delta = \lambda$: If we show that $\widetilde{inj}(\mathfrak{C}^\lambda) \sqsubseteq \mathfrak{T}^\lambda_G$, then we will have, for all $\vec{\ell} \in \mathcal{L}_F$:

$$\mathfrak{C}^{\lambda+1}[\vec{\ell}] = I_\top(\widetilde{inj}(\mathfrak{C}^\lambda))[\vec{\ell}] \sqsubseteq I_\top(\mathfrak{T}^\lambda_G)[\vec{\ell}] = \mathfrak{T}^{\lambda+1}_G[\vec{\ell}] \sqsubseteq \mathfrak{T}^{\lambda+2}_G[\vec{\ell}]$$

Expanding definitions, we must therefore show that

$$\widetilde{inj}(\bigsqcup_{\beta<\lambda} \mathfrak{C}^\beta) \sqsubseteq \bigsqcup_{\beta<\lambda} \mathfrak{T}^\beta_G = \mathfrak{T}^\lambda_G$$

We show the inequality by cases:

Subcase $\vec{\ell} \in \mathcal{L}_A$:

$$\widetilde{inj}(\bigsqcup_{\beta<\lambda} \mathfrak{C}^\beta)[\vec{\ell}] = \widetilde{\tau}(\bigsqcup_{\beta<\lambda} \mathfrak{C}^\beta[\vec{\ell}]) = \widetilde{\tau}(\mathfrak{C}^\lambda[\vec{\ell}]) \sqsubseteq \widetilde{\tau}(m[\vec{\ell}]) = \widetilde{inj}(m)\|_{\mathcal{L}_A}[\vec{\ell}] = \mathfrak{T}^\lambda_G[\vec{\ell}]$$

Where the final equality again follows from Lemma 20.

Subcase $\vec{\ell} \in \mathcal{L}_F$: It suffices to show that:

$$\widetilde{inj}(\bigsqcup_{\beta<\lambda} \mathfrak{C}^\beta)[\vec{\ell}] = \bigsqcup_{\beta<\lambda} \mathfrak{C}^\beta[\vec{\ell}] = \bot \sqcup \bigsqcup_{\beta<\lambda} \mathfrak{C}^{\beta+1}[\vec{\ell}] = \bigsqcup_{\beta<\lambda} \mathfrak{C}^{\beta+1}[\vec{\ell}] \sqsubseteq \mathfrak{T}^\lambda_G[\vec{\ell}]$$

that is, $\mathfrak{T}^\lambda_G[\vec{\ell}]$ is an upper bound for every $\mathfrak{C}^{\beta+1}[\vec{\ell}]$

By the definition of least upper bounds, we have that $\forall \alpha < \lambda . \mathfrak{C}^\alpha|_{\mathcal{L}_A} \sqsubseteq \mathfrak{C}^\lambda|_{\mathcal{L}_A} \sqsubseteq m|_{\mathcal{L}_A}$, whence by the transfinite hypothesis we have

$$\begin{aligned}\forall \alpha < \lambda . \mathfrak{C}^{\alpha+1}[\vec{\ell}] = \mathfrak{C}^{\alpha+1}|_{\mathcal{L}_F}[\vec{\ell}] &\sqsubseteq \mathfrak{T}^{\alpha+2}_G|_{\mathcal{L}_F}[\vec{\ell}] = \mathfrak{T}^{\alpha+2}_G[\vec{\ell}] \\ &\sqsubseteq \bigsqcup_{\beta<\lambda} \mathfrak{T}^\beta_G[\vec{\ell}] = \mathfrak{T}^\lambda_G[\vec{\ell}]\end{aligned}$$

which shows the upper bound as required.

□

Now, to prove $\mathfrak{C}^\epsilon \sqsubseteq \mathfrak{M}^\epsilon$, we prove the following:

Lemma 22. $\forall \delta \in \mu . \mathfrak{C}^\delta \sqsubseteq \mathfrak{M}^\delta$

Proof. By transfinite induction on δ.

Case $\delta = 0$: Trivial, $\mathfrak{C}^0 = \bot = \mathfrak{M}^0$

Case $\delta + 1$: It suffices to show that for any $\vec{\ell} \in \mathcal{L}_A$, $\mathfrak{C}^{\delta+1}[\vec{\ell}] \sqsubseteq \mathfrak{M}^{\delta+1}[\vec{\ell}]$ and similarly for $\vec{\ell} \in \mathcal{L}_F$.

Subcase $\ell \in \mathcal{L}_A$: We must show that:

$$C(\mathfrak{C}^\delta)[\vec{\ell}] = \widehat{F}(\widehat{inj}(\mathfrak{C}^\delta))[\ell] \sqsubseteq \bigsqcup_{i<\omega} U^i(\widehat{inj}(\mathfrak{M}^\delta))[\vec{\ell}]$$

which follows from the induction hypothesis, the monotonicity of $\widehat{inj}$, and Lemma 19.

Subcase $\ell \in \mathcal{L}_F$: We must show that $\mathfrak{C}^{\delta+1}[\ell] \sqsubseteq [\bigsqcup_{i<\omega} T(\widetilde{inj}(\mathfrak{M}^\delta)\Vert_{\mathcal{L}_A})^i(\bot)][\ell] = \mathfrak{T}^\omega_{\mathfrak{G}}[\ell]$, where $\mathfrak{G} = \widetilde{inj}(\mathfrak{M}^\delta)\Vert_{\mathcal{L}_A}$. From the induction hypothesis, we have that $\mathfrak{C}^\delta \sqsubseteq \mathfrak{M}^\delta$, from which it follows that $\mathfrak{C}^\delta|_{\mathcal{L}_A} \sqsubseteq \mathfrak{M}^\delta|_{\mathcal{L}_A}$. Thus, by Lemma 21 we conclude that:

$$\mathfrak{C}^{\delta+1}[\ell] \sqsubseteq \mathfrak{T}^{\delta+2}_{\mathfrak{G}}[\ell] \sqsubseteq \mathfrak{T}^{\omega}_{\mathfrak{G}}[\ell] \text{ (by Lemma 10)}$$

Case $\delta = \lambda$: Follows immediately from the definition of least-upper bound, transitivity, and the fact that, $\forall \beta < \lambda. \mathfrak{C}^\beta \sqsubseteq \mathfrak{M}^\beta$.

