**CONCERTO: A Framework for Combined Concrete and Abstract Interpretation**

**ANONYMOUS AUTHOR(S)**

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 implemented an initial prototype of **CONCERTO** for a subset of Java, and found that our combined interpretation significantly improves analysis precision and performance.

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 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
// FRAMEWORK
AppContext init(String configFile) {
    XMLDocument config = XML.parseFile(configFile);
    AppContext ctxt = ...
    for (Node n : config.getNode("delegates"))
        ctxt.delegates.put(n.get("name"), n.get("class"));
    ctxt.entryPoint = config.getNodeString("entryPoint");
    return ctxt;
}

Object delegate(AppContext ctxt, String delegateName) {
    String delegateClass = ctxt.delegates.get(delegateName);
    Object d = Class.forName(delegateClass).newInstance();
    return d.getClass().getMethod("handle").invoke(d);
}

void main() {
    AppContext ctxt = init("conf.xml");
    Class.forName(ctxt.entryPoint).getMethod("start").invoke(ctxt);
}

// APPLICATION
void start(AppContext ctxt) {
    while(true) {
        String request = Network.accept();
        if(request == "ping") {
            resp = delegate(ctxt, "ping");
        } else {
            resp = delegate(ctxt, "pong");
        }
        Network.send(resp);
    }
}

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.

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...
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.

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 expect provably improved 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. This program will also serve as our running example as we formalize our approach in the remainder of this paper.

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 \( \text{m} \) from application-specific identifiers to procedure names. The application start point, \( \text{s} \), is then called with \( \text{m} \) as its argument. The only other framework code is \( \text{dispatch} \), 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.

The program, framework, and language are significantly simpler than the full Java language and the example shown in Fig. 1, but they 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 example 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 the vast majority of 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. 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.) 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: \([m \mapsto ["b" \mapsto "f", "a" \mapsto "i"]]\). 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 \( s \)'s while loop, the signedness analysis analyzes the call to \( g \), which itself calls the framework's dispatch procedure in the abstract state \([m \mapsto ["b" \mapsto "f", "a" \mapsto "i"], p \mapsto \{+\}]\). 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 to provide 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 \(["b" \mapsto "f", "a" \mapsto "i"]\), 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 \( \text{arg} \) 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 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 form 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 provide provably improved precision.

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 goto, variable assignments, and constants and primitive operations 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 A (for application) and F (for framework) and, with a slight abuse of notation, will use A (respectively F) as a metavariable to range over the types in A (respectively F). Every variable is given a type drawn from one of these families. bcF and fop range over base constants and primitive operations respectively for types in F, and bcA and aop do the same for A. All operations in fop have type (F × . . . × F) → F, and similarly for aop and A. The <=> nonterminal ranges over comparison operators.
In our language, \( \ell_f \) label framework statements, and similarly for \( \ell_a \) and application statements. The label of a statement determines what operations may be performed by that statement: \( \textit{fops} \) and \( \textit{bcf} \) may only appear in code labeled with \( \ell_f \), and similarly for \( \textit{aop} \), \( \textit{bc} \) and \( \ell_a \). Further, we require that comparisons in statements labeled with \( \ell_a \) can only compare values of types in \( \mathcal{A} \), and similarly for \( \ell_f \) and \( \mathcal{F} \). However, statements labeled \( \ell_f \) may move values between variables with type \( \mathcal{A} \), and vice versa for statements labeled \( \ell_a \). As such, framework code must treat values of type \( \mathcal{A} \) (“application values”) opaquely, threading them through to statements labeled \( \ell_f \). Similarly, statements labeled \( \ell_a \) treat \( \mathcal{F} \) 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 \textit{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 \( \textit{aop} \) has a denotation \( [\textit{aop}]: (V_a \times \ldots V_a) \rightarrow V_a \); we assume that \( \textit{aops} \) are deterministic. In contrast, some \( \textit{fop} \) operations may produce values that depend on the value of some environment model \( E \), which models nondeterminism due to file contents, network requests, etc. Formally, each \( \textit{fop} \) has a denotation \( [\textit{fop}]: \mathcal{F} \times V_f \times \ldots V_f \rightarrow \mathcal{F}(V_f \times \mathcal{E}) \), where \( \langle v', E' \rangle \in [\textit{fop}](E, v_1, \ldots, v_n) \) means executing \( \textit{fop} \) 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 \( \llbracket \cdot \rrbracket \) denotes into a binary relation over values of the appropriate type, \( \llbracket <\rrbracket \) is the negation of \( \llbracket \cdot \rrbracket \), and \( \llbracket \textit{bcf} \rrbracket \) and \( \llbracket \textit{bc} \rrbracket \) produce values in \( V_f \) and \( V_a \) respectively corresponding to the interpretation of those constants.

Although the result of \( \textit{fops} \) depend on the current environment \( E \), in some cases we may have \textit{a priori} information such that an apparently nondeterministic \( \textit{fop} \), i.e., reading a file, is \textit{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 \textit{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 \textit{pred} and \textit{succ} are defined with the obvious definitions and there is a map \textit{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 (\textit{if/while}) can be encoded using the \texttt{goto} representation defined in Fig. 4, and we defer discussion of procedures to Section 5.

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

\[
\begin{align*}
\text{open} &: \text{Str} \rightarrow \text{File} \\
\text{read} &: \text{File} \rightarrow \text{Str} \\
\text{set} &: M \times \text{Str} \times \text{Str} \rightarrow \text{Str} \\
\text{get} &: M \times \text{Str} \rightarrow \text{Str}
\end{align*}
\]

We define \( \textit{bcf} \) as the set of literal string constants and \( \text{empty} : M \) which is an empty map. Finally, we take \( V_f \equiv \text{STR} \cup \text{FileContents} \cup (\text{STR} \rightarrow \text{STR}) \), where \( \text{STR} \) is the set of string values and \( \text{FileContents} \) is a finite stream of \( \text{STR} \) values. Given the syntactic constraints on where \( \textit{aops} \) and \( \textit{fops} \) 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.
\[ F(r)[\ell^0 \rightsquigarrow \ell^\bullet] = \bigcup_{\text{pred}(\ell)} \text{step}^F((in, \ell)) \cup \left\{ \bigcup_{e \in \ell^0} \text{step}^F((is, e), \ell) \right\}, \quad \ell = s \quad \text{o.w.} \]

\[ F(r)[p^* \rightsquigarrow \ell^\bullet] = \begin{cases} \{(s, E), (s, E) \in r[p^* \rightsquigarrow p^*] \land [s[x][\ell]] s[y] \} & \text{prog}[p] = \text{if } x \leftrightarrow y \text{ goto } \ell \\ \{(p^* \rightsquigarrow p^*) \} & \text{else} \end{cases} \]

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

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

Finally, the domain of environment models \(E\) is a map from strings to file contents, i.e., \(STR \rightarrow FileContents\). We encode the information about the configuration file by requiring that: \(E = \{e \mid e \in STR \rightarrow FileContents \land e["config"] = \langle "b", "f", "a", "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: \(S = E \times (X \rightarrow V)\) where \(V = V_a \cup V_r \cup \{\bot_V\}\). \(X\) is the domain of variables appearing in a program, \(E\) 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., for any concrete state \(s\) arising during execution: \(\text{type}(x) \in \mathcal{F} \Rightarrow s[x] \in V_f\) and \(\text{type}(x) \in \mathcal{A} \Rightarrow s[x] \in V_a\) and that all variables must be defined before use, i.e., a program never observes \(\bot_V\).

Next, define a domain of flow edges \(L\) as: \(L = \{\ell^0 \rightsquigarrow \ell^\bullet\} \cup \{p^* \rightsquigarrow \ell^0 \mid p \in \text{pred}(\ell)\}\), where \(\ell^0\) and \(\ell^\bullet\) respectively denote the “entrance to” and “exit from” \(\ell\). Elements \(\ell^0 \rightsquigarrow \ell^\bullet\) correspond to the flow of program control through the statement labelled \(\ell\), whereas an element \(p^* \rightsquigarrow \ell^0\) corresponds to the flow from some predecessor \(p\) to \(\ell\). When convenient, we will abbreviate \(\ell^0 \rightsquigarrow \ell^\bullet\) as simply \(\ell\). We will use \(\hat{\ell}\) to represent an arbitrary element of \(L\).

Our domain of concrete properties is \(R = L \rightarrow \varphi(S)\), 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, \(F : R \rightarrow R\), the least fixed-point of which is a complete set of reaching states for a program. That is, for every \(\tilde{\ell} \in L\), if \(\langle s, E' \rangle \in (\text{ifp } F)[\tilde{\ell}]\) then there is some execution that flows through edge \(\tilde{\ell}\) 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 \(F(r)[\ell^0 \rightsquigarrow \ell^\bullet]\): \(\iota_E\) is a set of possible initial environments, and \(\iota_S\) 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 \(\hat{S}\) with ordering and least-upper bound operator \(\sqsubseteq_{\hat{S}}\) and \(\sqcup_{\hat{S}}\) respectively. (We will use \(\sqsubseteq_D\) to indicate a partial order on a domain \(D\).) The domain of abstract properties, \(\hat{R} = L \rightarrow \hat{S}\), also forms a complete lattice defined by the pointwise extension of \(\sqsubseteq_{\hat{S}}\) and \(\sqcup_{\hat{S}}\). We further assume that the domain \(\hat{R}\) forms a Galois connection with the domain of concrete properties \(R\), defined by the abstraction and concretization functions \(\alpha_A\) and \(\gamma_A\). We use the standard notation \(R \xrightarrow{\gamma_A} \hat{R} \xleftarrow{\alpha_A}\).
to denote this connection. The abstract semantics of the abstract interpretation are given by a
monotone abstract semantic function \( \widehat{\ell} : \widehat{\mathcal{R}} \rightarrow \widehat{\mathcal{R}} \). We assume that this function is a sound abstraction
of \( \ell \) according to the above Galois connection, i.e., \( \alpha_A \circ \ell \subseteq \widehat{\mathcal{R}} \) \( \circ \alpha_A \), whence by [Cousot and
Cousot 1979b] we have that \( \alpha_A(\text{lfp} \ F) \subseteq \text{lfp} \ \widehat{\ell} \). In other words, the reaching states computed by the
least-fixed point of \( \ell \) are soundly over-approximated according to \( \alpha_A \) by the least-fixed point of \( \widehat{\ell} \).

Thus, an abstract interpretation is defined by the 8-tuple \( \langle \widehat{\mathcal{R}} ; \widehat{\mathcal{S}} ; \widehat{\ell} ; \alpha_A ; \langle Y_A \rangle , \subseteq_{\widehat{\mathcal{S}}} \subseteq_{\widehat{\mathcal{R}}} \rangle \) where:

\[
\widehat{\ell} : \widehat{\mathcal{R}} = \mathcal{L} \rightarrow \widehat{\mathcal{S}} \quad \widehat{\mathcal{R}} \xrightarrow{Y_A} \widehat{\ell} \quad \alpha_A \circ \widehat{\ell} \subseteq \text{lfp} \ \widehat{\ell} \circ \alpha_A
\]

Example 3.2 (Signedness Analysis). The abstract state for the intra-procedural signedness analysis
discussed in Section 2 is \( \widehat{\mathcal{S}} = \mathcal{X}_f \rightarrow \psi(\mathcal{V}_f)^\top \times \mathcal{X}_a \rightarrow \psi((-0,+)) \), and \( \alpha_A \) is defined as:

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

In the above definitions, \( \mathcal{X}_f \) is the set of program variables with type in \( \mathcal{F} \), \( \mathcal{X}_a \) are those with
types in \( \mathcal{A} \), and \( \text{sign} \) returns one of \(+\), \(-\), or \(0\) depending on the sign of its argument. We omit
the definition of \( Y_A \) as it can be derived from the definition of \( \alpha_A \) [Cousot and Cousot 1979b].

The domain \( \psi(\mathcal{V}_f)^\top \) 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 \( \subseteq_{\widehat{\mathcal{S}}} \) and \( \subseteq_{\widehat{\mathcal{R}}} \) as they are standard.

The abstract semantic function \( \widehat{\ell} \) 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 \([+],[\sim] \), etc. however \([+] \) is
defined such that \([0,+],[+] = [+] \), 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 executing the concrete semantic function \( \ell \) at
framework statements and applying the abstract semantic function \( \widehat{\ell} \) 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 the naïve formulation, 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 our combined interpretation can provide improved precision.

4.1 Naïve Combination

To motivate the need for mostly-concrete interpretation, we elaborate on the strawman CONCERTO₀ and enumerate why it is impractical as a basis for combined analysis.

CONCERTO₀ executes framework code (statements with label ℓᵣ) concretely and abstractly interprets application code (statements with label ℓₐ). 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 Š. 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 Ƒ, 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, CONCERTO₀ 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 fops), CONCERTO₀ 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 fops will be analysis-time deterministic, leading to an explosion in reaching concrete states. Further, CONCERTO₀ 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 ℒ into two sets, ℒₐ = {ℓᵣ → ℓₐ*} ∪ {ℓₐ* → ℓ₀ | ℓₐ ∈ pred(ℓᵣ)} and ℒₕ = {ℓ₀ → ℓₕ*} ∪ {ℓₕ* → ℓ₀ | ℓᵣ ∈ pred(ℓᵣ)}. ℒₐ are flow edges originating in ℓᵣ-labeled statements, and symmetrically for ℒₕ and ℓᵣ. We define this combined domain ͜R₀ and an abstraction function a₀ : R → ͜R₀ as:

\[ \begin{align*}
\{a₀(r) = (λℓ → ℒₕ. r[ℓ], λℓ → ℒₐ, aₐ(r)[ℓ]) \mid ℓₛ \in ℒₐ \land ℓ₀ \in ℒₕ \}\}
\end{align*} \]

Theorem 1. C₀ is sound, i.e., a₀ ⊗ F ⊆ R → ͜R₀ C₀ ⊗ aₐ.
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 $[m \mapsto ["b" \mapsto "f", "a" \mapsto "\#"], p \mapsto [+]]$ at the call to `dispatch` in Section 2 yields the following set of concrete states:

$$\bigcup_{e \in E} \left\{ ([m \mapsto ["b" \mapsto "f", "a" \mapsto "\#"], p \mapsto n], e) \mid n > 0 \right\}$$

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 `fops` 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 states the conditions under which combined interpretation yields improved precision.

