SANDIA REPORT
SAND2020-10486
Printed September, 2020

Sandia National Laboratories

Language Independent Static Analysis (LISA)

Douglas P. Ghormley, Geoffrey E. Reedy, Kirk T. Landin

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

ABSTRACT

Software is becoming increasingly important in nearly every aspect of global society and therefore in nearly every aspect of national security as well. While there have been major advancements in recent years in formally proving properties of program source code during development, such approaches are still in the minority among development teams, and the vast majority of code in this software explosion is produced without such properties. In these cases, the source code must be analyzed in order to establish whether the properties of interest hold. Because of the volume of software being produced, automated approaches to software analysis are necessary to meet the need.

However, this software boom is not occurring in just one language. There are a wide range of languages of interest in national security spaces, including well-known languages such as C, C++, Python, Java, Javascript, and many more. But recent years have produced a wide range of new languages, including Nim, (2008), Go (2009), Rust (2010), Dart (2011), Kotlin (2011), Elixir (2011), Red (2011), Julia (2012), Typescript (2012), Swift (2014), Hack (2014), Crystal (2014), Ballerina (2017) and more.

Historically, automated software analyses are implemented as tools that intermingle both the analysis question at hand with target language dependencies throughout their code, making re-use of components for different analysis questions or different target languages impractical. This project seeks to explore how mission-relevant, static software analyses can be designed and constructed in a language-independent fashion, dramatically increasing the reusability of software analysis investments.

ACKNOWLEDGMENT

Supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

We would especially like to thank Shelley Leger for her assistance on this project.

CONTENTS

1. INTRODUCTION

> Reusing research is freakishly hard.
>
> *(Dan Guido, Trail of Bits)*

Software is becoming increasingly important in nearly every aspect of global society and therefore in nearly every aspect of national security as well. While there have been major advancements in recent years in formally proving properties of program source code during development, such approaches are still in the minority among development teams, and the vast majority of code in this software explosion is produced without such properties. In these cases, the source code must be analyzed in order to establish whether properties of interest hold. Because of the volume of software being produced, automated approaches to software analysis are necessary to meet the need.

However, this software boom is not occurring in just one language. There are a wide range of languages of interest in national security spaces, including well-known languages such as C [30], C++ [47], Python [50], Java [2], Javascript [24], and many more. But recent years have produced a wide range of new languages, including Nim [41], Go [20], Rust [35], Dart [10], Kotlin [43], Elixir [6], Julia [8], Typescript [9], Swift [19], Hack [26], Crystal [4], Ballerina [5], and more.

On the other hand, as software becomes incorporated into more and more interconnected aspects of society, there is consequently an increase in the number of questions we need to ask and answer about software to ensure the public good. Examples of such questions include "*What network packets might this software generate?*", "*What indicators might this malware leave in the system?*", "*Does this software adequately protect user privacy information?*", and more.

We use the term *mission questions* to refer to such high-level questions and the term *mission application* to refer to a tool designed to help answer one such mission question in an automated fashion. We use the term *target language* to refer to the source code language of the program for which we are attempting to answer a mission question. In contrast to this, the term *meta-language* is used to refer to the language in which the mission application was written.

There has been considerable academic research in recent decades in automated software analysis techniques and approaches. As opposed to the high-level mission questions discussed above, much of this research is focused on techniques for learning particular low-level facts, approaches for assessing whether specific properties hold, improvements in scalability or accuracy, and the like. Specific examples of such research include new pointer analysis algorithms with better scalability, performance, or accuracy properties, new types of object sensitivity to selectively retain accuracy, heap shape analysis, etc. These advances represent very important but very

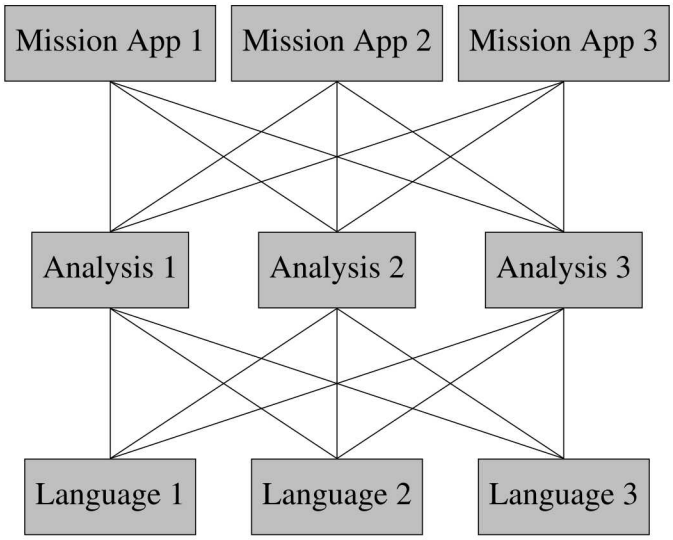


Figure 1-1. When multiple mission applications leveraging multiple analysis techniques are needed across multiple languages, then a naive approaches leads to reimplementing each analysis technique (A) for each target language (L), then synthesizing a new mission application (M) out of each of these reimplemented analyses. The effort involved in this approach is on the order of $A \times L \times M$.

focused elements, many of which must be brought together into a single mission application in order to answer a high-level mission question.