□

Theorem 9. $\mathfrak{C}^\epsilon \sqsubseteq \mathfrak{M}^\epsilon$

Proof. Immediate from Lemma 22. □

C.3 $\mathfrak{M}^\epsilon \sqsubseteq \mathfrak{C}^\epsilon$

We first need the following lemma:

Lemma 23. $\forall i \in \mathbb{N}, X : \overline{R}. X \sqsubseteq \mathfrak{C}^\epsilon \Rightarrow T(\widetilde{inj}(X)\Vert_{\mathcal{L}_A})^i(\bot) \sqsubseteq \widetilde{inj}(\mathfrak{C}^\epsilon)$

Proof. By induction on i.

Case $i = 0$: Trivial

Case $i + 1$: We assume $T(\widetilde{inj}(X)\Vert_{\mathcal{L}_A})^i(\bot) \sqsubseteq \widetilde{inj}(\mathfrak{C}^\epsilon)$ and show that $T(\widetilde{inj}(X)\Vert_{\mathcal{L}_A})^{i+1}(\bot) \sqsubseteq \widetilde{inj}(\mathfrak{C}^\epsilon)$. At $\vec{\ell} \in \mathcal{L}_A$, by Lemma 20 we have that $T(\widetilde{inj}(X)\Vert_{\mathcal{L}_A})^{i+1}(\bot)[\vec{\ell}] = \widetilde{inj}(X)\Vert_{\mathcal{L}_A}[\vec{\ell}] = \widetilde{\tau}(X[\vec{\ell}]) \sqsubseteq \widetilde{\tau}(\mathfrak{C}^\epsilon[\vec{\ell}]) = \widetilde{inj}(\mathfrak{C}^\epsilon)[\vec{\ell}]$. It remains to show that the inequality holds at some $\vec{\ell} \in \mathcal{L}_F$. By expanding definitions we have: $T(\widetilde{inj}(X)\Vert_{\mathcal{L}_A})^{i+1}(\bot)[\vec{\ell}] = I_{\mathsf{T}}(T(\widetilde{inj}(X)\Vert_{\mathcal{L}_A})^i(\bot))[\vec{\ell}] \sqsubseteq I_{\mathsf{T}}(\widetilde{inj}(\mathfrak{C}^\epsilon))[\vec{\ell}] = C(\mathfrak{C}^\epsilon)[\vec{\ell}] = \widetilde{inj}(C(\mathfrak{C}^\epsilon))[\vec{\ell}] = \widetilde{inj}(\mathfrak{C}^\epsilon)[\vec{\ell}]$, where the inequality holds from the monotonicity of I_{T} and the induction hypothesis, and the final equality comes from the definition of $\mathfrak{C}^\epsilon$ as a fixed point of C.

□

Lemma 24. $\forall i \in \mathbb{N}, X : \overline{R}. X \sqsubseteq \mathfrak{C}^\epsilon \Rightarrow U^i(\widehat{inj}(X)) \sqsubseteq \widehat{inj}(\mathfrak{C}^\epsilon)$

Proof. By induction on i.

Case $i = 0$: Trivial, by the monotonicity of $\widehat{inj}$.

Case $i + 1$: By a similar argument to that made in Lemma 23, except on the monotonicity of $\widehat{F}$.

□

We next prove the following theorem:

Theorem 10. $\forall \delta \in \mu. \mathfrak{M}^\delta \sqsubseteq \mathfrak{C}^\epsilon$

Proof. By transfinite induction on δ.

Case $\delta = 0$: Trivial as $\mathfrak{M}^0 = \bot$.

Case $\delta + 1$: It sufficient to show that $\forall \vec{\ell}. \mathfrak{M}^{\delta+1}[\vec{\ell}] \sqsubseteq \mathfrak{C}^\epsilon[\vec{\ell}]$. We proceed by the partition of $\vec{\ell}$:

Subcase $\vec{\ell} \in \mathcal{L}_A$: $\mathfrak{M}^{\delta+1}[\vec{\ell}] = [\bigsqcup_{i<\omega} U^i(\widehat{inj}(\mathfrak{M}^\delta))][\vec{\ell}] \sqsubseteq \widehat{inj}(\mathfrak{C}^\epsilon)[\vec{\ell}] = \mathfrak{C}^\epsilon[\vec{\ell}]$, where the inequality holds from using the inductive hypothesis and Lemma 24 to show that $\widehat{inj}(\mathfrak{C}^\epsilon)[\vec{\ell}]$ is an upper bound and thus greater than the least upper bound.

Subcase $\ell \in \mathcal{L}_F$: By a similar argument to the $\mathcal{L}_A$ case, but using Lemma 23.

Case $\delta = \lambda$: : By the transfinite hypothesis, $\forall \alpha < \lambda . \mathfrak{M}^{\alpha} \sqsubseteq \mathfrak{C}^{\epsilon}$, thus: $\bigsqcup_{\alpha<\lambda} \mathfrak{M}^{\alpha} \sqsubseteq \mathfrak{C}^{\epsilon}$.

□

Corollary 10.1. $\mathfrak{M}^{\epsilon} \sqsubseteq \mathfrak{C}^{\epsilon}$

Proof. Follows immediately from Theorem 10 and the definition of $\mathfrak{M}^{\epsilon}$ as the limit of the sequence $\mathfrak{M}^{\delta \in \mu}$. □

C.4 Proof of Theorem 6

Immediate from the definitions of $lfp\, M = \mathfrak{M}^{\epsilon}$ and $lfp\, C = \mathfrak{C}^{\epsilon}$, anti-symmetry, Theorem 9, and Corollary 10.1.

C.5 Fixpoints in the Abstract Interpreter

If $\widehat{F}$ is upper-ω-continuous, then the sequence $U^i(\widehat{inj}(\mathfrak{M}^{\delta}))$ converges to a fixpoint for all δ. To show this result, it is sufficient to show that the sequence $U^i(\widehat{inj}(\mathfrak{M}^{\delta}))$ is increasing for all δ. The result will then follow from a modified version of Kleene's fixpoint theorem. In the following, $\widehat{\mathfrak{M}^{\delta}}$ denotes $\widehat{inj}(\mathfrak{M}^{\delta})$.

Lemma 25. $\forall \delta, k . U^k(\widehat{\mathfrak{M}^{\delta}}) \sqsubseteq \widehat{\mathfrak{M}^{\delta+1}}$

Proof. By cases. If $k = 0$, then the result is immediate, as $\mathfrak{M}^{\delta}$ is increasing, $\widehat{inj}$ is monotone.