### 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 `fops` are deterministic and environment agnostic,1 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 $F$, the mostly-concrete domain uses powersets of values in $V_f$ extended with a special "unknown" value. For $\mathcal{A}$, the abstract

---

1We say an `fop` is environment agnostic if it does not depend on the environment model, i.e., $\forall e_1, e_2 \in E, v_1, \ldots, v_n \in V_f, \{v \mid \langle v, _\rangle \in [fop][e_1, v_1, \ldots, v_n]\} = \{v \mid \langle v, _\rangle \in [fop][e_2, v_1, \ldots, v_n]\}$
interpretation provides an abstraction domain \( \hat{A} \) that satisfies the properties described in Section 4.3.
Informally, the \( \hat{A} \) 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 \( \bar{R} \) is:
\[
\bar{R} = \mathcal{L} \rightarrow \bar{S} = (X_f \rightarrow \varphi(V_f) \uparrow) \times (X_a \rightarrow \hat{A})
\]

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. \( \varphi(V_f) \uparrow \) is the powerset domain of concrete values in \( \mathcal{F} \), extended with \( \top_\mathcal{F} \), which is the “unknown” value described in Extension 2. The abstractions for variables with types in \( \mathcal{A} \) in the mostly-concrete interpreter are drawn from the domain \( \hat{A} \) provided by the AI; \( \hat{A} \) is assumed to form a complete lattice. \( \hat{A} \) may or may not be used internally by the abstract interpreter. The domain of \( \bar{S} \) and \( \bar{R} \) 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 provides a monotone complete join-morphism \( \alpha_v : \varphi(V_a) \rightarrow \hat{A} \) that abstracts a set of concrete values of type \( \mathcal{A} \) to a value in \( \hat{A} \).

**Example 4.1** (Signedness Analysis). For the signedness analysis, \( \hat{A} \) is the same domain as used in the abstract interpretation, \( \varphi((-0, +)) \). \( \alpha_v \) is defined as: \( \alpha_v(I) = \{\text{sign}(i) \mid i \in I\} \) where \( \text{sign} \) 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 \( \text{arg} \) 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, \( \hat{A} = \text{Intv} \), where \( \text{Intv} \) is the plain interval domain [Cousot and Cousot 1977]. \( \alpha_v \) is defined as \( \alpha_v(I) = [lb, ub] \) or \( \bot \) if \( I = \emptyset \), where \( ub \) and \( lb \) return the upper (resp. lower) bound of a set, or \( \infty \) (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 \( \hat{A} \) for reasons discussed in Section 4.3.

We now define an abstraction function \( \alpha_F \) as follows:
\[
\alpha_F(r)[\ell] = \lambda x : X_f. \mathcal{V}(r[\ell], x), \lambda x : X_a. \alpha_v(\mathcal{V}(r[\ell], x))
\]

Where \( \mathcal{V} \) is the reaching value set for a set of states and variable, defined as: \( \mathcal{V}(S, x) = \{s(x) \mid \langle s, \_ \rangle \in S \land s(x) \neq \bot_\mathcal{V} \} \). \( \alpha_F \) 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 \( \alpha_v \) 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 \( \alpha_v \) is a complete join morphism, \( \alpha_F \) is also a complete join-morphism, and thus by [Cousot and Cousot 1979b], there exists some \( \gamma_F \) such that \( R \xrightarrow{\gamma_F} \bar{R} \).

In Fig. 7, we define the mostly-concrete semantic function \( I_\mathcal{F} : \bar{R} \rightarrow \bar{R} \). The structure of \( I_\mathcal{F} \) 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., \( \top_\mathcal{F} \)), then both branches are taken. The lifted version of \( f_\text{op} \) in Fig. 7 assumes that if any argument of an \( f_\text{op} \) is unknown (again, \( \top_\mathcal{F} \)) then the result is also unknown. Otherwise, the lifted version computes the result of applying the \( f_\text{op} \) 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 \( \top_\mathcal{F} \) value, which simplifies our formal presentation. In practice, our implementation falls back on \( \top_\mathcal{F} \) for infinite to unmanageably large sets of values (Section 7).
\[ I(\bar{\rho}(e)) \sim e^* = \text{step}^{\top}( \bigcup_{p \in \text{pred}(\ell)} \bar{\rho}(p^* \sim e^*), \ell) \cup \{ \text{step}^{\top}(\downarrow_S, \ell) \} \quad \ell = s \quad \text{o.w.} \quad I(\bar{\rho}(p^* \sim e^*)) = \begin{cases} \bar{\rho}(p) & \text{w.r.} \\ \top & \text{o.w.} \end{cases} \]

Where \( \bar{\rho} \) is the lifting of \( R \) defined by:

\[ \bar{\rho}(\ell) = \begin{cases} \tilde{i} & \text{s} = \text{if} \ldots \text{goto} \\
\tilde{i}n[x] & s = x = y \\
\tilde{i}n[x] \in [bcf] & s = x = bcf \\
\tilde{i}n[x] \in [fop] & s = x = \text{fop}(v_1, \ldots, v_n) \\
\tilde{i}n[x] \in T_A & s = x = \text{aexp} \land \text{type}(x) = A \end{cases} \]

\[ \tilde{\varphi}(\ell') = \begin{cases} \bar{\varphi}(\ell') = \begin{cases} \tilde{i} & \text{s} = \text{if} \ldots \text{goto} \\
\tilde{i}n[\ell'] & s = x = y \\
\tilde{i}n[\ell'] \in [bcf] & s = x = bcf \\
\tilde{i}n[\ell'] \in [fop] & s = x = \text{fop}(v_1, \ldots, v_n) \\
\tilde{i}n[\ell'] \in T_A & s = x = \text{aexp} \land \text{type}(x) = A \end{cases} & \text{o.w.} \end{cases} \]

Fig. 7. Mostly-concrete semantic function. In \( \text{step}^{\top} \), \( s \) is defined as \( \text{prog}(\ell) \).

\[ R = (L_F \rightarrow \tilde{S}) \times (L_A \rightarrow \tilde{S}) \quad \alpha_C(r) = \begin{cases} \tilde{\alpha}_C : L_F \cdot \alpha_C(r)[\tilde{\ell}] & \tilde{\alpha}_C : L_A \cdot \alpha_C(r)[\tilde{\ell}] \end{cases} \]

\[ \bar{\varphi} : \tilde{S} \rightarrow \tilde{S} \text{ such that: } \forall \tilde{\varphi}, r : R \cdot \alpha_C(r)[\tilde{\ell}] \in_R \tilde{\varphi}(\alpha_C(r)[\tilde{\ell}]) \quad \tilde{\varphi} : \tilde{S} \rightarrow \tilde{S} \text{ where: } \text{inj}(\tilde{m}, \tilde{n})[\tilde{\ell}] = \begin{cases} \tilde{m}[\tilde{\ell}] & \tilde{\ell} \in L_F \\
\tilde{m}[\tilde{\ell}] & \tilde{\ell} \in L_A \end{cases} \]

Fig. 8. Combined mostly-concrete and abstract domains; domain transformers; and injection functions.

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

Unlike \( fops \), 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 \( aops \) and therefore does not need to “understand” the abstraction domain \( \tilde{\mathcal{A}} \). 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 \( \ell_a \), 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.** \( \alpha_F \circ F \subseteq_R \text{step} \circ \alpha_F \), i.e., \( \text{step} \) is a sound over-approximation of \( F \).

**Proof Sketch.** By case analysis on the definitions of \( \text{step}^F \) and \( \text{step}^{\top} \). The full proofs are available in the accompanying appendix.

1:13

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 \( \widehat{R} \). Like it’s strawman counterpart \( \widehat{R}_0, \) \( \widehat{R} \) represents reaching states in the application with abstract state \( \widehat{S} \), but uses mostly-concrete states \( \widehat{S} \) for the framework instead of \( \phi(S). \) 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 \( \widehat{R} \) is defined in Fig. 8, along with an abstraction function \( \alpha_C. \) As \( \alpha_A \) and \( \alpha_F \) are complete join morphisms, \( \alpha_C \) is itself a complete join morphism, and thus there exists some \( \gamma_C \) such that \( \overline{R} \) and \( \overline{\widehat{R}} \) form a Galois connection. The monotone functions \( \overline{\tau} \) and \( \overline{\widehat{\tau}} \) in Fig. 8 are the domain transformers described above: \( \overline{\tau} \) transforms a state from the abstract interpreter into a mostly-concrete state, and \( \overline{\widehat{\tau}} \) performs a transformation in the opposite direction. They are both functions provided by the analysis that must fulfill the following conditions:

\[
\forall \vec{t}, r : R, \overline{\tau}(\alpha_A(r)[\vec{t}]) \sqsubseteq_S \overline{\tau}(\alpha_F(r)[\vec{t}]) \tag{12} \quad \forall \vec{t}, r : R, \overline{\alpha}(r)[\vec{t}] \sqsubseteq_S \overline{\tau}(\alpha_A(r)[\vec{t}]) \tag{13}
\]

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 \( \overline{\widehat{R}} \) must be discarded when moving to the non-relational domain \( \overline{R}. \) The \( \overline{\widehat{A}} \) values produced by \( \alpha_F \) are the result of a non-relational abstraction function \( \alpha_V, \) and by the inequality of Eq. (13), the result of \( \overline{\tau} \) can do no better. Despite this restriction on relational abstractions in the mostly-concrete domain, the above requirements on 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 \( \alpha_V \) from Example 4.1. The two state representations are equal \( \widehat{S} = S, \) and \( \overline{\widehat{\tau}} = \widehat{\tau} = id. \) In other words, the two domains have the same expressive power.

Example 4.4 (Relational Domain). Suppose instead representing integers with of the signedness domain used in Example 3.2, we used the Pentagon domain of Example 4.2 with the corresponding \( \alpha_V \) and \( \widehat{A} = Intv. \) Then the abstract domain is \( X_a \rightarrow Intv \times X_a \rightarrow \phi(X_a) \times X_f \rightarrow \phi(V_f)^T, \) 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 \( \phi(V_f)^T \) 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., \( fops \) will not appear in code analyzed by the abstract interpreter.

The domain transformers may be defined as:

\[
\overline{\tau}(\langle b, s, m \rangle) = \langle b, m \rangle \quad \overline{\widehat{\tau}}(\langle b, m \rangle) = \langle b, \lambda x : X_a. 0, m \rangle
\]

That is, \( \overline{\tau} \) discards the relational information from \( \widehat{S} \) when moving to \( \widehat{S}, \) and \( \overline{\widehat{\tau}} \) 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., \( \widehat{S} \leq_{\overline{\widehat{\tau}}} S. \)

Example 4.5 (Trivial Abstract Domain). Consider a domain \( X_a \rightarrow \phi(\langle - , 0, + \rangle) \times X_f \rightarrow 1, \) where \( 1 \) is unary domain whose single element \( t \) represents “any possible value”, i.e., the analysis does not
Theorem 3. \( \alpha_C \circ F \subseteq R \rightarrow \bar{R} \quad C \circ \alpha_C \)

Proof Sketch. By Theorem 2, the assumed soundness of \( \bar{F} \), and from the fact that \( \alpha_F \subseteq \bar{inj} \circ \alpha_C \) and that \( \alpha_A \subseteq \bar{inj} \circ \alpha_C \). \( \square \)

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

4.4 Conditions for Increased Precision

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

First, we define a function \( \text{LB} : \bar{R} \rightarrow \bar{R} \) as shown in Fig. 9. Intuitively, \( \text{LB} \) provides a lower-bound on the precision of the abstract semantic function; showing that \( I_T \) can "do better" than this lower bound will imply that \( I_T \) provides improved precision compared to \( \bar{F} \) on framework code.

\( \text{LB} \) 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 \( \text{LB} \) is non-relational, the definition implies that \( \bar{F} \) must also be non-relational.

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

\[
\begin{align*}
\tau \circ \tau &= \text{id} & \text{LB} \circ \tau &\subseteq \bar{R} \rightarrow \bar{R} & \tau \circ \bar{F} &\subseteq \bar{S} \circ \tau \circ \text{lfp} \tau (\bar{S}) & \forall U \subseteq \bar{S} \circ \tau \circ \text{lfp} \tau (\bar{S}) = \cup_{z \in U} \bar{S} \circ \tau (\bar{S}) 
\end{align*}
\]  

(19)
That is, if no precision is lost by moving to the domain \( \bar{S} \) and then back to \( \hat{S} \), if \( LB \) is a lower bound on the precision of \( \hat{F} \), and if \( \tilde{T} \) is a complete join morphism.

As \( C \) and \( \bar{F} \) operate over different domains, we first introduce a function \( \text{proj} : \bar{R} \rightarrow \bar{R} \) to project the \( \bar{R} \) domain into the combined domain \( \bar{F} \):

\[
\text{proj}(\bar{s}) = (\lambda \ell' : L.F.\tilde{T}(\bar{s}[\ell'])).
\]

\( \text{proj} \) uses \( \tilde{T} \) to translate abstract to mostly-concrete states, reversing the \( \text{inj} \) operation.

If the above conditions hold, we can prove:

**Theorem 4.** \( \text{inj}(lfp \ C) \subseteq \bar{R} \ lfp \bar{F} \)

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

**Example 4.7** (Signedness Analysis). The signedness example of Section 2 satisfies the above conditions, and thus combined interpretation with \textbf{CONCERTO} is at least as precise as plain abstract interpretation. As both \( \bar{T} \) and \( \tilde{T} \) 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, \textbf{CONCERTO} does not need AI developers to provide the full domain transformers (\( \bar{T} \) and \( \tilde{T} \)) 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:

\[
\begin{align*}
\text{prog} & ::= \ D^* \\
\text{D} & ::= \ \text{proc} \ f(p)( \ (\text{stmt}^f | \ \text{stmt}^G)^*) \\
\text{stmt} & ::= \ldots | \ \text{return} \ x \ | \ x = f(y)
\end{align*}
\]

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

We extend the state definition to track the runtime stack: \( R = \mathcal{L} \rightarrow \varphi(S^* \times \ell^* \times \mathcal{E}) \), where \( S^* \) is a sequence of states \( S \) as defined in Section 3.2, and \( \ell^* \) 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 \( p \).

We have so far treated procedure call and return as orthogonal to control-flow transfer between the
\(\ell\) words, each procedure may only contain \(D\) As a consequence, instead of using
\(f\) from the application to framework, the AI simply provides a mostly-concrete representation of the
state, transforming the mostly-concrete values into a "native" abstract representation as necessary.
If the abstract interpretation is also sound with respect to this updated
definition, and the \(\tau\) and \(\tilde{\tau}\) 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.

\[ F(\rho)[\ell_c^r \leadsto \ell_c^*] = \bigcup_{p \in \text{pred}(\ell_c)} \{ (s \circ s_x[x \mapsto s_c[\rho]],[\mathcal{R},E]) | (s \circ s_y \circ s_c, [\mathcal{R},E]) \in r[\ell_c^r \leadsto \ell_c^*] \} \]

\[ I_r(\tilde{\tau})(\ell_c^r \leadsto \ell_c^*) = \bigcup_{p \in \text{pred}(\ell_c)} \tilde{\tau}(\ell_c)[x \mapsto \tilde{\tau}([p^* \leadsto \ell_c^*][\rho])] \]

\[ F(\rho)[p^* \leadsto \ell_c^*] = \{(s, [\mathcal{R},E]) | (s, [\mathcal{R} \circ \ell, E]) \in r[p]\} \]

\[ I_r(\tilde{\tau})([p^* \leadsto \ell_c^*] = \tilde{\tau}(p) \]

\[ I_r(\tilde{\tau})([\ell_c^r \leadsto \ell_c^*]) = \{ [p \mapsto \tilde{\tau}(\ell_c)[y]] | (p \circ s_x \circ s_c, [\mathcal{R},E]) \in r[\ell_c^r \leadsto \ell_c^*] \} \]

\[ \text{step}(\tilde{\tau}(\ell), \ell) = \begin{cases} \{ \text{in}(\rho \mapsto \text{in}(x)) \} & \text{if } \text{prog}[\ell] = \text{return } x \\ \text{as before} & \text{otherwise} \end{cases} \]

\[ \text{step}(\tilde{\tau}(\ell), \ell) = \begin{cases} \{ \text{in}(\rho \mapsto \text{in}(x)) \} & \text{if } \text{prog}[\ell] = \text{return } x \\ \text{as before} & \text{otherwise} \end{cases} \]

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 \(\ell_f\) and \(\ell_a\) statements. In practice, we
require all transfers between the application and framework occur at procedure boundaries. In other
words, each procedure may only contain \(\ell_a\) statements or \(\ell_f\) 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 \(\tau\) at application-to-framework calls, the AI may directly provide
mostly-concrete arguments to the mostly-concrete component, which then binds the arguments in
an 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 be-
cause our language is statically scoped and has no global variables, heaps are discussed in Section 8.)

As a consequence, instead of using \(\tau\) at framework-to-application calls, the mostly-concrete inter-
preter 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 \(\tau\) at return flows
from the application to framework, the AI simply provides a mostly-concrete representation of the

return slot \( \rho \). We likewise assume that the AI is interested only in the value of \( \rho \) 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 \( \tau \) and \( \bar{\tau} \), but to retain soundness, the values exchanged must be consistent with those produced by some \( \bar{\tau} \) and \( \bar{\tau} \) that satisfy the definitions given in Section 4.3. For example, consider an application-to-framework call where the abstract entry state is \( \bar{s} \). The mostly-concrete argument provided to the mostly-concrete interpreter must therefore be \( \bar{\tau}(\bar{s})[\rho] \) for some \( \bar{\tau} \), and where \( \rho \) is the parameter of the called procedure. Thus, any valid domain transformers \( \bar{\tau} \) and \( \bar{\tau} \) provide correctness specifications for the exchange of values. However, a trivial way to ensure soundness is to generate complete definitions for some \( \bar{\tau} \) and \( \bar{\tau} \), 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 \( \lambda \bar{s}.\bar{\tau}(\bar{s})[\rho] \).

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. We have shown in Appendix C that this process converges to the least-fixed point of C.

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. 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 \( \bar{R} \) does not possess an ACC property 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 \( \lambda \bar{s}.\tau_{F}(\bar{s}) \) in practice to avoid materializing infinite sets.
We have implemented a prototype combined interpreter for a subset of Java. 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, \texttt{read()}, reads an integer from a deterministic stream. This primitive models reading from a deterministic configuration file.

The second primitive, \texttt{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 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.

\textbf{Widening.} Following the vocabulary of [Bourdoncle 1993], we require two \textit{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 $\nabla_A$ for values of type $\widetilde{A}$ and $\nabla$ for abstract states $\tilde{\mathcal{S}}$. From $\nabla_A$, we derive a widening operator for mostly-concrete states:

\[ (m_f, m_a) \nabla (m'_f, m'_a) = \left\{ \begin{array}{ll} m_f' & \text{if } m'_f \leq m_f[x] \\ \top_{o.w.} & \text{otherwise} \end{array} \right. \]

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 $\tilde{F}$ and $I_{\tau}$ 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 \textit{lfp} $C$.

\textbf{Precision.} Section 4.4 gave conditions for when \textit{lfp} $C$ will be at least as precise as \textit{lfp} $\tilde{F}$. Whether this increased 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 $\tilde{F}$ and $I_{\tau}$ 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.

\textbf{Finitization.} Finally, to ensure Concerto manipulates only finite sets, we require that the AI does not provide infinite argument or return value representations to the mostly-concrete interpreter. We also extend the definition of $I_{\tau}$ to finitize the result of all $\textit{fop}$ operations. Whenever applying $\textit{fop}$ 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 $\top_{o.w.}$. This finitization is sound, while avoiding materializing infinite sets in our implementation.

\section{8 \textbf{EXTENSIONS FOR A REALISTIC PROTOTYPE}}

We have implemented a prototype combined interpreter for a subset of Java. 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, \texttt{read()}, reads an integer from a deterministic stream. This primitive models reading from a deterministic configuration file. The second primitive, \texttt{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 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).
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 \( F \) to be the framework classes and \( A \) the application classes.

We also extend our concrete state to include a concrete heap: \( S = E \times (X \to V) \times H \). Object allocation and field manipulation are defined via \( aops \) and \( fops \) 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. 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., \( S = E \times (X \to V) \times H_a \times H_f \). 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 \( \overline{H}_f \).\(^2\) \( \overline{H}_f \) 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 heap component.\(^3\) CONCERTO does not make any assumptions on the internal heap representation used by the abstract interpretation. However, the abstract interpretation must provide two functions: \( \text{project}_H : \overline{S} \to \overline{H}_a \) and \( \text{inject}_H : \overline{H}_a \to \overline{S} \to \overline{S} \) such that \( \forall \overline{s}, \overline{s}' \subseteq \overline{S} \): \( \text{inject}_H (\text{project}_H \overline{s}) \subseteq \overline{s} \) for some type \( \overline{H}_a \) defined by the analysis. The AI must provide widening, join, and ordering operations for \( \overline{H}_a \).

To tie these two heap representations together, we extend the mostly-concrete state representation to include mostly-concrete and abstract heaps: \( \overline{S} = (X_f \to \varphi(V_f)\top) \times (X_a \to \hat{A}) \times \overline{H}_f \times \overline{H}_a \). \( V_f \) is the domain of mostly-concrete heap locations, and \( \hat{A} \) is an abstract representation of objects of type \( A \). During execution, the mostly-concrete interpreter updates the mostly-concrete heap and threads the 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, \( \overline{S} \times \overline{H}_f \). 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 \( F \) with the domain \( \varphi(V_f)\top \). Next, when performing a write of \( v' : \varphi(V_f)\top \) to a field-variable of type \( F \) with the existing value \( v : \varphi(V_f)\top \), the value \( v'' \) of the field-variable after the write must satisfy the constraint \( v'' \subseteq v' \cup v \), 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 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 \( A \) 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 \( \overline{\text{Loc}} \to \hat{O} \) where \( \overline{\text{Loc}} \) is a finite domain of abstract locations and \( \hat{O} \) are abstract objects, and to take \( \hat{A} = \varphi(\overline{\text{Loc}}) \) and \( \overline{H}_a = \overline{\text{Loc}} \to \hat{O} \).

---

\(^2\)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.

\(^3\)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 \( H_a \), and

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 \texttt{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 \( \hat{r} \). When Concerto encounters a method call in the mostly-concrete interpreter with a base pointer \( \hat{r} \) of type \( \hat{A} \) (i.e., static type \( A \)), it yields into the abstract interpreter by passing the mostly-concrete arguments and \( C \) to \texttt{interpret}. \texttt{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, \texttt{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. Concerto is parameterized over the choice of contexts; when computing the analysis context for application method call, Concerto passes information about the current state, call site, and call stack to a \texttt{mkContext} method exposed by the AI. \texttt{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 \texttt{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 Shared Types and Primitives

In the language presented thus far, the choice of whether a variable is modeled by an abstract value or mostly-concrete value could be made statically by dividing classes into framework or application
classes. However, this approach is infeasible when adding primitive types, interfaces, and shared libraries to our supported subset of Java. Although we could restrict the use of primitives and libraries to only application or framework code, and require that all implementers of an interface must be either application or framework classes, such a restriction would be unrealistic.

### 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: $\varphi(\mathbb{N})^+ + \widehat{A}$. When an integer value of type $A$ 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 type of abstractions for objects and integers.

When executing an arithmetic or comparison operation, CONCERTO checks if both operands are of type $\varphi(\mathbb{N})^T$. If so, then the interpreter executes the concrete arithmetic operation lifted to the $\varphi(\mathbb{N})^T$ domain. If one or both of the operands are of type $\widehat{A}$, then CONCERTO uses a \texttt{lift} : $\varphi(\mathbb{N})^T \rightarrow \widehat{A}$ 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 $\varphi(\mathbb{N})^+ + \widehat{A}$ into $\varphi(\mathbb{N})^T$. The abstract interpretation must also expose a function $\texttt{lower} : \widehat{A} \rightarrow \varphi(\mathbb{N})^T$. A sound choice for this function is to simply return $\top_D$.

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 $\varphi(V_f) \times \widehat{A}$ for values with an interface type. Intuitively, $\langle v_f, a \rangle : \varphi(V_f)^T \times \widehat{A}$ represents \textit{either} a framework object that is abstracted by $v_f$ \textit{or} an application object abstracted by $a$. 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 \texttt{ArrayList}, \texttt{HashMap}, etc. are ubiquitous. CONCERTO supports these \textit{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 \texttt{ArrayList} allocated in the application will be modeled using abstract values, whereas an \texttt{ArrayList} allocated in the framework will be modeled mostly-concretely. This approach is possible because we assume that library types, like interfaces, only export functional 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.
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.

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 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 $Inv$ and inequalities between access paths, giving a weakly relational Pentagon domain [Logozzo and Fähndrich 2008]. As the choice of $\tilde{A}$ must be non-relational, the abstract representation of integers in $\tilde{A}$ is simply $Inv$. 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 235× 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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>CS?</th>
<th>Heap</th>
<th>Domain</th>
<th>Relational?</th>
<th>PS?</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTA</td>
<td>No</td>
<td>Type-based</td>
<td>Reaching Types</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>IFlow</td>
<td>Caller Method</td>
<td>Type-based</td>
<td>Access Paths/Reaching Types</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ABC</td>
<td>Call site 1-CFA</td>
<td>Abstract Location</td>
<td>Pentagons/Abstract Locations</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
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 29 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 [Nielsen 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.
REFERENCES


A PROOFS FOR SECTION 4

A.1 Soundness of $I_T$

We now prove that $\alpha_F \circ F \subseteq I_T \circ \alpha_F$. We first note the following fact about $\alpha_F$ that we will exploit during our proofs:

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

$$
\alpha_e(\emptyset) = \bot \hat{\alpha} \quad (21)
$$

To begin, we will ignore the “initial state” term from the definition of $F$, and first prove $I_T$ sound with respect to

$$
F_0(r)[\bar{\ell}] = \begin{cases} 
\bigcup_{p \in \text{pred}(\ell)} \text{step}^F(\text{in}, \ell) & \bar{\ell} \equiv s^* \leadsto s^* \\
F(r)[\ell] & \text{o.w.}
\end{cases}
$$

which is the original concrete semantic function without the initialization term. We first prove that the inequality holds for some arbitrary $r$ and $\ell^o \leadsto \ell^*$. We first need the following lemmas.

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

**Proof.** For every $x$, clearly $\mathcal{V}(F_0(r)[\bar{\ell}], x) = \mathcal{V}([\ell \mapsto \mathcal{F}, \mathcal{L}])[\bar{\ell}], x)$, and thus by (20), $\alpha_F([\ell \mapsto \mathcal{F}, \mathcal{L}])[\bar{\ell}][x] = \alpha_F(F_0(r))([\ell])[x]$. By transitivity, we have $\alpha_F \circ F_0(r)[\ell] \subseteq \bar{s}$. $\Box$

Intuitively, **Lemma 1** ensures that to establish pointwise inequality, it suffices to consider the abstract state for each $\ell$ individually.

**Lemma 2.**

$$
\forall r, \ell, p \in \text{pred}(\ell), \bar{s}. 
\left(\forall \nu. (\text{in}, E) \in r[p^* \leadsto \ell^o] \land \alpha_F([\ell \mapsto \text{step}^F(\text{in}, E), \ell])[\ell] \subseteq \bar{s}\right) \\
\Rightarrow \alpha_F([\ell \mapsto \mathcal{F}, \mathcal{L}])[\ell] \subseteq \bar{s}
$$

where $\mathcal{F}, \mathcal{L}$ is defined as in **Lemma 1**.

**Proof.** First, we show that $\bot_R[\ell \mapsto \bar{s}]$ is an upper bound for the set $\mathcal{M} = [[\ell \mapsto \text{step}^F(\text{in}, E), \ell]] | p \in \text{pred}(\ell) \land (\text{in}, E) \in r[p^* \leadsto \ell^o]$. Let $m$ be some element of $\mathcal{M}$. By assumption, $\alpha_F(m)[\ell] \subseteq \bar{s}$, so it remains to show that $\forall \nu. \alpha_F(m)[\ell'] \subseteq \bot_R[\ell \mapsto \bar{s}][\ell'] = \bot_{\bar{s}}$. This follows from the fact that for any $x \alpha_F(\bot_R)[\ell'][x] = \bot_{\bar{s}}$ (where $\bot_{\bar{s}}$ is $\bot_A$ 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 $\alpha_F(m)[\ell'][x] = \bot_{\bar{s}}$.

Next, observe that $[\ell \mapsto \mathcal{F}, \mathcal{L}] = [\ell \mapsto \bigcup_{(\text{in}, E) \in r[p^* \leadsto \ell^o]} \text{step}^F((\text{in}, E), \ell)]$, i.e., $\bigcup \mathcal{M}$. As $\alpha_F$ preserves least upper bounds, and from the fact that $\bot_R[\ell \mapsto \bar{s}]$ is an upper bound of the set $\mathcal{M}$, we have that: $\alpha_F(\bigcup_{(\text{in}, E) \in r[p^* \leadsto \ell^o]} [\ell \mapsto \text{step}^F((\text{in}, E), \ell)]) \subseteq \bot_R[\ell \mapsto \bar{s}]$, from which it is immediate that $\alpha_F([\ell \mapsto \mathcal{F}, \mathcal{L}])[\ell] \subseteq \bar{s}$. $\Box$

**Lemma 2** implies that to show an abstract state $\bar{s}$ over-approximates the result of stepping all incoming concrete states, it suffices to show that $\bar{s}$ over-approximates stepping each individual incoming state. We now show that for any $\ell$ and $r$, $I_T(\alpha_F(r))[\ell^o \leadsto \ell^*]$ is such an $\bar{s}$.