For $k > 0$, we establish the result pointwise. At $\vec{\ell} \in \mathcal{L}_F$, $U^k(\widehat{\mathfrak{M}^{\delta}})[\vec{\ell}] = \widehat{\mathfrak{M}^{\delta}}[\vec{\ell}]$, whence the inequality holds by the reasoning above. At $\vec{\ell} \in \mathcal{L}_A$, we have:

$$U^k(\widehat{\mathfrak{M}^{\delta}})[\vec{\ell}] \sqsubseteq \bigsqcup_i U^i(\widehat{\mathfrak{M}^{\delta}})[\vec{\ell}] = \mathfrak{M}^{\delta+1}[\vec{\ell}] = \widehat{inj}(\mathfrak{M}^{\delta+1})[\vec{\ell}]$$

□

Next define the following monotone function $PF : \overline{R} \to \overline{R}$:

$$PF = \lambda\langle \widetilde{m}, \widehat{m}\rangle . \langle \widetilde{m}, U(\widehat{inj}(\langle \widetilde{m}, \widehat{m}\rangle))|_{\mathcal{L}_A}\rangle$$

Lemma 26. $\forall \delta . \mathfrak{M}^{\delta} \sqsubseteq PF(\mathfrak{M}^{\delta})$

Proof. By transfinite induction.

Case $\delta = 0$: Trivial.

Case $\delta + 1$: The inequality is immediate at all points in $\mathcal{L}_F$. We must therefore establish the inequality at some $\vec{\ell} \in \mathcal{L}_A$.

As $\mathfrak{M}^{\delta+1}[\vec{\ell}] = \bigsqcup_i U^i(\widehat{\mathfrak{M}^{\delta}})[\vec{\ell}]$, it suffices to show that for all i, $U^i(\widehat{\mathfrak{M}^{\delta}})[\vec{\ell}] \sqsubseteq U(\widehat{\mathfrak{M}^{\delta+1}})[\vec{\ell}] = PF(\mathfrak{M}^{\delta+1})[\vec{\ell}]$. We consider the form of i.

Subcase $i = 0$: we must show that $\widehat{\mathfrak{M}^{\delta}}[\vec{\ell}] = \mathfrak{M}^{\delta}[\vec{\ell}] \sqsubseteq U(\widehat{\mathfrak{M}^{\delta+1}})[\vec{\ell}]$. By the induction hypothesis, we have that $\mathfrak{M}^{\delta} \sqsubseteq PF(\mathfrak{M}^{\delta})$, whence we have $\mathfrak{M}^{\delta}[\vec{\ell}] \sqsubseteq PF(\mathfrak{M}^{\delta})[\vec{\ell}] = U(\widehat{inj}(\mathfrak{M}^{\delta}))[\vec{\ell}] \sqsubseteq U(\widehat{inj}(\mathfrak{M}^{\delta+1}))[\vec{\ell}]$, from the monotonicity of U and $\widehat{inj}$ and the fact that $\mathfrak{M}^{\delta}$ is inreasing.

Subcase $i > 0$: Then $i = k+1$ for some k, whence we must show $U^i(\widehat{\mathfrak{M}^{\delta}})[\vec{\ell}] = U(U^k(\widehat{\mathfrak{M}^{\delta}}))[\vec{\ell}] \sqsubseteq U(\widehat{\mathfrak{M}^{\delta+1}})[\vec{\ell}]$. By the monotonicity of U, it suffices to show that $U^k(\widehat{\mathfrak{M}^{\delta}}) \sqsubseteq \widehat{\mathfrak{M}^{\delta+1}}$, which holds from Lemma 25.

Case $\delta = \lambda$: By the induction hypothesis, $\mathfrak{M}^{\lambda}$ is defined as the least upper bound of post-fixed points of PF, whence $\mathfrak{M}^{\lambda}$ is also a post-fixed point of PF.

□

Theorem 11. *$\forall \delta \in Ord$, the sequence $U^i(\widehat{\mathfrak{M}^\delta})$ is increasing.*

Proof. We will prove the sequence increasing by induction on i for some arbitrary δ.

Case $i = 0$: The inequality is immediate at $\vec{\ell} \in \mathcal{L}_F$. At $\vec{\ell} \in \mathcal{L}_A$, we have: $\widehat{inj}(\mathfrak{M}^\delta)[\vec{\ell}] = \mathfrak{M}^\delta[\vec{\ell}] \sqsubseteq PF(\mathfrak{M}^\delta)[\vec{\ell}] = U(\widehat{inj}(\mathfrak{M}^\delta))[\vec{\ell}] = U^1(\widehat{\mathfrak{M}^\delta})[\vec{\ell}]$ by Lemma 26.

Case $i + 1$: Immediate from the monotonicity of U and the induction hypothesis.

□

D FORMALISMS, SOUNDNESS, AND TERMINATION OF WIDENING ITERATION

We first define the instrumented semantic functions and extend the iteration functions defined in Appendix C:

$$\widehat{F}_\triangledown(X) = \begin{cases} X[\vec{\ell}]\widehat{\triangledown}\widehat{F}(X)[\vec{\ell}] & \vec{\ell} \in \mathcal{W}_A \\ \widehat{F}(X)[\vec{\ell}] & o.w. \end{cases} \qquad I_\top^\triangledown(X) = \begin{cases} X[\vec{\ell}]\widetilde{\triangledown}I_\top(X)[\vec{\ell}] & \vec{\ell} \in \mathcal{W}_F \\ I_\top(X)[\vec{\ell}] & o.w. \end{cases}$$

$$T_\triangledown(m)(X) = m\|_{\mathcal{L}_A} \sqcup I_\top^\triangledown(X)\|_{\mathcal{L}_F} \qquad U_\triangledown(X) = X\|_{\mathcal{L}_F} \sqcup \widehat{F}_\triangledown(X)\|_{\mathcal{L}_A}$$

$$M_\triangledown(X) = \left\langle \bigsqcup_{i<\omega} U_\triangledown^i(\widehat{inj}(X))|_{\mathcal{L}_A},\ X|_{\mathcal{L}_F} \sqcup \bigsqcup_{i<\omega} T_\triangledown(\widetilde{inj}(X)|_{\mathcal{L}_A})^i(\bot)|_{\mathcal{L}_F} \right\rangle$$

We recall that $\widehat{\triangledown}$ is a widening operator on abstract states provided by the analysis, and $\widetilde{\triangledown}$ is the widening operator on mostly-concrete states, defined as

$$\langle m_f, m_a\rangle\widetilde{\triangledown}\langle m_f', m_a'\rangle = \left\langle \left(\lambda x : X_f. \begin{cases} m_f & m_f'[x] \sqsubseteq m_f[x] \\ \top_F & o.w. \end{cases}\right), \left(\lambda x : X_a.m_a[x] \triangledown_{\widehat{A}} m_a'[x]\right)\right\rangle$$

where $\triangledown_{\widehat{A}}$ is a widening operator on values from $\widehat{A}$ provided by the analysis. We assume that $\bot \triangledown_{\widehat{A}} x = x$ and similarly that $\bot \widehat{\triangledown} x = x$.

These functions mirror the definitions in Appendix C, with two key differences. First, $\widehat{F}_\triangledown$ and $I_\top^\triangledown$ apply the widening operator at the widening points, which ensures termination (as we prove below). In addition, $M_\triangledown$ extends the mostly-concrete iteration term to join the results of mostly-concrete iteration with the input states. This joining ensures that the iteration is increasing, which is a key condition of our termination proofs.

D.1 Soundness

We first prove the soundness of the widening version of subfixpoint iteration. Define a sequence $\mathfrak{M}_\triangledown^{\delta \in \mu}$ as:

$$\mathfrak{M}_\triangledown^0 = \bot \qquad \mathfrak{M}_\triangledown^{\delta+1} = M_\triangledown(\mathfrak{M}_\triangledown^\delta) \qquad \mathfrak{M}_\triangledown^\lambda = \bigsqcup_{\beta<\lambda} \mathfrak{M}_\triangledown^\beta$$

We have shown that this sequence over-approximates the sequence $\mathfrak{M}^\delta$ as follows:

Theorem 12. *$\forall \delta.\mathfrak{M}^\delta \sqsubseteq \mathfrak{M}_\triangledown^\delta$*

Lemma 27. *$\forall i \in \mathbb{N}, m, m' : \widetilde{R}.m \sqsubseteq m' \Rightarrow T(m)^i(\bot) \sqsubseteq T_\triangledown(m')^i(\bot)$*

Proof. By induction on i.

Case $i = 0$: Trivial

Case $i + 1$: For the values at $\vec{\ell} \in \mathcal{L}_A$, the inequality is immediate by the assumption on m and m'. It remains to show that the inequality holds at some arbitrary $\vec{\ell} \in \mathcal{L}_F$. For $\vec{\ell} \in \mathcal{W}_F$, we have by the induction hypothesis $T(m)^i(\bot) \sqsubseteq T_\triangledown(m')^i(\bot)$ whence from the monotonicity of $I_\top$ we therefore have $I_\top(T(m)^i(\bot)) \sqsubseteq I_\top(T_\triangledown(m')^i(\bot))$, and thus: $T(m)^{i+1}(\bot)[\vec{\ell}] = I_\top(T(m)^i(\bot))[\vec{\ell}] \sqsubseteq I_\top(T_\triangledown(m')^i(\bot))[\vec{\ell}] \sqsubseteq T_\triangledown(m')^i(\bot)[\vec{\ell}]\widetilde{\triangledown}I_\top(T_\triangledown(m')^i(\bot))[\vec{\ell}] = T_\triangledown(m')^{i+1}(\bot)$, where the inequality holds as $\widetilde{\triangledown}$ is an upper bound operator. For $\vec{\ell} \notin \mathcal{W}_F$, then the result holds from the induction hypothesis and the monotonicity of $I_\top$.

□

Lemma 28. $\forall i \in \mathbb{N}, m, m' : \widehat{R}.m \sqsubseteq m' \Rightarrow U^i(m) \sqsubseteq U^i_\triangledown(m')$

Proof. By symmetric reasoning to that in Lemma 27, except using the monotonicity of $\widehat{F}$ and that $\widehat{\triangledown}$ is an upper bound operator. □

Proof of Theorem 12. By transfinite induction. In the inductive step, the inductive hypothesis, monotonicity of $\widehat{inj}$ and $\widetilde{inj}$, and Lemmas 27 and 28 give that the subfixpoint terms of $\mathfrak{M}^{\delta+1}$ are smaller than those in $\mathfrak{M}^{\delta+1}_\triangledown$ giving the desired result. The limit case follows from the definition of least upper bounds. □

As a direct corollary of this theorem, we have: $\mathit{lfp}\,C = \mathit{lfp}\,M = \mathfrak{M}^\epsilon \sqsubseteq \mathfrak{M}^\epsilon_\triangledown$. If we show that if the sequence $\mathfrak{M}^\delta_\triangledown$ converges in a finite number of k steps to a fixed-point of $M_\triangledown$, we will then have: $\mathit{lfp}\,C = \mathit{lfp}\,M = \mathfrak{M}^\epsilon \sqsubseteq \mathfrak{M}^\epsilon_\triangledown = \mathfrak{M}^k_\triangledown$, i.e., $\alpha_F(\mathit{lfp}\,F) \sqsubseteq \mathfrak{M}^k_\triangledown$.

D.2 Termination

We first show that $\forall i \in \mathbb{N}$, the sequence $\mathfrak{M}^i_\triangledown$ is increasing.

Lemma 29. $\forall j.\mathfrak{M}^j_\triangledown \sqsubseteq \mathfrak{M}^{j+1}_\triangledown$

Proof. Consider an arbitrary j. We establish the inequality pointwise:

Case $\vec{\ell} \in \mathcal{L}_F$: $\mathfrak{M}^j_\triangledown[\vec{\ell}] = \mathfrak{M}^j_\triangledown|_{\mathcal{L}_F}[\vec{\ell}] \sqsubseteq \mathfrak{M}^j_\triangledown|_{\mathcal{L}_F}[\vec{\ell}] \sqcup \bigsqcup_{i<\omega} T_\triangledown(\widetilde{inj}(\mathfrak{M}^j_\triangledown)\Vert_{\mathcal{L}_A})^i(\bot)|_{\mathcal{L}_F}[\vec{\ell}] = \mathfrak{M}^{j+1}_\triangledown[\vec{\ell}]$
Case $\vec{\ell} \in \mathcal{L}_A$: $\mathfrak{M}^j_\triangledown[\vec{\ell}] = \widehat{inj}(\mathfrak{M}^j_\triangledown)[\vec{\ell}] = U^0_\triangledown(\widehat{inj}(\mathfrak{M}^j_\triangledown))[\vec{\ell}] \sqsubseteq \bigsqcup_{i<\omega} U^i_\triangledown(\widehat{inj}(\mathfrak{M}^j_\triangledown))[\vec{\ell}] = \mathfrak{M}^{j+1}_\triangledown[\vec{\ell}]$

□

We next show that every subfixpoint iteration in the abstract interpreter terminates. Define the following family of sequences indexed by n:

$$\mathfrak{P}_{(n,j)} = U^j_\triangledown(\widehat{inj}(\mathfrak{M}^n_\triangledown))$$

We need the following lemma.