**Lemma 3.** $\forall r, \ell, p \in \text{pred}(\ell), (\text{in}, E) \in r[p^* \leadsto \ell^o]$ :
∀x, y. \mathcal{V}(\text{step}^F(⟨\langle in, E⟩, \ell⟩), x) ⊆ \mathcal{V}(⟨\langle in, E⟩⟩, y)
\Rightarrow \alpha_F([\ell \mapsto \text{step}^F(⟨\langle in, E⟩, \ell⟩)])[\ell][x] ⊆ \left( \bigcup_{p' \in \text{pred}(\ell)} \alpha_F(r)[p'^* \sim \ell^o] \right)[y]

(2) \alpha_F([\ell \mapsto \text{step}^F(⟨\langle in, E⟩, \ell⟩)])[\ell] \subseteq I^\top(\alpha_F(r))[\ell]

\textsf{Proof.}

(1) \langle in, E⟩ ∈ r[p^* \sim \ell^o] implies that \mathcal{V}(⟨\langle in, E⟩⟩, y) ⊆ \mathcal{V}(r[p^* \sim \ell^o], y), whence by transitivity and (20) we then have:
\alpha_F([\ell \mapsto \text{step}^F(⟨\langle in, E⟩, \ell⟩)])[\ell][x] ⊆ \alpha_F(r)[p^* \sim \ell^o][y]

Transitivity and least upper bounds gives
\alpha_F([\ell \mapsto \text{step}^F(⟨\langle in, E⟩, \ell⟩)])[\ell][x] ⊆ \alpha_F(r)[p^* \sim \ell^o][y] ⊆ \left( \bigcup_{p' \in \text{pred}(\ell)} \alpha_F(r)[p'^* \sim \ell^o] \right)[y]

(2) By the pointwise nature of the domains, it suffices to show that:
\forall x. \alpha_F([\ell \mapsto \text{step}^F(⟨\langle in, E⟩, \ell⟩)])[\ell][x] ⊆ \text{step}^\top(\tilde{\text{in}}, \ell)[x]
\subseteq I^\top(\alpha_F(r))[\ell][x]

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

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

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

\textbf{Case (6):} We must only establish the inequality for the left-hand side of the assignment. As \mathcal{V}(\text{step}^F(⟨\langle in, E⟩, \ell⟩), \text{lhs}) = \mathcal{V}(⟨\langle in, E⟩⟩, \text{rhs}), we have by part (1) above:
\alpha_F([\ell \mapsto \text{step}^F(⟨\langle in, E⟩, \ell⟩)])[\ell][\text{lhs}] \subseteq \text{in}[\text{rhs}] = \text{step}^\top(\tilde{\text{in}}, \ell)[\text{lhs}]

\textbf{Case (7):} As above, we need only establish the relationship holds for the \text{lhs}. By definition of \text{step}^F, \mathcal{V}(\text{step}^F(⟨\langle in, E⟩, \ell⟩), \text{lhs}) = \mathcal{V}(⟨\langle in[\text{lhs} \mapsto [bc_{\ell}⟩], E⟩⟩), \text{lhs}) = ([bc_{\ell}]). By the definition of \alpha_F:
\alpha_F([\ell \mapsto \text{step}^F(⟨\langle in, E⟩, \ell⟩)])[\ell][\text{lhs}] = ([bc_{\ell}])
= \text{in}[\text{lhs} \mapsto [bc_{\ell}])][\text{lhs}]
= \text{step}^\top(\tilde{\text{in}}, \ell)[\text{lhs}]

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

\textbf{Case (9):} By definition of \alpha_F, \alpha_F([\ell \mapsto \text{step}^F(⟨\langle in, E⟩, \ell⟩)])[\ell][\text{lhs}] = \tilde{\mathcal{A}}: \tilde{\mathcal{A}} \subseteq \top = \text{step}^\top(\tilde{\text{in}}, \ell)[\text{lhs}].
\Box
Let us now prove soundness of $I_T$ w.r.t $F_0$ for $\ell^0 \rightsquigarrow \ell^*$:

**Lemma 4.** $\forall \ell, r. \alpha_F \circ F_0(r)[\ell^0 \rightsquigarrow \ell^*] \subseteq I_T \circ \alpha_F(r)[\ell^0 \rightsquigarrow \ell^*]$

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

**Lemma 5.** $\forall \ell \circ \circ, r. \alpha_F \circ F_0(r)[\ell^0 \rightsquigarrow \ell^0] \subseteq I_T \circ \alpha_F(r)[\ell^0 \rightsquigarrow \ell^0]$

**Proof.** As $\alpha_F(\emptyset) = \emptyset$, if $F_0(r)[\ell^0 \rightsquigarrow \ell^0] = \emptyset$, the inequality trivially holds. Otherwise, by definition, $F_0(r)[\ell^0 \rightsquigarrow \ell^0] \subseteq \alpha_F(F_0(r))(\ell^0 \rightsquigarrow \ell^0)$ for $p$ is shorthand for $\ell, \alpha_F(F_0(r))[\ell^0 \rightsquigarrow \ell^0]$ is true, then $I_T(\ell) \subseteq \alpha_F(F_0(r))[\ell^0 \rightsquigarrow \ell^0]$ as required. It therefore suffices to show that $F_0(r)[\ell^0 \rightsquigarrow \ell^0] \neq \emptyset \Rightarrow \alpha_F(F_0(r), \ell \circ \circ) \circ \circ$. We proceed by cases on the form of $\text{prog}[p]$. If $p \in \ell_f$ and $\text{prog}[p]$ is conditional over $x$ and $y$ with target $\ell$, then there exists some state $s \in r[p]$ such that $s[x] = \top$ or $\alpha_F(r)[p][y] = \top$, then by definition $\alpha_F(F_0(r))[\ell \circ \circ] \subseteq \alpha_F(r)[\ell \circ \circ]$. A similar argument holds for when $p \notin \ell_f$, and $\text{prog}[p]$ is a condition with fallthrough target $\ell$. Finally, for any other statement, $\alpha_F(F_0(r), \ell \circ \circ) \circ \circ$ holds.

We can now prove our main result:

**Proof of Theorem 2.** For $\ell \neq s^\circ \Rightarrow s^\bullet$, $\alpha_F \circ F_0$ 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 $\ell \equiv s^\circ \Rightarrow s^\bullet$ (which we will abbreviate in the following as $s$).

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

$$\alpha_F([s \mapsto \bigcup_{p \in \text{pred}(\ell)} \text{step}^\circ (in, \ell)] \cup [s \mapsto \bigcup_{e \in \ell} \text{step}^\circ ([S, e], \ell)]) \subseteq I_T(\alpha_F(r))$$

As $\alpha_F$ is a complete join morphism, this is equivalent to showing that:

$$\alpha_F([s \mapsto F_0(r)(\ell)]) \subseteq \alpha_F([s \mapsto \bigcup_{e \in \ell} \text{step}^\circ ([S, e], \ell)]) \subseteq I_T(\alpha_F(r))$$

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

$$\alpha_F([s \mapsto \bigcup_{e \in \ell} \text{step}^\circ ([S, e], \ell)]) \subseteq I_T(\alpha_F(r))$$

for which it suffices to show that:

$$\alpha_F([s \mapsto \bigcup_{e \in \ell} \text{step}^\circ ([S, e], \ell)])[s] \subseteq \text{step}^\top (\exists_{\ell \succeq}, \ell) \subseteq I_T(\alpha_F(r))[s]$$

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

We conclude by noticing that we were careful to show that soundness of $I_T$ is preserved when any $\text{prog}[$ returns $\top$, 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 $\overline{F}$, for some input argument $X$.

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

Case $\vec{t} \in \mathcal{L}_F$: As above, by the soundness of $I_T$, we have that $\alpha_F(F(X))[\vec{t}] \sqsubseteq I_T(\alpha_F(X))[\vec{t}]$, so it suffices to show that $\alpha_F(X) \sqsubseteq \text{inj}(\alpha_C(X))$. As in the above case, the inequality immediately holds at labels in $\mathcal{L}_F$. Otherwise we have $\forall \vec{t}' \in \mathcal{L}_A, \text{inj}(\alpha_C(X))[\vec{t}'] = \tau(\alpha_A(X)) \equiv \alpha_F(X)[\vec{t}']$ by (13).

\[\square\]

A.3 Increased Precision

Lemma 6. $C \circ \overrightarrow{\text{proj}} \equiv_{\overrightarrow{R} \rightarrow \overrightarrow{R}} \overrightarrow{\text{proj}} \circ \widehat{F}$

Proof. Suffices to show that, for some $\vec{r}$, and for all $\vec{t}$, $C \circ \overrightarrow{\text{proj}}(\vec{t}) \equiv \overrightarrow{\text{proj}} \circ \widehat{F}(\vec{t})$. Before proceeding we observe that: $\text{inj} \circ \overrightarrow{\text{proj}} \equiv \vec{r}$ and $\text{inj} \circ \overrightarrow{\text{proj}} \equiv \text{id}$.

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

Case $\vec{t} \in \mathcal{L}_A$: Then $C \circ \overrightarrow{\text{proj}}(\vec{t}) = \vec{r} \circ \text{inj} \circ \overrightarrow{\text{proj}}(\vec{t})$. Because $\text{inj} \circ \overrightarrow{\text{proj}} \equiv \text{id}$, we must show $\widehat{F}(\vec{t}) = \vec{r} \circ \overrightarrow{\text{proj}} \circ \hat{F}(\vec{t})$ which holds trivially.

Case $\vec{t} \in \mathcal{L}_F$: Then we must show that $C \circ \overrightarrow{\text{proj}}(\vec{t}) = \hat{F}(\vec{t}) = \tau(\hat{F}(\vec{t}))$. By assumption, $LB \circ \vec{r} \equiv \vec{r} \circ \hat{F}$, so it suffices to show that $C \circ \tau(\hat{F}(\vec{t})) \equiv LB \circ \vec{r}(\hat{F}(\vec{t}))$. By cases on the form of $\vec{t} \in \mathcal{L}_F$:

Subcase $\vec{t} = \ell_f^o \leadsto \ell_f^*$: From the definitions of $I_T$ and $LB$, we must show that:

$\text{step}^T(\vec{s}, \ell_f) \sqsubseteq \text{step}^{LB}(\vec{s}, \ell_f)$

where $\vec{s} = \bigsqcup_{p \in \text{pred}(\ell_f)} \tau(\vec{p})[\vec{p}^* \leadsto \ell_f^o]$. For branches (15) and (16) in $\text{step}^{LB}$, the corresponding branch (5) and (6) in $\text{step}^T$ clearly yield identical results. As branch (18) returns $\vec{s}[x \mapsto \tau], f(x)$, and the corresponding branches (7) and (8) of $\text{step}^T$ produce states of the form: $\vec{s}[x \mapsto \nu]$ (where $\nu$ is some member of $\phi(V_T)^T$), the inequality trivially holds. Finally, we do not have to consider branch (17) of $\text{step}^{LB}$ nor branch (9) of $\text{step}^T$ because the syntactic constraints of the language rule such cases out for a label of the form $\ell_f$.

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

Subcase $\vec{t} = \ell_f^* \leadsto \ell_f^o$: By definition, $I_T(\tau(\vec{t}))[\ell_f^* \leadsto \ell_f^o]$ either returns $\tau(\vec{t})[\ell_f^o \leadsto \ell_f^*]$ or $\perp$. As $LB(\tau(\vec{t}))[\ell_f^* \leadsto \ell_f^o] = \tau(\vec{t})[\ell_f^o \leadsto \ell_f^*]$, the inequality trivially holds.

\[\square\]

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

$\begin{align*}
F^0 &= \perp \\
F^{\delta+1} &= F(F^{\delta}) \\
F^\lambda &= \bigsqcup_{\beta < \lambda} F^\beta \\
\widehat{F}^0 &= \perp \\
\widehat{F}^{\delta+1} &= \widehat{F}(\widehat{F}^{\delta}) \\
\widehat{F}^\lambda &= \bigsqcup_{\beta < \lambda} \widehat{F}^\beta
\end{align*}$

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

Proof. By transfinite induction.

Case $\delta = 0$: Then $F^0 = \perp = \mu(\perp) = \mu(\widehat{F}^0)$, where the equality between $\perp$ terms comes from the fact that $\mu$ 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.
We note that Eq. (20) still applies under the updated definition of $\text{inj}$.

\begin{equation}
F^\lambda = \bigsqcup_{\beta < \lambda} F^\beta \subseteq \bigsqcup_{\beta < \lambda} \mu(\tilde{F}^\beta) = \mu(\bigsqcup_{\beta < \lambda} \tilde{F}^\beta) = \mu(\tilde{F}^\lambda)
\end{equation}

Where the second-to-last equality follow from $\mu$ being a complete join morphism.

\end{proof}

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

\end{proof}

**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 $\alpha F$, this in turn implies that the values in the abstracted state are only influenced 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 step $^T$ 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 $\ell^\circ_c \rightsquigarrow \ell^*_c$:** By Lemma 1, it suffices to show that

$$
\alpha F([\ell_c \mapsto \bigsqcup_{p \in \text{pred}(\ell_c)} r[p^* \mapsto \ell^\circ_c])] = \bigsqcup_{p \in \text{pred}(\ell_c)} \alpha F(r[p^* \mapsto \ell^\circ_c])
$$

From the definition of the domain $R$, we have:

$$
\alpha F([\ell_c \mapsto \bigsqcup_{p \in \text{pred}(\ell_c)} r[p^* \mapsto \ell^\circ_c]]) = \alpha F(\bigsqcup_{p \in \text{pred}(\ell_c)} [\ell_c \mapsto \{\text{in}\}])
$$

By the definition of least upper bounds, it suffices to show that, for some arbitrary $p' \in \text{pred}(\ell_c), in \in r[p^* \mapsto \ell^\circ_c]$:

$$
\alpha F([\ell_c \mapsto \{\text{in}\}])[\ell_c] \subseteq \bigsqcup_{p \in \text{pred}(\ell_c)} \alpha F(r[p^* \mapsto \ell^\circ_c])
$$

We show that \( \alpha_F([\ell_c \mapsto \{in\}])[\ell_c] \subseteq \alpha_F(r)[p^\star \leadsto \ell_c^\circ] \), 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 \( in \in r[p^\star \leadsto \ell_c^\circ] \), it is immediate that:

\[
\mathcal{V}([\ell_c \mapsto \{in\}])[\ell_c], v) = \mathcal{V}(\{in\}, v) \subseteq \mathcal{V}(r[p^\star \leadsto \ell_c^\circ], v)
\]

whence by Eq. (20) we have that \( \alpha_F([\ell_c \mapsto \{in\}])[\ell_c][v] \subseteq \alpha_F(r)[p^\star \leadsto \ell_c^\circ][v] \), as required.

**Case** \( p^\star \leadsto \ell_c^\circ \): Trivial, by the definition of \( \alpha_F \) and reasoning similar to the above.