Lemma 30.

(1) $\forall n.U^n_\triangledown(\widehat{inj}(\mathfrak{M}^n_\triangledown)) \sqsubseteq \widehat{inj}(\mathfrak{M}^{n+1}_\triangledown)$
(2) The sequence $U^j_\triangledown(\widehat{inj}(\mathfrak{M}^0_\triangledown = \bot_{\widehat{R}}))$ is increasing.
(3) For all n and $\vec{\ell} \in \mathcal{L}_A$, if all sequences $\mathfrak{P}_{(m<n,j)}$ are increasing and converge in a finite number of steps, then $\widehat{inj}(\mathfrak{M}^n_\triangledown)[\vec{\ell}] = \bot_{\widehat{S}}$ or $\exists k < n, j > 0.\widehat{inj}(\mathfrak{M}^n_\triangledown)[\vec{\ell}] = U^j_\triangledown(\widehat{inj}(\mathfrak{M}^k_\triangledown))[\vec{\ell}]$.
(4) For all n, if all sequences $\mathfrak{P}_{(m\leq n,j)}$ are increasing and converge in a finite number of steps, then the sequence $U^j_\triangledown(\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown))$ is increasing.

Proof.

(1) At $\vec{\ell} \in \mathcal{L}_F$, we have that $U^n_\triangledown(\widehat{inj}(\mathfrak{M}^n_\triangledown))[\vec{\ell}] = \widehat{inj}(\mathfrak{M}^n_\triangledown)[\vec{\ell}] \sqsubseteq \widehat{inj}(\mathfrak{M}^{n+1}_\triangledown)[\vec{\ell}]$ from Lemma 29. At $\vec{\ell} \in \mathcal{L}_A$, we have: $U^n_\triangledown(\widehat{inj}(\mathfrak{M}^n_\triangledown))[\vec{\ell}] \sqsubseteq \bigsqcup_{i<\omega} U^i_\triangledown(\widehat{inj}(\mathfrak{M}^n_\triangledown))[\vec{\ell}] = \mathfrak{M}^{n+1}_\triangledown[\vec{\ell}] = \widehat{inj}(\mathfrak{M}^{n+1}_\triangledown)[\vec{\ell}]$.

(2) By straightforward induction on j, using the fact that $\widehat{F}$ is monotone and that $\widehat{\triangledown}$ is an upper bound operator. The base case holds because at $\vec{\ell} \in \mathcal{L}_A$, $\widehat{inj}(\bot_{\widehat{R}})[\vec{\ell}] = \bot_{\widehat{S}}$ and for $\vec{\ell} \in \mathcal{L}_F$ $\forall i. U^i_\triangledown(\widehat{inj}(\bot_{\widehat{R}}))[\vec{\ell}] = \widehat{\tau}(\bot_{\widehat{S}})$.

(3) By induction on n.

Case $n = 0$: Then $\widehat{inj}(\mathfrak{M}^0_\triangledown)[\vec{\ell}] = \bot_{\widehat{R}}[\vec{\ell}] = \bot_{\widehat{S}}$, trivially giving the desired result.

Case $n + 1$: Then $\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown)[\vec{\ell}] = \bigsqcup_{i<\omega} U^i_\triangledown(\widehat{inj}(\mathfrak{M}^n_\triangledown))[\vec{\ell}]$. By hypothesis, the sequence $U^i_\triangledown(\widehat{inj}(\mathfrak{M}^n_\triangledown))$ is increasing and converges in a finite number of steps, whence $\exists p. \bigsqcup_{i<\omega} U^i_\triangledown(\widehat{inj}(\mathfrak{M}^n_\triangledown))[\vec{\ell}] = U^p_\triangledown(\widehat{inj}(\mathfrak{M}^n_\triangledown))[\vec{\ell}]$. If $p = 0$, then the inductive hypothesis gives us the desired result (it is immediate that if the sequences $\mathfrak{P}_{(m<n+1,j)}$ are terminating increasing sequences, then the sequences $\mathfrak{P}_{(m<n,j)}$ must as well). Otherwise, we take $k = n, j = p$ completing the proof.

(4) By induction on j.

Case $j = 0$: We must show that $\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown) \sqsubseteq U_\triangledown(\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown))$. By cases:

Subcase $\vec{\ell} \in \mathcal{W}_A$: $U_\triangledown(\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown))[\vec{\ell}] = \widehat{F}_\triangledown(\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown))[\vec{\ell}] = \widehat{inj}(\mathfrak{M}^{n+1}_\triangledown)[\vec{\ell}]\widehat{\triangledown}\widehat{F}(\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown))[\vec{\ell}]$, and we have the required inequality from $\widehat{\triangledown}$ being an upper bound operator.

Subcase $\vec{\ell} \in \mathcal{L}_A \wedge \vec{\ell} \notin \mathcal{W}_A$: We must show that

$$\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown)[\vec{\ell}] \sqsubseteq U_\triangledown(\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown))[\vec{\ell}] = \widehat{F}(\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown))[\vec{\ell}]$$

By hypothesis, the sequences $\mathfrak{P}_{(m\le n,j)}$ are increasing. As $m \le n$ implies $m < n+1$, we can satisfy the hypothesis of part (3) for $\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown)$. Suppose then that $\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown)[\vec{\ell}] = \bot$. Then we trivially have $\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown)[\vec{\ell}] = \bot \sqsubseteq \widehat{F}(\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown))[\vec{\ell}]$. Next, suppose that there exists some $k < n+1, j > 0$ such that $\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown)[\vec{\ell}] = U^j_\triangledown(\widehat{inj}(\mathfrak{M}^k_\triangledown))[\vec{\ell}]$. Expanding the definition of $U_\triangledown$, we must then show that $U^j_\triangledown(\widehat{inj}(\mathfrak{M}^k_\triangledown))[\vec{\ell}] = \widehat{F}(U^{j-1}_\triangledown(\widehat{inj}(\mathfrak{M}^k_\triangledown)))[\vec{\ell}] \sqsubseteq \widehat{F}(\widehat{inj}(\mathfrak{M}^n_\triangledown))[\vec{\ell}]$. By part (1) and Lemma 29, we have that $U^{j-1}_\triangledown(\widehat{inj}(\mathfrak{M}^k_\triangledown)) \sqsubseteq \widehat{inj}(\mathfrak{M}^{k+1}_\triangledown) \sqsubseteq \widehat{inj}(\mathfrak{M}^{n+1}_\triangledown)$, giving the desired result via monotonicity of $\widehat{F}$.

Subcase $\vec{\ell} \in \mathcal{L}_F$: Trivial.

Case $j + 1$: Immediate at $\vec{\ell} \in \mathcal{W}_A$ from the definition of $\widehat{\triangledown}$ as an upper bounds operator, trivial at $\vec{\ell} \in \mathcal{L}_F$, and from the induction hypothesis and the monotonicity of $\widehat{F}$ for $\vec{\ell} \in \mathcal{L}_A \wedge \vec{\ell} \notin \mathcal{W}_A$.

□

We are now ready to prove that, for any n, $\mathfrak{P}_{(n,j)}$ terminates in finite steps, i.e., the abstract iteration terminates.

THEOREM 13. *$\forall n. \mathfrak{P}_{(n,j)}$ is an increasing sequence that converges in a finite number of steps.*

PROOF. By strong induction on n.

Case $n = 0$: Lemma 30 part (2) gives us that the sequence is increasing. To show that the sequence converges in a finite number of steps, it suffices to show that all labels in the widening set converge in a finite number of steps, whence the overall termination will follow from the definiton of widening sets. Consider an arbitrary $\vec{\ell} \in \mathcal{W}_A$, and the values computed at each element of the sequence $\mathfrak{P}_{(0,j)}$.

$$\mathfrak{P}_{(0,0)}[\vec{\ell}] = \widehat{inj}(\mathfrak{M}^0_\triangledown)[\vec{\ell}] = \widehat{inj}(\bot_{\widehat{R}})[\vec{\ell}] = \bot_{\widehat{S}}$$

$$\begin{aligned}\mathfrak{P}_{(0,i+1)}[\vec{\ell}] &= U_\triangledown(U^i_\triangledown(\widehat{inj}(\bot_{\widehat{R}})))[\vec{\ell}] = \widehat{F}_\triangledown(U^i_\triangledown(\widehat{inj}(\bot_{\widehat{R}})))[\vec{\ell}]\\ &= U^i_\triangledown(\widehat{inj}(\bot_{\widehat{R}}))[\vec{\ell}]\widehat{\triangledown}\widehat{F}(U^i_\triangledown(\widehat{inj}(\bot_{\widehat{R}})))[\vec{\ell}]\\ &= \mathfrak{P}_{(0,i)}[\vec{\ell}]\widehat{\triangledown}\widehat{F}(U^i_\triangledown(\widehat{inj}(\bot_{\widehat{R}})))[\vec{\ell}]\end{aligned}$$

By the definition of widening operators, this sequence will converge in a finite number of steps (that the sequence $\bot_{\widehat{S}}, \widehat{F}(\widehat{inj}(\bot_{\overline{R}})), \ldots, \widehat{F}(U^n_{\triangledown}(\widehat{inj}(\bot_{\overline{R}})))$ is increasing follows from Lemma 30 part (2) and the monotonicity of $\widehat{F}$). Thus, all widening points will converge in a finite number of steps, implying that $\mathfrak{P}_{(0,j)}$ converges.

Case $n+1$**:** We assume that all sequences $\mathfrak{P}_{(m<n+1,j)}$ are increasing and converge in a finite number of steps. From Lemma 30 part (4), $\mathfrak{P}_{(n+1,j)}$ is increasing. To show that $\mathfrak{P}_{(n+1,j)}$ converges in a finite number of steps, we will again show that any arbitrary widening point $\vec{\ell}$ converges in a finite number of steps. Define a sequence finitization operator $\lfloor \mathfrak{S} \rfloor = \langle s_0, \ldots, s_{k-1} \rangle$ where $s_i = \mathfrak{S}_{i+1}$ and k is defined as the smallest natural number such that:

$$k \geq 1 \qquad\qquad \forall m \geq k. \mathfrak{S}_m = \mathfrak{S}_k$$

In other words, the finitization operator extracts a sequence of at least length one, starting from the *second* element of the sequence, up to the first element after which the sequence repeats. This operation is undefined on sequences that do not converge in finite steps.
Define now the following sequence:

$$\mathfrak{L} = \bowtie_{m<n+1} \lfloor \mathfrak{P}_{(m,j)}[\vec{\ell}] \rfloor \bowtie \mathfrak{P}_{(n+1,j\geq 1)}[\vec{\ell}]$$

where $\bowtie$ denotes sequence concatenation. The $\lfloor \mathfrak{P}_{(m,j)}[\vec{\ell}] \rfloor$ operation is well defined from the induction hypothesis that all such sequences converge in finite steps. As a result, $\mathfrak{L}$ has a finite prefix of values before reaching elements drawn from $\mathfrak{P}_{(n+1,j)}[\vec{\ell}]$. If we show that the sequence $\mathfrak{L}$ converges in a finite number of steps, this will in turn imply that $\mathfrak{P}_{(n+1,j)}[\vec{\ell}]$ also converges in a finite number of steps. We now show that $\mathfrak{L}_{i>0} = \mathfrak{L}_{i-1} \widehat{\triangledown} \mathfrak{W}_i$ and $\mathfrak{L}_0 = \mathfrak{W}_0$, where $\mathfrak{W}_i$ is an increasing sequence. From the definition of widening operators, this sequence will converge and thus, as argued above, will prove convergence of the sequence $\mathfrak{P}_{(n+1,j)}[\vec{\ell}]$.
From the definition of $\mathfrak{L}$ it is immediate that:

$$\forall i. \mathfrak{L}_i \equiv \mathfrak{P}_{(m,k)}[\vec{\ell}] = U^k_{\triangledown}(\widehat{inj}(\mathfrak{M}^m_{\triangledown}))[\vec{\ell}] \text{ (for some } m \leq n+1 \text{ and } k \geq 1) \tag{22}$$

$$\forall i, k > 0, m \leq n+1. \mathfrak{L}_{i+1} \equiv U^{k+1}_{\triangledown}(\widehat{inj}(\mathfrak{M}^m_{\triangledown}))[\vec{\ell}] \Rightarrow \mathfrak{L}_i \equiv U^k_{\triangledown}(\widehat{inj}(\mathfrak{M}^m_{\triangledown}))[\vec{\ell}] \tag{23}$$