**Case** \( \ell_c^\circ \leadsto \ell_c^\circ \): We first abbreviate \( \ell_c^\circ \leadsto \ell_c^\circ \) as \( \mathcal{c} \) and (as usual) \( \ell_c^\circ \leadsto \ell_c^\circ \) as \( \ell_c \). By reasoning similar to that in case \( \ell_c^\circ \leadsto \ell_c^\circ \), it suffices to show that:

\[
\alpha_F([\ell \mapsto \{r \circ r \circ s_r \circ \{y\} \mapsto \{\mathcal{R} \circ \ell, E\} \}][\ell_c]) \in [p \mapsto \alpha_F(r)[\ell_c][y]]
\]

for which it suffices to show, for some arbitrary \( r \circ r \circ s_r \circ \{y\} \mapsto \{\mathcal{R} \circ \ell, E\} \in r[\ell_c] \) that:

\[
\alpha_F([\ell \mapsto \mathcal{K}])[\ell_c][v] = \alpha_F(\parallel \mathcal{R} \parallel)[\ell_c][v]
\]

where \( \mathcal{K} = \{r \circ r \circ s_r \circ \{y\} \mapsto \{\mathcal{R} \circ \ell, E\} \} \). 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:

\[
\alpha_F([\ell \mapsto \mathcal{K}])[\ell_c][v] = \alpha_F(\parallel \mathcal{R} \parallel)[\ell_c][v]
\]

It therefore remains to show that:

\[
\alpha_F([\ell \mapsto \mathcal{K}])[\ell_c][p] \subseteq [p \mapsto \alpha_F(r)[\ell_c][y]]\parallel \mathcal{R} \parallel [p] = \alpha_F(r)[\ell_c][y]
\]

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

**Case** \( \ell_c^\circ \leadsto \ell_c^\circ \): We will prove that, for some some arbitrary \( r, p \in pred(\ell_c), r \circ s_r \circ s_c, \{y\} \mapsto \{\mathcal{R} \circ \ell, E\} \in r[\ell_c^\circ \leadsto \ell_c^\circ] \) where \( r \circ s_r \circ s_c, \{y\} \mapsto \{\mathcal{R} \circ \ell, E\} \in r[\ell_c^\circ \leadsto \ell_c^\circ] \) that:

\[
\alpha_F([\ell_r \mapsto \mathcal{J}])[\ell_r] \subseteq \alpha_F(r)[\ell_c^\circ \leadsto \ell_c^\circ][p]
\]

where \( \mathcal{J} = \{r \circ s_r \circ \{x \mapsto s_c[p]\} \mapsto \{\mathcal{R} \circ \ell, E\} \} \)

For any variable \( v \neq x \), we have that \( \mathcal{V}(\mathcal{J}, v) \subseteq \mathcal{V}(r[\ell_c^\circ], v) \), whence the inequality holds by Eq. (20). For the variable \( x \), we have that \( \mathcal{V}(\mathcal{J}, x) \subseteq \mathcal{V}(r[p^\star \leadsto \ell_c^\circ], p) \), whence the inequality again holds by assumption on \( \alpha_F \).

\[\square\]

### 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:

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

We will abuse notation and for an element \( \langle \bar{m}, \bar{m} \rangle : \bar{R} \), define \( \langle \bar{m}, \bar{m} \rangle|_{\mathcal{L}_A} = \bar{m} \) and \( \langle \bar{m}, \bar{m} \rangle|_{\mathcal{L}_F} = \bar{m} \).

We define the subfixpoint iteration process as a function \( M : \bar{R} \to \bar{R} \):

\[
M(X) = \left\langle \bigcup_{i < \omega} U^i(\bar{m}j(X))|_{\mathcal{L}_A}, \left\langle \bigcup_{i < \omega} T(\bar{m}j(X))|_{\mathcal{L}_A}^i(\perp)|_{\mathcal{L}_F} \right\rangle U(X) = X|_{\mathcal{L}_F} \sqcup \bar{F}(X)|_{\mathcal{L}_A}
\]

\[
T(m)(X) = m|_{\mathcal{L}_A} \sqcup I_T(X)|_{\mathcal{L}_F}
\]
In the above definitions, the \( U \) and \( T \) play the role of iterating the two interpreters to fixpoint.\(^4\) In the definition of \( U \), after abstract semantic function \( \widetilde{F} \) is applied to the input argument, the results from this application in \( L_F \) are discarded in favor of the values in the input argument \( X \). Thus, while iterating \( U^i(\widetilde{m}(X)) \), the states at flow edges \( L_F \) are effectively fixed to \( \widetilde{m}(X) \), while the states at flow edges \( L_A \) evolve through repeated application of \( \widetilde{F} \). 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 \( I_T \). Unlike the \( \sqcup_{i \leq \omega} U^i(\widetilde{m}(X)) \) term, \( \sqcup_{i \leq \omega} T(\widetilde{m}(X)) \) 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) \( \forall m, m', X, X' : \bar{R}.m \sqsubseteq m' \land X \sqsubseteq X' \Rightarrow T(m)(X) \sqsubseteq T(m')(X') \)

(2) \( \forall m, m', X, X' : \bar{R}.m \sqsubseteq m' \land X \sqsubseteq X' \Rightarrow \sqcup_{i \leq \omega} T(m)^i(X) \sqsubseteq \sqcup_{i \leq \omega} T(m')^i(X') \)

(3) \( \forall X, X' : \bar{R}.X \sqsubseteq X' \Rightarrow U(X) \sqsubseteq U(X') \)

(4) \( \forall X, X' : \bar{R}.X \sqsubseteq X' \Rightarrow \sqcup_{i \leq \omega} U^i(X) \sqsubseteq \sqcup_{i \leq \omega} U^i(X') \)

**Proof.**

(1) By cases on whether \( \bar{\ell} \in L_A \) or \( \bar{\ell} \in L_F \):

   - Case \( \bar{\ell} \in L_F \): \( T(m)(X)[\bar{\ell}] = I_T(X)[\bar{\ell}] \sqsubseteq I_T(X')[\bar{\ell}] = T(m')(X')[\bar{\ell}] \) by the monotonicity of \( I_T \) (by Lemma 18, proved below).

   - Case \( \bar{\ell} \in L_A \): \( T(m)(X)[\bar{\ell}] = m[\bar{\ell}] \sqsubseteq m'[\bar{\ell}] = T(m')(X')[\bar{\ell}] \) by assumption.

(2) By induction, using the monotonicity of \( T \) shown above it can be shown that \( \forall \ell < \omega. SF(m)^i(X) \sqsubseteq SF(m')^i(X') \), whence the result follows from the definition of least upper bounds.

(3) Immediate from the fact that \( \widetilde{F} \) is monotone.

(4) By similar proof to (2) above.

\[ \square \]

**Theorem 5.** \( M \) is monotone.

**Proof.** Immediate from parts 2 and 4 of Lemma 8 and the monotonicity of \( \bar{T} \) and \( \bar{\tau} \).

\[ \square \]

We next define two ordinal termed sequences \( C^\delta \in M^\delta \) and \( M^\delta \), where \( \beta \) is an ordinal defined as in [Cousot and Cousot 1979a], via transfinite recursion as:

\[
\begin{align*}
C^0 &= \bot_R^R \\
M^0 &= \bot_R \\
C^{\delta+1} &= C(C^\delta) \\
M^{\delta+1} &= M(M^\delta) \\
C^\lambda &= \bigsqcup_{\beta < \lambda} C^\beta \\
M^\lambda &= \bigsqcup_{\beta < \lambda} M^\beta
\end{align*}
\]

As both \( M \) and \( C \) are monotone functions over complete lattices, then by Corollary 3.3 of [Cousot and Cousot 1979a], the limit of \( C^\delta \), \( C^\tau \), exists and is the least fixpoint of \( C \) and the limit of \( M^\delta \) 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 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 \( \xi \) between \([0, \ldots, |L|]\) and \( L \). We can define an indexed representation of an element of \( R \), where \( X_i \) corresponds to the mostly-concrete or abstract state at \( \xi(i) \). We

\[ \text{Proceedings of the ACM on Programming Languages, Vol. 1, No. CONF, Article 1. Publication date: January 2018.} \]
We do not define the value of $S$ 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 $\xi(i) \in L_A$, define:

$$F_i(I, E, D) = \tilde{F}(\text{inj}(\eta(I, E)))[\xi(i)]$$

$$\eta(I, E) = \left\{ \lambda \ell^\omega_a : L_A.1_{\xi^{-1}(\ell_a)}, \lambda \ell^\omega_f : L_F.1_{\xi^{-1}(\ell_f)} \right\}$$

$\eta$ transforms the indexed representation into the product-of-maps representation expected by $\text{inj}$.

When constructing the abstract state map it uses the values from $I$, and those from $E$ to construct the mostly-concrete map.

Before proving Theorem 6 we need the following technical lemmas.

C.1 Technical Lemmas

Before proving Theorem 6 we need the following technical lemmas.

Lemma 9. The function $T(m)(X)$ is $\omega$-upper-continuous in its second argument.
Proof. \[
\begin{align*}
\prod_{n} T(m)(x_n) &= \prod_{n} I_T(x_n) \uparrow L_F \sqcup m \downarrow L_A \\
&= m \downarrow L_A \sqcup \prod_{n} I_T(x_n) \downarrow L_F \\
&= m \downarrow L_A \sqcup I_T(\prod_{n} x_n) \downarrow L_F \text{(by the } \omega\text{-upper-continuity of } I_T, \text{ see Appendix C.1.1)} \\
&= T(m)(\prod_{n} x_n)
\end{align*}
\]
\[\square\]

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

Proof. Follows immediately from Lemma 9 and Kleene’s fixpoint theorem. \[\square\]

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

1. The (potentially infinite) join of a set \(S\) of pre-fixed points of a monotone function \(F\) is itself a pre-fixed point of \(F\).
2. For a monotone function \(g : L \to L\) on a complete lattice \(L\), the ordinal termed sequence defined as:

   \[
   \delta^0 = \bot_L \\
   \delta^{\delta+1} = g(\delta^\delta) \\
   \delta^\lambda = \bigcup_{\alpha < \lambda} \delta^\alpha
   \]

   is increasing.

Next define the family of sequences \(\mathcal{T}_m^\delta, \delta \in \text{Ord}\) via transfinite recursion as:

\[
\begin{align*}
\mathcal{T}_m^0 &= \bot_R \\
\mathcal{T}_m^{\delta+1} &= T(m)(\mathcal{T}_m^\delta) \\
\mathcal{T}_m^\lambda &= \bigcup_{\alpha < \lambda} \mathcal{T}_m^\alpha
\end{align*}
\]

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 \(\mathcal{T}_m^\omega\).

Lemma 10. \(\forall m, \delta \in \text{Ord}. \mathcal{T}_m^\delta \subseteq \mathcal{T}_m^\omega\)

Proof. Immediate from the definition of \(\mathcal{T}_m^\delta\) and by the fact that \(\mathcal{T}_m^\omega\) is the least fixpoint of \(T(m)\). \[\square\]

C.1.1 Continuity of \(I_T\). We now demonstrate the \(\omega\)-upper-continuity of \(I_T\) on increasing chains of \(R\).

Lemma 11. For any increasing chain \(x_j : \varphi(V_f)^T, v \in \bigsqcup_i x_i \Rightarrow \exists k. v \in x_k\)

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 \(\varphi(V_f)^T\) 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 \(x_j\). \[\square\]

Corollary 11.1. For an increasing chain of products \(c_i \equiv \langle x_1, \ldots, x_n \rangle : \varphi(V_f)^T \times \ldots \times \varphi(V_f)^T\), if there exists some \(\langle v_1, \ldots, v_n \rangle\) 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 \( \nu_j \in \pi_j(c_{k_j}) \). Let \( m \) be the max over these \( k_j \). As the chain \( c_i \) is increasing, if \( \nu_j \in \pi_j(c_{k_j}) \), then necessarily \( \nu_j \in \pi_j(c_m) \).

\[ \]  

Lemma 12. \( \lceil \text{fop} \rceil \) is \( \omega \)-upper-continuous for all increasing chains \( \arg_i : \varphi(V_f)^\top \times \ldots \times \varphi(V_f)^\top \)

Proof. Consider the case where, for some \( j \) and \( k \), \( \pi_j(\arg_k) = \top_F \). Then \( \lceil \text{fop} \rceil(\arg_k) = \top_F \), and further \( \pi_j(\arg_k) = \top_F \) whence \( \lceil \text{fop} \rceil(\arg_k) = \top_F \).

Consider now the case where the chain contains no \( \top_F \) values. Then, by the definition of \( \lceil \text{fop} \rceil \) and \( \lceil \text{fop} \rceil \), the sequence \( \lceil \text{fop} \rceil(\arg_k) \) will not contain \( \top_F \), and thus \( \lfloor \lfloor \arg_k \rceil \rceil \lceil \text{fop} \rceil(\arg_k) \) will not equal \( \top_F \), and similarly \( \lceil \text{fop} \rceil(\arg_k) \).

It therefore suffices to show that \( \nu \in \lfloor \lfloor \arg_k \rceil \rceil \lceil \text{fop} \rceil(\arg_k) \iff \nu \in \lceil \text{fop} \rceil(\arg_k) \).

Case \( \Rightarrow \): By the definition of lub and \( \lceil \text{fop} \rceil \), \( \exists \nu. \nu \in \lceil \text{fop} \rceil(\arg_k) \). But then \( \forall j. \nu_j(\arg_k) \in \pi_j(\arg_k) \) and thus by definition of \( \lceil \text{fop} \rceil \), \( \nu \in \lceil \text{fop} \rceil(\arg_k) \).

Case \( \Leftarrow \): If \( \nu \in \lceil \text{fop} \rceil(\arg_k) \), then there must exist some set of values \( \nu_j \) and some \( E \in \mathcal{E} \) such that \( \forall j. \nu_j \in \pi_j(\arg_k) \) and \( \lceil \text{fop} \rceil(E, \nu_0, \ldots, \nu_n) = \langle \nu, \_ \rangle \). By Corollary 11.1, there must be some \( k \), such that \( \forall j. \nu_j \in \pi_k(\arg_k) \), whence \( \nu \in \lceil \text{fop} \rceil(\arg_k) \subseteq \lceil \text{fop} \rceil(\arg_k) \).

Lemma 13. \( \lceil \text{fop} \rceil \) is pointwise monotone.

Proof. Consider the case of some \( \lceil \text{fop} \rceil(\nu_0, \ldots, \nu_n) \) where one of \( \nu_i = \top_F \). Then, \( \lceil \text{fop} \rceil(\nu_0, \ldots, \nu_n) = \top_F \), and for any \( \nu_j, \nu_j \subseteq \nu_j = \top_F \), whence \( \lceil \text{fop} \rceil(\nu_0, \ldots, \nu_n) = \top_F \), for any values \( \nu_j \) pointwise greater than \( \nu_i \). Next, consider the case of some arguments \( \nu_i \subseteq \nu_i \), \( \forall j. \nu_j \neq \top_F \). Then \( \lceil \text{fop} \rceil(\nu_0, \ldots, \nu_n) \neq \top_F \). If any of \( \nu_j \) is \( \top_F \), then monotonicity holds trivially. Finally, consider when, for each \( j \), \( \nu_j \subseteq \nu_j \). Then by the definition of \( \lceil \text{fop} \rceil \), it is immediate that \( \lceil \text{fop} \rceil(\nu_0, \ldots, \nu_n) \subseteq \lceil \text{fop} \rceil(\nu_0, \ldots, \nu_n) \).