We will now show that for all $i > 0$, $\mathfrak{L}_{i>0}$ is defined as $\mathfrak{P}_{(m,k)}[\vec{\ell}]$ for some $m \leq n+1$ and $k \geq 1$, and further that $\mathfrak{L}_i = \mathfrak{L}_{i-1} \widehat{\triangledown} \widehat{F}(\mathfrak{P}_{(m,k-1)})[\vec{\ell}]$.
Consider an arbitrary $i > 0$. Then $\mathfrak{L}_i = U^k_{\triangledown}(\widehat{inj}(\mathfrak{M}^m_{\triangledown}))[\vec{\ell}], m \leq n+1, k > 0$ by Eq. (22). Suppose $k \neq 1$. Then $k = j+1$, and

$$U^{j+1}_{\triangledown}(\widehat{inj}(\mathfrak{M}^m_{\triangledown}))[\vec{\ell}] = U^{j}_{\triangledown}(\widehat{inj}(\mathfrak{M}^m_{\triangledown}))[\vec{\ell}] \triangledown \widehat{F}(U^{j}_{\triangledown}(\widehat{inj}(\mathfrak{M}^m_{\triangledown}))[\vec{\ell}] = \mathfrak{L}_{i-1} \widehat{\triangledown} \widehat{F}(\mathfrak{P}_{(m,k-1)})[\vec{\ell}]$$

where the final equality holds from the definition of $\mathfrak{P}$ and (23) above.
Next, suppose that $k = 1$. Then

$$U_{\triangledown}(\widehat{inj}(\mathfrak{M}^m_{\triangledown}))[\vec{\ell}] = \widehat{inj}(\mathfrak{M}^m_{\triangledown})[\vec{\ell}] \triangledown \widehat{F}(\widehat{inj}(\mathfrak{M}^m_{\triangledown}))[\vec{\ell}] = \mathfrak{M}^m_{\triangledown}[\vec{\ell}] \widehat{\triangledown} \widehat{F}(\widehat{inj}(\mathfrak{M}^m_{\triangledown}))[\vec{\ell}] = \mathfrak{M}^m_{\triangledown}[\vec{\ell}] \widehat{\triangledown} \widehat{F}(\mathfrak{P}_{(m,k-1)})[\vec{\ell}]$$

We must show that $\mathfrak{M}^m_{\triangledown}[\vec{\ell}] = \mathfrak{L}_{i-1}$. First, observe that m cannot be 0 by our assumption that $i > 0$.
As $i > 0$, there must be some preceding element $\mathfrak{L}_{i-1}$ and by definition this element is the final element of $\lfloor \mathfrak{P}_{(m-1,j)} \rfloor$. (This follows from the fact that $\mathfrak{P}_{(m,1)}[\vec{\ell}]$ must be the first element of one of the sequences that are concatenated together to form $\mathfrak{L}$.)
From the definition of $\lfloor \mathfrak{S} \rfloor$, this final element is $\mathfrak{P}_{(m-1,p)}[\vec{\ell}]$, where p is the first index greater than or equal to 1 after which the sequence $\mathfrak{P}_{(m-1,j)}[\vec{\ell}]$ repeats infinitely. Further, as the sequence $\mathfrak{P}_{(m-1,j)}$ is increasing by the induction hypothesis, we have:

$$\mathfrak{M}^m_{\triangledown}[\vec{\ell}] = \bigsqcup_{i<\omega} \mathfrak{P}_{(m-1,i)}[\vec{\ell}] = \mathfrak{P}_{(m-1,p)}[\vec{\ell}] = \mathfrak{L}_{i-1}$$

Finally, by expanding definitions we have:

$$\mathfrak{L}_0 = \mathfrak{P}_{(0,1)}[\vec{\ell}] = \widehat{F}_\triangledown(\widehat{inj}(\mathfrak{M}^0_\triangledown))[\vec{\ell}] = \widehat{inj}(\bot_{\widehat{R}})[\vec{\ell}]\widehat{\triangledown}\widehat{F}(\widehat{inj}(\mathfrak{M}^0_\triangledown))[\vec{\ell}]$$
$$= \widehat{F}(U^0_\triangledown(\widehat{inj}(\mathfrak{M}^0_\triangledown)))[\vec{\ell}] = \widehat{F}(\mathfrak{P}_{(0,0)})[\vec{\ell}]$$

We therefore define the sequence $\mathfrak{W}_i$ as:

$$\mathfrak{W}_0 = \mathfrak{L}_0 = \widehat{F}(\mathfrak{P}_{(0,0)})[\vec{\ell}]$$
$$\mathfrak{W}_{i+1} = \widehat{F}(\mathfrak{P}_{(m,k-1)})[\vec{\ell}] \text{ where } \mathfrak{L}_{i+1} \equiv \mathfrak{P}_{(m,k)}$$

It remains to show that this is an increasing sequence. From the definition of $\mathfrak{L}$, if $\mathfrak{W}_i \equiv \widehat{F}(\mathfrak{P}_{(m,k-1)})[\vec{\ell}]$ for some $m \le n+1$ and k, then $\mathfrak{W}_{i+1} = \widehat{F}(\mathfrak{P}_{(m,k)})[\vec{\ell}]$ or $\mathfrak{W}_{i+1} = \widehat{F}(\mathfrak{P}_{(m+1,1)})[\vec{\ell}]$. By the monotonicity of $\widehat{F}$, it suffices to show that $\mathfrak{P}_{(m,k-1)} \sqsubseteq \mathfrak{P}_{(m,k)}$ and that $\mathfrak{P}_{(m,k-1)} \sqsubseteq \mathfrak{P}_{(m+1,1)}$. For the former, if $m < n+1$ the result holds from the induction hypothesis, otherwise if $m = n+1$ the result holds from Lemma 30 part (4). The latter is only possible if $m < n+1$. By Lemma 30 part (1), $\mathfrak{P}_{(m,k-1)} \sqsubseteq \mathfrak{P}_{(m+1,0)}$ for any m. It remains to show that $\mathfrak{P}_{(m+1,0)} \sqsubseteq \mathfrak{P}_{(m+1,1)}$. This holds by either the induction hypothesis ($m < n$) or Lemma 30 part (4) ($m = n$).