Lemma 14. If \( \nu_1 : S \) is an increasing sequence, and if for such sequences \( \lfloor \lfloor \nu_1 \rceil \rceil \nu_1 = f(\lfloor \lfloor \nu_1 \rceil \rceil \nu_1) \), then:

\[ \forall x. Q = \lfloor \lfloor \nu_1 \rceil \rceil \nu_1 \leftarrow f(\lfloor \lfloor \nu_1 \rceil \rceil \nu_1) \rceil = R \]

Proof. \( Q[x] = \lfloor \lfloor \nu_1 \rceil \rceil \nu_1 \leftarrow f(\lfloor \lfloor \nu_1 \rceil \rceil \nu_1) \rceil = R[\nu] \) and \( Q[y \neq x] = \lfloor \lfloor \nu_1 \rceil \rceil \nu_1 \leftarrow f(\lfloor \lfloor \nu_1 \rceil \rceil \nu_1) \rceil = R[\nu] \).

Lemma 15. For an increasing sequence \( \nu : S \), \( \forall \ell. \lfloor \lfloor \nu \rceil \rceil \text{step} \cdot (\nu, \ell) = \text{step} \cdot (\lfloor \lfloor \nu \rceil \rceil \nu, \ell) \).

Proof. By cases on the branch taken in \( \text{step} \):

Branch (5): Trivial
Branches (6, 7, 9): Follows from the Lemma 14 and that \( \lambda_. \cdot \text{step} \cdot \lambda_.(\lfloor \lfloor \nu \rceil \rceil \nu) \), and \( \lambda s. s[y] \) are trivially continuous on increasing sequences.

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

Lemma 16. \( \text{step} \cdot \) is monotone.

Proof. Immediate from the monotonicity of \( \lceil \text{fop} \rceil \) (Lemma 13), and the definition of \( \text{step} \cdot \).

Lemma 17.

\[ \forall \nu, \zeta, \nu^* \Rightarrow \ell^0. \overline{\text{F}^\top}(\nu, \nu^* \Rightarrow \ell^0) \land \zeta \subseteq \zeta' \Rightarrow \overline{\text{F}^\top}(\zeta', \nu^* \Rightarrow \ell^0) \]

(2) For an increasing sequence \( r_i, \forall p^* \leadsto \ell^0. \overrightarrow{F}T(\sqcup_i r_i, p^* \leadsto \ell^0) \Leftrightarrow \exists j. \overrightarrow{F}T(r_j, p^* \leadsto \ell^0)

**Proof.**

(1) Let \( \overrightarrow{s} = \overrightarrow{r}[p] \) and \( \overrightarrow{s}' = \overrightarrow{r}'[p] \). Consider by cases why \( \overrightarrow{F}T \) returned true:

**Branch 2** Then \( \overrightarrow{s}[x] \mathbin{\ll suspects=} \overrightarrow{s}'[y] \), thus either \( \overrightarrow{s}[x] = \top_F, \overrightarrow{s}'[y] = \top_F \), or \( v \in \overrightarrow{s}[x] \) and \( v' \in \overrightarrow{s}'[y] \) such that that \( v \mathbin{\ll suspects=} v' \). In the former two cases, \( \overrightarrow{s}[x] \mathbin{\ll suspects=} \overrightarrow{s}'[y] \). Consider the final case. Then by definition of \( \sqsubseteq \), \( \overrightarrow{s}[x] = \top_F \) or \( \overrightarrow{s}[x] \subseteq \overrightarrow{s}'[y] \) and similarly for \( \overrightarrow{s}[y] \) and \( \overrightarrow{s}'[y] \). In the case where either \( \overrightarrow{s}[x] \subseteq \overrightarrow{s}'[y] \), then \( \overrightarrow{s}[x] \mathbin{\ll suspects=} \overrightarrow{s}'[y] \) trivially returns true. Otherwise, we have \( v \in \overrightarrow{s}[x] \) and similarly for \( v' \) and \( \overrightarrow{s}'[y] \), and thus by definition \( \overrightarrow{s}[x] \mathbin{\ll suspects=} \overrightarrow{s}'[y] \) holds.

**Branch 3** By similar reasoning to the above, but on the definition of \( \ll suspects \).

**Branch 4** \( \overrightarrow{F}T \) must always return true.

(2) For the forward implication, consider by cases why \( \overrightarrow{F}T \) returned true:

**Branch 2** By similar reasoning to the above, and without loss of generality, let us consider the case where \( (\sqcup_i r_i)[p][x] = \top_F \). Then, there must exist some \( j \) such that \( r_j[p][x] = \top_F \), and thus \( \overrightarrow{F}T(r_j, p^* \leadsto \ell^0) \) trivially returns true. The case when \( (\sqcup_i r_i)[p][y] = \top_F \) follows from similar reasoning. Finally, let us consider the case where \( \forall v \in (\sqcup_i r_i)[p][x], v' \in (\sqcup_i r_i)[p][y] \) and \( v \mathbin{\ll suspects=} v' \). 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 \( r_j[p][x] \mathbin{\ll suspects=} r_j[p][y] \) and thus \( \overrightarrow{F}T(r_j, p^* \leadsto \ell^0) \) must return true.

**Branch 3** By similar reasoning to the above.

**Branch 4** \( \forall j. \overrightarrow{F}T(r_j, p, \ell) \) must be true.

The backward implication follows immediately from part one of the lemma and from the fact that \( \forall j. r_j \subseteq \sqcup_i r_i \).

\[\Box\]

**Lemma 18.** \( I_\tau \) is monotone

**Proof.** Consider some \( \overrightarrow{r} \subseteq \overrightarrow{r}' \), and some arbitrary \( \ell^0 \leadsto \ell^* \):

\[
I_\tau(\overrightarrow{r})[\ell^0 \leadsto \ell^*] = \text{step}^\tau \left( \bigsqcup_{p \in \text{pred}(\ell)} \overrightarrow{r}[p^* \leadsto \ell^0], \ell \right)
\]

\[
\sqsubseteq \text{step}^\tau \left( \bigsqcup_{p \in \text{pred}(\ell)} \overrightarrow{r}'[p^* \leadsto \ell^0], \ell \right) \quad \text{(Lemma 16)}
\]

\[
= I_\tau(\overrightarrow{r}')[\ell]
\]

We have ignored the initialization terms, as it is constant and by the monotonicity of least upper bounds as \( \text{step}^\tau \) is monotone monotonicity is preserved.

Next, consider some \( p^* \leadsto \ell^0 \), and whether \( \overrightarrow{F}T(\overrightarrow{r}, p^* \leadsto \ell^0) \) is true. If so, then Lemma 17 implies that \( \overrightarrow{F}T(\overrightarrow{r}, p^* \leadsto \ell^0) \) must also be true, which from the fact that \( \overrightarrow{r}[p^0 \leadsto p^*] \subseteq \overrightarrow{r}'[p^0 \leadsto p^*] \), gives us that \( I_\tau(\overrightarrow{r})[p^* \leadsto \ell^0] = I_\tau(\overrightarrow{r}')[p^* \leadsto \ell^0] \). Otherwise, \( I_\tau(\overrightarrow{r})[p^* \leadsto \ell^0] = \bot \sqsubseteq I_\tau(\overrightarrow{r}')[p^* \leadsto \ell^0] \). \( \Box \)

We can now prove the continuity of \( I_\tau \).

**Theorem 8.** For an increasing sequence \( r_i : \overrightarrow{R}, I_\tau \) is continuous.

**Proof.** We first note that the initialization term of \( I_\tau \) at the start label \( s \) is a constant term and can be easily factored out.
Due to the pointwise nature of $\widetilde{R}$ it suffices to consider an arbitrary $\ell^0 \leadsto \ell^*$ and $p^* \leadsto \ell^0$.

\[
\bigcup_i I_\tau(r_i)[\ell^0 \leadsto \ell^*] = \bigcup_i \text{step}^\top\left( \bigcup_{\rho \in \text{pred}(\ell)} r_i[p^* \leadsto \ell^0], \ell \right)
\]

\[
= \text{step}^\top\left( \bigcup_{\rho \in \text{pred}(\ell)} \bigcup_i r_i[p^* \leadsto \ell^0], \ell \right)
\]

\[
= \text{step}^\top\left( \bigcup_{\rho \in \text{pred}(\ell)} \bigcup_i (r_i)[p^* \leadsto \ell^0], \ell \right)
\]

\[
= I_\tau\left( \bigcup_i r_i[\ell] \right)
\]

The second equality follows from $\sqcup_{\rho \in \text{pred}(\ell)} r_i[p^* \leadsto \ell^0]$ being and increasing sequence and Lemma 15.

Next, let us consider some $p^* \leadsto \ell^0$. Consider the case where $\widetilde{F}\tau (\sqcup_i r_i, p^* \leadsto \ell^0)$ is false, thus $I_\tau(\sqcup_i r_i)[p^* \leadsto \ell^0] = \perp$. By Lemma 17 (2), we have that $\forall j. \widetilde{F}\tau (r_j, p^* \leadsto \ell^0)$ is also false, and thus $\sqcup_i I_\tau(r_i)[p^* \leadsto \ell^0] = \sqcup_i \perp = I_\tau(\sqcup_i r_i)[p^* \leadsto \ell^0]$.

Finally, consider the case where $\widetilde{F}\tau (\sqcup_i r_i, p^* \leadsto \ell^0)$ is true, whence $I_\tau(\sqcup_i r_i)[p^* \leadsto \ell^0] = \sqcup_i r_i[\ell^0 \leadsto p^*]$. By Lemma 17, there exists some $j$ such that $\forall k \geq j. \widetilde{F}\tau (r_k, p^* \leadsto \ell^0)$ is true. As $I_\tau$ is monotone, we have that $I_\tau(r_i)$ is an increasing sequence, and thus

\[
\left( \bigcup_i I_\tau(r_i) \right)[p^* \leadsto \ell^0] = \left( \bigcup_{k \geq j} I_\tau(r_k) \right)[p^* \leadsto \ell^0] (I_\tau(r_i) \text{ is increasing})
\]

\[
= \bigcup_{k \geq j} r_k[\ell^0 \leadsto p^*] \text{ (As } \forall k \geq j. \widetilde{F}\tau (r_k, p^* \leadsto \ell^0) \text{ must be true})
\]

\[
= \bigcup_i r_i[\ell^0 \leadsto p^*] (r_i \text{ is increasing})
\]

\[
= I_\tau\left( \bigcup_i r_i[p^* \leadsto \ell^0] \right)
\]

\[\Box\]

C.2 $\mathcal{E}^\ell \subseteq \mathcal{Y}^\ell$

We first need the following lemmas.

**Lemma 19.** $\forall \hat{\ell} \in \mathcal{L}_A, X \subseteq X' \in \hat{\mathcal{R}}.\overline{\tau} \circ \overline{\text{inj}}(X)[\hat{\ell}] \subseteq \bigcup_{i < \omega} U_i(\overline{\text{inj}}(X'))[\hat{\ell}]$

**Proof.**

\[
\overline{\tau}(\overline{\text{inj}}(X))[\hat{\ell}] \subseteq \overline{\tau}(\overline{\text{inj}}(X'))[\hat{\ell}] = U(\overline{\text{inj}}(X'))[\hat{\ell}] \subseteq \bigcup_{i < \omega} U_i(\overline{\text{inj}}(X'))[\hat{\ell}]
\]

\[\Box\]

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

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

At a limit ordinal, we have that $\mathcal{Z}_m^\delta[\ell] = \bigcup_{\lambda < \delta} \mathcal{Z}_m^\delta[\ell] = \perp \sqcup \bigcup_{\lambda < \delta} \mathcal{Z}_m^{\alpha + 1}[\ell] = \bigcup_{\lambda < \delta} \mathcal{Z}_m^{\alpha + 1}[\ell] = \bigcup_{\lambda < \delta} m[\ell] = m[\ell]$ \[\Box\]

**Lemma 21.** $\forall \delta \in \mu, m, \mathcal{C}^\delta|_{\mathcal{L}_A} \subseteq m|_{\mathcal{L}_A} \Rightarrow \mathcal{C}^{\delta + 1}_{\mathcal{L}_F} \subseteq \mathcal{Z}_{G\delta + 2}_{\mathcal{L}_F}$ where $G = \overline{\text{inj}}(m)|_{\mathcal{L}_A}$

**Proof.** By transfinite induction on $\delta$: 

Case $\delta = 0$: It suffices to show that for some $\ell \in \mathcal{L}_F, \mathcal{C}^1[\ell] = C(\mathcal{L}_F)[\ell] = I_T(\text{inj}(\mathcal{L}_F))[\ell] \subseteq \tau_G^2[\ell] = I_T(\tau_G^1)[\ell]$. This follows from the monotonicity of $I_T$ if we show that $\text{inj}(\mathcal{L}_F) \subseteq \tau_G^1$.

At points in $\mathcal{L}_F$ this is immediate, as $\text{inj}(\mathcal{L}_F)[\ell] = \bot \subseteq \tau_G^1[\ell]$. At some point $\tilde{\ell}' \in \mathcal{L}_A$, observe that $\text{inj}(\mathcal{L}_F)[\tilde{\ell}'] = \tau(\bot) = \tau$.

Case $\delta + 1$: It suffices to show that for some $\ell \in \mathcal{L}_F$ we have $\mathcal{C}^{\delta+2}[\ell] = I_T(\text{inj}(\mathcal{C}^{\delta+1}))[\ell] \subseteq \tau_G^{\delta+2}[\ell] = I_T(\tau_G^1)[\ell]$, which holds if we can prove that $\text{inj}(\mathcal{C}^{\delta+1}) \subseteq \tau_G^{\delta+2}$.

As the sequence $\mathcal{C}$ is increasing, we have that $\mathcal{C}^{\delta} \subseteq \mathcal{C}^{\delta+1}$, whence by transitivity we have that $\mathcal{C}^{\delta} \subseteq \mathcal{C}^{\delta+1}$, thus by the inductive hypothesis, we have $\mathcal{C}^{\delta+1} \subseteq \mathcal{C}^{\delta+1}_L \subseteq \mathcal{L}_A$.

To establish the required inequality, we proceed by subcases on the partition of some arbitrary $\ell'$.

Subcase $\tilde{\ell}' \in \mathcal{L}_F$: $\text{inj}(\mathcal{C}^{\delta+1})[\tilde{\ell}'] = \mathcal{C}^{\delta+1}[\tilde{\ell}'] \subseteq \tau_G^{\delta+2}[\ell']$. By the inductive hypothesis, we have $\tau_G^{\delta+2}[\ell'] \subseteq \tau_G^{\delta+2}[\ell]$, whence the result holds from the transitivity of $\tau$.

Subcase $\tilde{\ell}' \in \mathcal{L}_A$: $\text{inj}(\mathcal{C}^{\delta+1})[\tilde{\ell}'] = \tau(\mathcal{C}^{\delta+1}[\tilde{\ell}]) \subseteq \tau(m[\tilde{\ell}]) = \text{inj}(m)[\tilde{\ell}'] = G[\tilde{\ell}'] = \tau_G^{\delta+2}[\ell']$.

Expanding definitions, we must therefore show that $\text{inj}(\mathcal{C}^{\delta+1})[\ell] \subseteq \tau_G^{\delta+2}[\ell]$. Whence the final equality follows from Lemma 20.

Expanding definitions, we must therefore show that $\text{inj}(\mathcal{C}^{\delta+1})[\ell] \subseteq \tau_G^{\delta+2}[\ell]$. Whence the final equality follows from Lemma 20.

Now, to prove $\mathcal{C}^{\ell} \subseteq \mathcal{M}^{\ell}$, we prove the following:

**Lemma 22.** $\forall \delta \in \mu. \mathcal{C}^{\delta} \subseteq \mathcal{M}^{\delta}$

**Proof.** By transfinite induction on $\delta$.

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

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

We first need the following lemma:

\[ C(\mathcal{A})[\ell] = \bar{\nu}(\mathcal{A})[\ell] \subseteq \bigcup_{i < \omega} U^i(\mathcal{A})[\ell] \]

which follows from the induction hypothesis and Lemma 19.

Subcase \( \ell \in L_\mathcal{A} \): We must show that:

\[ C(\mathcal{A})[\ell] = \bar{\nu}(\mathcal{A})[\ell] \subseteq \bigcup_{i < \omega} U^i(\mathcal{A})[\ell] \]

Proof. Immediate from Lemma 22.

Theorem 9. \( \mathcal{C}^\varepsilon \subseteq \mathcal{M}^\varepsilon \)

Proof. Immediate from Lemma 22.

C.3 \( \mathcal{M}^\varepsilon \subseteq \mathcal{C}^\varepsilon \)

We first need the following lemma:

Lemma 23. \( \forall i \in \mathbb{N}, X : \bar{\nu}(X) \subseteq \mathcal{C}^\varepsilon \Rightarrow T(\bar{\nu}(X)[\mathcal{A}]) \subseteq \bar{\nu}(\mathcal{C}) \)

Proof. By induction on \( i \).

Case \( i = 0 \): Trivial

Case \( i + 1 \): We assume \( T(\bar{\nu}(X)[\mathcal{A}]) \subseteq \bar{\nu}(\mathcal{C}) \) and show that \( T(\bar{\nu}(X)[\mathcal{A}]) \subseteq \bar{\nu}(\mathcal{C}) \).

Proof. By induction on \( i \).

Case \( i = 0 \): Trivial, by the monotonicity of \( \bar{\nu} \).

Case \( i + 1 \): By a similar argument to that made in Lemma 23, except on the monotonicity of \( \bar{\nu} \).

We next prove the following theorem:

Theorem 10. \( \forall \delta \in \mu, \mathcal{M}^\delta \subseteq \mathcal{C}^\varepsilon \)

Proof. By transfinite induction on \( \delta \).

Case \( \delta = 0 \): Trivial as \( \mathcal{M}^0 = \bot \).

Case \( \delta + 1 \): It suffices to show that \( \forall \ell, \mathcal{M}^{\delta+1}[\ell] \subseteq \mathcal{C}^\varepsilon[\ell] \). We proceed by the partition of \( \ell \):

Subcase \( \ell \in L_\mathcal{A} \): \( \mathcal{M}^{\delta+1}[\ell] = [\bigcup_{i < \omega} U^i(\bar{\nu}(\mathcal{M}^\delta))][\ell] \subseteq \bar{\nu}(\mathcal{C}^\varepsilon)[\ell] = \mathcal{C}^\varepsilon[\ell] \), where the inequality holds from using the inductive hypothesis and Lemma 24 to show that \( \bar{\nu}(\mathcal{C}^\varepsilon)[\ell] \) is an upper bound and thus greater than the least upper bound.

Subcase \( \ell \in L_F \): By a similar argument to the \( L_\mathcal{A} \) case, but using Lemma 23.

Case $\delta = \lambda$: By the transfinite hypothesis, $\forall \alpha < \lambda. \mathcal{M}^\alpha \subseteq \mathcal{C}^\epsilon$, thus: $\bigcup_{\alpha < \lambda} \mathcal{M}^\alpha \subseteq \mathcal{C}^\epsilon$. □

Corollary 10.1. $\mathcal{M}^\epsilon \subseteq \mathcal{C}^\epsilon$

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

C.4 Proof of Theorem 6

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

C.5 Fixpoints in the Abstract Interpreter

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

Lemma 25. $\forall \delta, k. U^k(\mathcal{M}^{\delta}) \subseteq \mathcal{M}^{\delta + 1}$

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

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

$$U^k(\mathcal{M}^{\delta})[\ell] = \bigcup_i U^i(\mathcal{M}^{\delta})[\ell] = \mathcal{M}^{\delta + 1}[\ell] = \text{inj}(\mathcal{M}^{\delta + 1})[\ell]$$

□

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

$$PF \equiv \lambda(\overline{m}, \overline{m}) : \overline{m} \times U(\text{inj}(\langle \overline{m}, \overline{m} \rangle)))|_{L_A}$$

Lemma 26. $\forall \delta. \mathcal{M}^{\delta} \subseteq PF(\mathcal{M}^{\delta})$

Proof. By transfinite induction.

Case $\delta = 0$: Trivial.

Case $\delta + 1$: The inequality is immediate at all points in $L_F$. We must therefore establish the inequality at some $\ell \in L_A$.

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

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

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

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

□

Theorem 11. $\forall \delta \in \text{Ord}$, the sequence $U^i(\mathcal{M}^{\delta})$ is increasing.
We first prove the soundness of the widening version of subfixpoint iteration. Define a sequence \( M^\delta_\nu \) as:

\[
M^0_\nu = \perp \quad M^{\delta+1}_\nu = M_\nu(M^\delta_\nu) \quad M^\delta_\nu = \bigsqcup_{\beta < \lambda} M^\beta_\nu
\]

We have shown that this sequence over-approximates the sequence \( M^\delta \) as follows:

**Theorem 12.** \( \forall \delta. M^\delta \subseteq M^\delta_\nu \)

**Lemma 27.** \( \forall i \in \mathbb{N}, m, m' : \exists \perp. m \sqsubseteq m' \Rightarrow T(m)^i(\perp) \subseteq T_\nu(T(m)^i(\perp)) \)

**Proof.** By induction on \( i \).

**Case** \( i = 0 \): Trivial, as the partial order is reflexive.

**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 \( T(m)^i(\perp) \subseteq T_\nu(T(m)^i(\perp)) \) whence from the monotonicity of \( I_\tau \) we therefore have \( I_\tau(T(m)^i(\perp)) \subseteq I_\tau(T_\nu(T(m)^i(\perp))) \), and thus: \( T(m)^{i+1}(\perp)[\vec{\ell}] = \)
We first show that \( n \in \mathbb{N}, m, m' : \mathcal{R}, m \subseteq m' \Rightarrow U^i(m) \subseteq U^i_{\mathcal{V}}(m') \)

**Proof.** By symmetric reasoning to that in **Lemma 27**, except using the monotonicity of \( \mathcal{F} \) and that \( \mathcal{V} \) as an upper bound operator. \( \square \)

**Proof of Theorem 12.** By transfinite induction. In the inductive step, the inductive hypothesis, monotonicity of \( \mathcal{I} \), and \( \mathcal{J} \), and Lemmas 27 and 28 give that the suffixpoint terms of \( \mathcal{V}^{\delta+1} \) are smaller than those in \( \mathcal{V}^{\delta+1} \) giving the desired result. The limit case follows from the definition of least upper bounds. \( \square \)

As a direct corollary of this theorem, we have: \( \text{lfp} C = \text{lfp} M = \mathcal{V}_x \subseteq \mathcal{V}_x \). If we show that if the sequence \( \mathcal{V}^{\delta} \) converges in a finite number of \( k \) steps to a fixed-point of \( \mathcal{V} \), we will then have: \( \text{lfp} C = \text{lfp} M = \mathcal{V}_x \subseteq \mathcal{V}_x \), i.e., \( \alpha_F(F) \subseteq \mathcal{V}_x \).

### D.2 Termination

We first show that \( \forall i \in \mathbb{N} \), the sequence \( \mathcal{V}_x \) is increasing.

**Lemma 29.** \( \forall j, \mathcal{V}_x \subseteq \mathcal{V}_{x+1} \)

**Proof.** Consider an arbitrary \( i \). We establish the inequality pointwise:

1. **Case \( \ell \in L_\mathcal{F} \):** \( \mathcal{V}_x(\ell) = \mathcal{V}_x(\ell) \subseteq \mathcal{V}_x(\mathcal{F}_x(\ell)) = \mathcal{V}_x(\mathcal{F}_x)_1(\mathcal{L}_\mathcal{A})_1(\ell) \subseteq \mathcal{V}_x(\ell) \)

2. **Case \( \ell \in L_\mathcal{A} \):** \( \mathcal{V}_x(\ell) = \mathcal{V}_x(\mathcal{L}_\mathcal{A}(\ell)) \subseteq \mathcal{V}_x(\mathcal{F}_x(\ell)) \)

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

\[ \Psi_{i,j} = \mathcal{V}_x(\mathcal{F}(\mathcal{V}_x)) \]

We need the following lemma.

**Lemma 30.**

1. \( \forall n, \mathcal{V}_x(\mathcal{F}(\mathcal{V}_x)) \subseteq n, \mathcal{V}_x(\mathcal{V}_x) \)

2. The sequence \( n, \mathcal{V}_x(\mathcal{F}(\mathcal{V}_x)) \) is increasing.

3. For all \( n \) and \( \ell \in L_\mathcal{A} \), if all sequences \( \Psi_{i < n, j} \) are increasing and converges in a finite number of steps, then \( \mathcal{V}_x(\mathcal{V}_x)(\ell) = \mathcal{L}_\mathcal{V}(\ell) \).

4. For all \( n \), if all sequences \( \Psi_{i < n, j} \) are increasing and converge in a finite number of steps, then the sequence \( \mathcal{V}_x(\mathcal{F}(\mathcal{V}_x)) \) is increasing.

**Proof.**

1. At \( \ell \in L_\mathcal{F} \), we have that \( \mathcal{V}_x(\ell) = \mathcal{V}_x(\ell) \subseteq \mathcal{V}_x(\mathcal{F}(\mathcal{V}_x))(\ell) \) from **Lemma 29**. At \( \ell \in L_\mathcal{A} \), we have: \( \mathcal{V}_x(\ell) \subseteq \mathcal{V}_x(\ell) \) for \( \ell \in L_\mathcal{F} \).

2. By straightforward induction on \( j \), using the fact that \( \mathcal{F} \) is monotone and that \( \mathcal{V} \) is an upper bound operator. The base case holds because at \( \ell \in L_\mathcal{A} \), \( \mathcal{V}_x(\ell) = \mathcal{L}_\mathcal{V}(\ell) \) and for \( \ell \in L_\mathcal{F} \).

3. By induction on \( n \).

Case $n = 0$: Then $\widehat{\text{inj}}(\mathcal{M}_n^0)[\vec{t}] = \bot_{\mathcal{F}}[\vec{t}] = \bot_{\mathcal{S}}$, trivially giving the desired result.

Case $n + 1$: Then $\widehat{\text{inj}}(\mathcal{M}_{n+1}^0)[\vec{t}] = \bigsqcup_{i < \omega} U_V^i (\widehat{\text{inj}}(\mathcal{M}_n^0))[\vec{t}]$. By hypothesis, the sequence $U_V^i (\widehat{\text{inj}}(\mathcal{M}_n^0))$ is increasing and converges in a finite number of steps, whence $\exists p . \bigsqcup_{i < \omega} U_V^i (\widehat{\text{inj}}(\mathcal{M}_n^0))[\vec{t}] = U_V^p (\widehat{\text{inj}}(\mathcal{M}_n^0))[\vec{t}]$. If $p = 0$, then the inductive hypothesis gives us the desired result (it is immediate that if the sequences $\mathcal{P}_{(m < n+1,j)}$ are terminating increasing sequences, then the sequences $\mathcal{P}_{(m < n,j)}$ 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 $\widehat{\text{inj}}(\mathcal{M}_n^0) \subseteq U_V (\widehat{\text{inj}}(\mathcal{M}_n^0))$. By cases:

Subcase $\vec{t} \in \mathcal{W}_A$: $U_V (\widehat{\text{inj}}(\mathcal{M}_n^0))[\vec{t}] = F_V (\widehat{\text{inj}}(\mathcal{M}_n^0))[\vec{t}] = \widehat{\text{inj}}(\mathcal{M}_{n+1}^0)[\vec{t}] = \hat{\mathcal{F}} (\widehat{\text{inj}}(\mathcal{M}_{n+1}^0))[\vec{t}]$, and we have the required inequality from $\hat{\mathcal{V}}$ being an upper bound operator.

Subcase $\vec{t} \in \mathcal{L}_A \land \vec{t} \notin \mathcal{W}_A$: We must show that $\widehat{\text{inj}}(\mathcal{M}_n^0)[\vec{t}] \subseteq U_V (\widehat{\text{inj}}(\mathcal{M}_n^0))[\vec{t}] = \hat{\mathcal{F}} (\widehat{\text{inj}}(\mathcal{M}_{n+1}^0))[\vec{t}]$. By hypothesis, the sequences $\mathcal{P}_{(m < n,j)}$ are increasing. As $m \leq n$ implies $m < n + 1$, we can satisfy the hypothesis of part (3) for $\widehat{\text{inj}}(\mathcal{M}_n^0)$. Suppose then that $\widehat{\text{inj}}(\mathcal{M}_n^0)[\vec{t}] = \bot$. Then we trivially have $\widehat{\text{inj}}(\mathcal{M}_n^0)[\vec{t}] = \bot \subseteq \hat{\mathcal{F}} (\widehat{\text{inj}}(\mathcal{M}_{n+1}^0))[\vec{t}]$. Next, suppose that there exists some $k < n + 1$, $j > 0$ such that $\widehat{\text{inj}}(\mathcal{M}_{n+1}^0)[\vec{t}] = U_V^j (\widehat{\text{inj}}(\mathcal{M}_k^0))[\vec{t}]$. Expanding the definition of $U_V$, we must then show that $U_V^j (\widehat{\text{inj}}(\mathcal{M}_k^0))[\vec{t}] = \hat{\mathcal{F}} (U_V^{j-1} (\widehat{\text{inj}}(\mathcal{M}_k^0)))[\vec{t}] \subseteq \hat{\mathcal{F}} (\widehat{\text{inj}}(\mathcal{M}_n^0))[\vec{t}]$. By part (1) and Lemma 29, we have that $U_V^{j-1} (\widehat{\text{inj}}(\mathcal{M}_k^0)) \subseteq \widehat{\text{inj}}(\mathcal{M}_{n+1}^0) \subseteq \widehat{\text{inj}}(\mathcal{M}_{n+1}^0)$, giving the desired result via monotonicity of $\hat{\mathcal{F}}$.