□

We next show that all iterations in the mostly-concrete interpreter converge in finite time.

Lemma 31.
(1) $\forall m : \widetilde{R}.T_\triangledown(m)^i(\bot)$ is increasing.
(2) $\forall m : \widetilde{R}.T_\triangledown(m)^i(\bot)$ converges.

Proof.

(1) By induction on i. The in the inductive step, for flow edges in $\mathcal{W}_F$ the inequality is due to the $\widetilde{\triangledown}$ being an upper bound operator, whereas for flow edges not in $\mathcal{W}_F$ the result holds from the monotonicity of $I_\top$ and the IH.

(2) As in Theorem 13, we consider an arbitrary widening point $\vec{\ell}$. Then the values computed at each step during subfixpoint iteration, $T_\triangledown(m)^i(\bot_{\widetilde{R}})[\vec{\ell}]$ form the following sequence:

$$T_\triangledown(m)^0(\bot_{\widetilde{R}})[\vec{\ell}] = \bot_{\widetilde{S}}$$
$$T_\triangledown(m)^{i+1}(\bot_{\widetilde{R}})[\vec{\ell}] = T_\triangledown(m)^i(\bot_{\widetilde{R}})[\vec{\ell}]\widetilde{\triangledown}I_\top(T_\triangledown(m)^i(\bot_{\widetilde{R}}))[\vec{\ell}]$$

The sequence $\bot_{\widetilde{S}}, I_\top(\bot_{\widetilde{R}})[\vec{\ell}], \ldots, I_\top(T_\triangledown(m)^i(\bot_{\widetilde{R}}))[\vec{\ell}]$ is increasing from the monotonicity of $I_\top$ and that $T_\triangledown(m)^i(\bot_{\widetilde{R}})$ is increasing proved in part (1) above. Thus the above sequence converges after a finite number of steps, giving us the termination result.

□

Lemma 32. *The sequence $\mathfrak{M}^i_\triangledown|_{\mathcal{L}_A}$ converges in a finite number of steps*

Proof. By Theorem 13, as each subfixpoint iteration is finite, we can construct a sequence similar to $\mathfrak{L}$ in the proof Theorem 13. Using similar reasoning, we show that this sequence stabilizes in a finite number of steps. This implies that there is some index k, such that when computing $\mathfrak{M}^m_\triangledown|_{\mathcal{L}_A}, m \ge k$, the values at all widening points in $\mathcal{W}_A$ will converge to the same values. This implies that after a k steps, the sequence $\mathfrak{M}^k_\triangledown|_{\mathcal{L}_A}$ stabilizes. □

Theorem 14. *The sequence $\mathfrak{M}^i_\triangledown$ converges in a finite number of steps.*

Proof. We first note that, as $I^\triangledown_\top$ is deterministic, if $m|_{\mathcal{L}_A} = m'|_{\mathcal{L}_A}$ then $\bigsqcup_{i<\omega} T_\triangledown(\widetilde{inj}(m)\|_{\mathcal{L}_A})^i(\bot) = \bigsqcup_{i<\omega} T_\triangledown(\widetilde{inj}(m')\|_{\mathcal{L}_A})^i(\bot)$ By Lemma 32, there is some k such that $\mathfrak{M}^k_\triangledown|_{\mathcal{L}_A} = \mathfrak{M}^{k+1}_\triangledown|_{\mathcal{L}_A} = \mathfrak{M}^{k+2}_\triangledown|_{\mathcal{L}_A}$.

As $\mathfrak{M}_{\triangledown}^{k}|_{\mathcal{L}_A} = \mathfrak{M}_{\triangledown}^{k+1}|_{\mathcal{L}_A}$ we have that

$$\bigsqcup_{i<\omega} T_{\triangledown}(\widetilde{inj}(\mathfrak{M}_{\triangledown}^{k})\Vert_{\mathcal{L}_A})^{i}(\bot) = \bigsqcup_{i<\omega} T_{\triangledown}(\widetilde{inj}(\mathfrak{M}_{\triangledown}^{k+1})\Vert_{\mathcal{L}_A})^{i}(\bot)$$

Consider next some arbitrary $\vec{\ell} \in \mathcal{L}_F$. Let $\mathcal{U}_{k+1} = \bigsqcup_{i<\omega} T_{\triangledown}(\widetilde{inj}(\mathfrak{M}_{\triangledown}^{k})\Vert_{\mathcal{L}_A})^{i}(\bot)[\vec{\ell}]$ and $\mathcal{U}_{k+2} = \bigsqcup_{i<\omega} T_{\triangledown}(\widetilde{inj}(\mathfrak{M}_{\triangledown}^{k+1})\Vert_{\mathcal{L}_A})^{i}(\bot)[\vec{\ell}]$. By the above equality, we have that $\mathcal{U}_{k+1} = \mathcal{U}_{k+2}$, and thus:

$$\mathfrak{M}_{\triangledown}^{k+1}[\vec{\ell}] = \mathfrak{M}_{\triangledown}^{k}[\vec{\ell}] \sqcup \mathcal{U}_{k+1} = \mathfrak{M}_{\triangledown}^{k}[\vec{\ell}] \sqcup \mathcal{U}_{k+1} \sqcup \mathcal{U}_{k+1} =$$
$$(\mathfrak{M}_{\triangledown}^{k}[\vec{\ell}] \sqcup \mathcal{U}_{k+1}) \sqcup \mathcal{U}_{k+2} = \mathfrak{M}_{\triangledown}^{k+1}[\vec{\ell}] \sqcup \mathcal{U}_{k+2} = \mathfrak{M}_{\triangledown}^{k+2}[\vec{\ell}]$$

We therefore have $\mathfrak{M}_{\triangledown}^{k+1}|_{\mathcal{L}_A} = \mathfrak{M}_{\triangledown}^{k+2}|_{\mathcal{L}_A}$ and $\mathfrak{M}_{\triangledown}^{k+1}|_{\mathcal{L}_F} = \mathfrak{M}_{\triangledown}^{k+2}|_{\mathcal{L}_F}$, whence $\mathfrak{M}_{\triangledown}^{k+1} = \mathfrak{M}_{\triangledown}^{k+2} = M_{\triangledown}(\mathfrak{M}_{\triangledown}^{k+1})$. Thus, in $k+1$ steps, the sequence reaches a fixpoint of $M_{\triangledown}$, and thus the entire sequence stabilizes in finite time. □