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

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

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

Theorem 13. $\forall n. \mathcal{P}_{(n,j)}$ 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 definition of widening sets. Consider an arbitrary $\vec{t} \in \mathcal{W}_A$, and the values computed at each element of the sequence $\mathcal{P}_{(0,j)}$.

$$\mathcal{P}_{(0,0)}[\vec{t}] = \widehat{\text{inj}}(\mathcal{M}_0^0)[\vec{t}] = \widehat{\text{inj}}(\bot_{\mathcal{F}})[\vec{t}] = \bot_{\mathcal{S}}$$

$$\mathcal{P}_{(0,i+1)}[\vec{t}] = U_V (U_V^i (\widehat{\text{inj}}(\bot_{\mathcal{F}})))[\vec{t}] = \hat{\mathcal{F}} (U_V^i (\widehat{\text{inj}}(\bot_{\mathcal{F}})))[\vec{t}] = U_V^i (\widehat{\text{inj}}(\bot_{\mathcal{F}}))[\vec{t}]$$

By the definition of widening operators, this sequence will converge in a finite number of steps (that the sequence $\bot_{\mathcal{S}} = \hat{\mathcal{F}} (\widehat{\text{inj}}(\bot_{\mathcal{F}}))$, the values computed from Lemma 30 part (2) and the monotonicity of $\hat{\mathcal{F}}$). Thus, all widening points will converge in a finite number of steps, implying that $\mathcal{P}_{(0,j)}$ converges.
**Case $n + 1$:** We assume that all sequences $\mathcal{P}_{(m < n + 1, j)}$ are increasing and converge in a finite number of steps. From Lemma 30 part (4), $\mathcal{P}_{(n + 1, j)}$ is increasing. To show that $\mathcal{P}_{(n + 1, j)}$ converges in a finite number of steps, we will again show that any arbitrary widening point $\bar{\ell}$ converges in a finite number of steps. Define a sequence finitization operator $[\underline{\mathcal{S}}] = [s_0, \ldots, s_{k-1}]$ where $s_i = \mathcal{S}_{i+1}$ and $k$ is defined as the smallest natural number such that:

$$
    k \geq 1 \quad \forall m \geq k, \mathcal{S}_m = \mathcal{S}_k
$$

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 converges. This operation is undefined on sequences that do not converge in finite steps. Define now the following sequence:

$$
    \mathcal{U} \equiv \mathcal{U}_{m < n + 1} \{[\mathcal{P}_{(m, j)}] \bar{\ell}] \Rightarrow \mathcal{P}_{(n + 1, j > 1)}[\bar{\ell}] \quad \text{where} \Rightarrow \text{denotes sequence concatenation. The} \ [\mathcal{P}_{(m, j)}] \bar{\ell}] \text{operation is well defined from}
$$

the induction hypothesis that all such sequences terminate. As a result, $\mathcal{U}$ has a finite prefix of values before reaching elements drawn from $\mathcal{P}_{(n + 1, j)}[\bar{\ell}]$. If we show that the sequence $\mathcal{U}$ converges in a finite number of steps, this will in turn imply that $\mathcal{P}_{(n + 1, j)}[\bar{\ell}]$ also converges in a finite number of steps. We now show that $\mathcal{U}_{i > 0} = \mathcal{U}_{i-1} \triangledown \mathcal{W}_i$ and $\mathcal{U}_0 = \mathcal{W}_0$, where $\mathcal{W}_i$ 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 $\mathcal{P}_{(n + 1, j)}[\bar{\ell}]$.

From the definition of $\mathcal{U}$ it is immediate that:

$$
    \forall i, \mathcal{U}_i \equiv \mathcal{P}_{(m, k)}[\bar{\ell}] = U^k \left( (\mathcal{W}_m) \mathcal{U} \right) \left( \bar{\ell} \right) \text{ for some } m \leq n + 1 \text{ and } k \geq 1 \quad (22)
$$

$$
    \forall i, k > 0, m \leq n + 1, \mathcal{U}_{i+1} \equiv U^{k+1} \left( (\mathcal{W}_m) \mathcal{U} \right) \left( \bar{\ell} \right) \Rightarrow \mathcal{U}_i \equiv U^k \left( (\mathcal{W}_m) \mathcal{U} \right) \left( \bar{\ell} \right) \quad (23)
$$

We will now show that for all $i > 0$, if $\mathcal{U}_{i>0}$ is defined as $\mathcal{P}_{(m, k)}[\bar{\ell}]$ for some $m \leq n + 1$ and $k \geq 1$, then $\mathcal{U}_i = \mathcal{U}_{i-1} \triangledown \mathcal{W}_i \mathcal{P}_{(m, k-1)}[\bar{\ell}]$.

Consider an arbitrary $i > 0$. Then $\mathcal{U}_i = U^k \left( (\mathcal{W}_m) \mathcal{U} \right) \left( \bar{\ell} \right)$, $m \leq n + 1$, $k > 0$ by Eq. (22) and the definition of $\mathcal{W}$. Suppose $k \neq 1$. Then $k = j + 1$, and

$$
    U^{j+1} \left( (\mathcal{W}_m) \mathcal{U} \right) \left( \bar{\ell} \right) = U^j \left( (\mathcal{W}_m) \mathcal{U} \right) \left( \bar{\ell} \right) \triangledown \mathcal{W}_i \left( (\mathcal{W}_m) \mathcal{U} \right) \left( \bar{\ell} \right) = \mathcal{U}_{i-1} \triangledown \mathcal{W}_i \mathcal{P}_{(m, k-1)}[\bar{\ell}]
$$

where the final equality holds from the definition of $\mathcal{W}$ and (23) above.

Next, suppose that $k = 1$. Then

$$
    U \left( (\mathcal{W}_m) \mathcal{U} \right) \left( \bar{\ell} \right) = \left( \mathcal{W}_m \mathcal{U} \right) \left( \bar{\ell} \right) \triangledown \mathcal{W}_i \left( \mathcal{W}_m \mathcal{U} \right) \left( \bar{\ell} \right) = \mathcal{W}_m \left( \bar{\ell} \right) \triangledown \mathcal{W}_i \mathcal{P}_{(m, k-1)}[\bar{\ell}]
$$

We must show that $\mathcal{W}_m \left[ \bar{\ell} \right] = \mathcal{U}_{i-1}$. First, observe that $m$ cannot be 0 by our assumption that $i > 0$ (if it was, then we would have $U \left( (\mathcal{W}_m) \mathcal{U} \right) \left( \bar{\ell} \right)$, which is $\mathcal{W}_0$, violating our assumption). As $i > 0$, there must be some preceding element $\mathcal{U}_{i-1}$ and by definition this element is the final element of $[\mathcal{P}_{(m-1, j)}]$. (The follows from the fact that $\mathcal{P}_{(m-1, j)}[\bar{\ell}]$ must be the first element of one of the sequences that are concatenated together to form $\mathcal{U}$.)

From the definition of $[\mathcal{S}]$, this final element is $\mathcal{P}_{(m-1, \rho)}[\bar{\ell}]$, where $\rho$ is the first index greater than or equal to 1 after which the sequence $\mathcal{P}_{(m-1, j)}[\bar{\ell}]$ repeats infinitely. Further, as the sequence $\mathcal{P}_{(m-1, j)}$ is increasing by the induction hypothesis, we have:

$$
    \mathcal{W}_m \left[ \bar{\ell} \right] = \bigcup_{i < \omega} \mathcal{P}_{(m-1, \rho)}[\bar{\ell}] = \mathcal{P}_{(m-1, \rho)}[\bar{\ell}] = \mathcal{U}_{i-1}
$$

Finally, by expanding definitions we have:

$$
    \mathcal{U}_0 = \mathcal{P}_{(0, 1)}[\bar{\ell}] = \overline{F} \mathcal{W} \left( (\mathcal{W}_0) \mathcal{U} \right) \left( \bar{\ell} \right) = \overline{F} \mathcal{W} \left( \mathcal{W}_0 \right) \left( \bar{\ell} \right) \triangledown \mathcal{W} \left( (\mathcal{W}_0) \mathcal{U} \right) \left( \bar{\ell} \right)
$$

$$
    = \overline{F} \left( \mathcal{W}_0 \right) \left( \bar{\ell} \right) \triangledown \mathcal{W} \left( (\mathcal{W}_0) \mathcal{U} \right) \left( \bar{\ell} \right)
$$

We therefore define the sequence $\mathcal{W}_i$ as:
$$
\mathcal{W}_0 = \mathcal{P}_0 = \bar{F}(\mathcal{P}(0,0))[\vec{\ell}]
$$
$$
\mathcal{W}_{i+1} = \bar{F}(\mathcal{W}(m,k-1))[\vec{\ell}] \text{ where } \mathcal{P}_{i+1} = \mathcal{P}(m,k)
$$
It remains to show that this is an increasing sequence. From the definition of $\mathcal{P}$, if $\mathcal{W}_i \equiv \bar{F}(\mathcal{P}(m,k-1))[\vec{\ell}]$ for some $m \leq n+1$ and $k$, then $\mathcal{W}_{i+1} = \bar{F}(\mathcal{P}(m,k))[\vec{\ell}]$ or $\mathcal{W}_{i+1} = \bar{F}(\mathcal{P}(m+1,1))[\vec{\ell}]$.

By the monotonicity of $\bar{F}$, it suffices to show that $\mathcal{P}(m,k-1) \subseteq \mathcal{P}(m,k)$ and that $\mathcal{P}(m,k-1) \subseteq \mathcal{P}(m+1,1)$. For the former, if $m < n+1$ the result holds from the induction hypothesis otherwise the result holds from Lemma 30 part (4). The latter is only possible if $m < n+1$. By Lemma 30 part (1), $\mathcal{P}(m,k-1) \subseteq \mathcal{P}(m+1,0)$, and $\mathcal{P}(m+1,0) \subseteq \mathcal{P}(m+1,1)$ by the induction hypothesis or Lemma 30 part (4) depending on whether $m = n$.

\[\square\]

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

Lemma 31.

1. $∀m : \bar{R}.T_\gamma (m)^i(⊥)$ is increasing.
2. $∀m : \bar{R}.T_\gamma (m)^i(⊥)$ converges.

Proof.

(1) By induction on $i$. The in the inductive step, for flow edges in $\mathcal{W}_P$ the inequality is due to the $\bar{\nabla}$ being an upper bound operator, whereas for flow edges not in $\mathcal{W}_P$ the result holds from the monotonicity of $I_T$ 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, $T_\gamma (m)^i(⊥)$ form the following sequence:

$$
T_\gamma (m)^0(⊥) = ⊥\bar{\nabla} \nabla I_T(T_\gamma (m)^i(⊥)) = T_\gamma (m)^i(⊥) \nabla I_T(T_\gamma (m)^i(⊥))
$$

The sequence $⊥\bar{\nabla} \nabla I_T(⊥), \ldots, I_T(T_\gamma (m)^i(⊥))$ is increasing from the monotonicity of $I_T$ and that $T_\gamma (m)^i(⊥)$ is increasing proved in part (1) above. Thus the above sequence converges after a finite number of steps, giving us the termination result.

\[\square\]

Lemma 32. The sequence $\mathcal{M}_\gamma |_{\mathcal{L}_A}$ converges in a finite number of steps

Proof. By Theorem 13, as each subfixpoint iteration is finite, we can construct a sequence similar to $\mathcal{P}$ in the proof of 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 $\mathcal{M}_\gamma |_{\mathcal{L}_A}, m \geq k$, 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 $\mathcal{M}_\gamma |_{\mathcal{L}_A}$ stabilizes.

\[\square\]

Theorem 14. The sequence $\mathcal{M}_\gamma$ converges in a finite number of steps.

Proof. We first note that, as $I_T$ is deterministic, if $m|_{\mathcal{L}_A} = m'|_{\mathcal{L}_A}$ then $\bigl[\bigcup_{i<\omega} T_\gamma (\text{inj}(m)|_{\mathcal{L}_A})^i(⊥) = \bigcup_{i<\omega} T_\gamma (\text{inj}(m)|_{\mathcal{L}_A})^i(⊥)$. By Lemma 32, there is some $k$ such that $\mathcal{M}_\gamma |_{\mathcal{L}_A} = \mathcal{M}_\gamma |_{\mathcal{L}_A} = \mathcal{M}_\gamma |_{\mathcal{L}_A}$. As $\mathcal{M}_\gamma |_{\mathcal{L}_A} = \mathcal{M}_\gamma |_{\mathcal{L}_A}$ we have that

$$
\bigcup_{i<\omega} T_\gamma (\text{inj}(\mathcal{M}_\gamma |_{\mathcal{L}_A})^i(⊥) = \bigcup_{i<\omega} T_\gamma (\text{inj}(\mathcal{M}_\gamma |_{\mathcal{L}_A})^i(⊥)
$$
Consider next some arbitrary $\vec{\ell} \in L_F$. Let $U_{k+1} = \bigcup_{i < \omega} T_V((\text{inj}(M_{\vec{\ell}}^k))_{|L_A})^i(\perp)[\vec{\ell}]$ and $U_{k+2} = \bigcup_{i < \omega} T_V((\text{inj}(M_{\vec{\ell}}^{k+1}))_{|L_A})^i(\perp)[\vec{\ell}]$. By the above equality, we have that $U_{k+1} = U_{k+2}$, and thus:

$$M_{\vec{\ell}}^{k+1} = M_{\vec{\ell}}^k \cup U_{k+1} = M_{\vec{\ell}}^k \cup U_{k+1} \cup U_{k+2} = (M_{\vec{\ell}}^k \cup U_{k+1}) \cup U_{k+2} = M_{\vec{\ell}}^{k+2}$$

We therefore have $M_{\vec{\ell}}^{k+1}|_{L_A} = M_{\vec{\ell}}^{k+2}|_{L_A}$ and $M_{\vec{\ell}}^{k+1}|_{L_F} = M_{\vec{\ell}}^{k+2}|_{L_F}$, whence $M_{\vec{\ell}}^{k+1} = M_{\vec{\ell}}^{k+2} = M_{\vec{\ell}}(M_{\vec{\ell}}^{k+1})$. Thus, in $k+1$ steps, the sequence reaches a fixpoint of $M_{\vec{\ell}}$, and thus the entire sequence stabilizes in finite time. $\square$