There are a few projects which bring together a number of low-level software analysis algorithms to create a mission application to answer a specific mission questions, though the most common mission questions addressed (often implicity) relate to software vulnerabilities. Such efforts include angr [45], Infer [11], CodeSonar [51], Coverity [7], Mayhem [12], and more. In the vast majority of such tools, the low-level software analysis components are intermingled with the details of both the target language and the high-level analysis needed to answer the high-level mission question. Consequently, investments in automated software analysis tools are typically not reusable across either target language or mission application. Therefore, most tools must be completely rewritten in order to incorporate new analysis techniques, support a new target language, or answer a different mission question. Illustrating this point, Dan Guido of Trail of Bits, a company specializing in software analysis for security assessments, commented during his keynote talk at BSides Lisbon, 2016 that "re-using research is freakishly hard, since you have to write all the stuff from scratch."

Illustrated mathematically, the predominant approach for building M mission applications targeting L languages while using A analysis techniques requires order $A \times L \times M$ work to produce. See Figure 1-1. Given the vast and growing array of mission questions, analysis techniques, and target languages, the investment required to produce all needed analyses in a portable fashion is untenable.

The focus of our work is not on how various analysis components can be assembled to answer a given high-level mission question, but rather on how to decouple these analysis factors (A, L, and

M) to enable development investments to be more easily reused. To accomplish this, our work explores how an analysis infrastructure and target-language support components might be designed to maximize code reuse across mission applications and target languages, reducing the work involved in constructing a new mission analysis for a new target language.

1.1. Our Approach

When the need for cross-language support in static software analysis is raised, a solution commonly offered is that of using an intermediate language (IR). Roughly speaking, this idea is to translate a target program from its original source code into an IR, and to write all analyses against this IR. A translation is needed from each target language of interest, but once this work is complete, the idea is that the mission applications operate over the IR, enabling them to be used across a broad set of target languages.

While we agree that it is vital to decouple the $A \times L \times M$ factors, for reasons discussed in detail in Section 3, we believe an IR-based approach is inadequate to solve the problem. Instead, our approach to this same goal is to conceive of a mission application as primarily constructed from a collection of well-designed components, which can be mixed-and-matched to create different analyses against different languages to answer different mission questions. We consider different types of reusable components: components used to construct a flexible analysis framework, components used to track the data values of the program necessary to answer a particular mission question (often referred to as *domains*), and components needed to implement individual semantic features of the target language. For a flexible analysis framework, we take inspiration from the Abstracting Definitional Interpreters (ADI) [18] work. For our set of domains, we take inspiration from Apron [28], a well-known library of reusable domains for numerical values. For semantic features of target languages, we take inspiration from nearly any undergraduate programming languages design class. A novel mission application would be constructed from the appropriate analysis framework, domain, and language components needed to answer the mission question at hand on the selected target language.

As mentioned, our interpreter design has been inspired by the ADI work, which takes this component-based approach, albeit only for a single, simple language. That work demonstrated that replacing the specific component implementations with various alternative strategies yields analyses with different characteristics (e.g., scalability, precision). We explore features of more realistic languages in this work. We also design the decomposition approach with a view to generalizing across many programming languages. For example, Python and C both support arrays, but with semantics that differ in important ways; our approach is to factor out the commonality between them to maximize code reuse across different target languages and mission applications. Finally, we design the component interfaces so that different instantiations of those components can be freely substituted.

1.2. Approach Examples

We illustrate our approach with two examples.

First, many programs make use of locking primitives. In some cases, these locking primitives are built-in to the language (e.g., Java, Python, x86), and in other cases they are provided either by the operating system or runtime system (e.g., C library). While the forms of the locking primitives vary, the concept is nearly universal. Rather than base any given mission application on the *syntax* of the target language's locking mechanism, basing it on configurable *semantics* would enable more code reuse across target languages. However, the detail to which the semantics of the locking mechanisms are tracked depends on the mission question being answered as well as aspects of the target program. Depending on the mission application, different strategies for tracking the lock states may be appropriate: for small programs, fully-precise information regarding which threads hold which locks while accessing which shared variables may be appropriate; for much larger code-bases, it may be necessary and appropriate to abstract away such details and merely keep track of which shared elements had some lock held during access; for other mission questions, ignoring locking behavior altogether may be appropriate. Isolating the implementation of the locking semantics to a pluggable component in the analysis engine helps maximize code reuse not only across different target languages, but also across different mission questions.

As a second example, different languages employ different semantics for object and field name resolution. In assembly representations of C programs, objects have addresses and the fields in them have offsets; object addresses may be determined dynamically or by the loader at runtime, but offsets are fixed by the compiler at compile time based on certain layout rules. In assembly representations of C++ with class inheritance, a name must be resolved by determining which class in the hierarchy defines it. In a language with dynamic typing, such as Python, a dictionary must be consulted at runtime at each layer of the object hierarchy to determine how it should be resolved. Though these examples differ, there is actually a relatively small set of different name resolution semantics from which the majority of languages choose in their designs. There are also various strategies for reasoning about these object fields that may be appropriate for different mission applications. Some applications may demand precise reasoning that distinguishes between parent and child classes in a class hierarchy; other applications may opt for performance or scalability gains by ignoring the boundary between parent/child classes and lumping all fields together into a single, monolithic instance; and yet other applications may abstract even further and not distinguish among different instances of a particular class.

This research is a preliminary exploration into a method of structuring static analysis tools so that the bulk of the code for a given mission application is independent of any given target language. By decomposing analyses along language feature lines we avoid rewriting the entire analysis system to accommodate each language's semantics and we increase component reuse when implementing a different analysis. There is much work left to be done.

2. RELATED WORK

This section presents a number of lines of research that are related to and have informed our work.

2.1. Modular Interpreter Design

Foundational work in a modular way of viewing program languages, and hence the analysis thereof, came from Moggi in [39]. This work (which is actually course notes from a class he taught) explored leveraging category theory [42] from mathematics as a method for capturing the semantics of a computation, independent from its syntax. Building on this view, Moggi then explored leveraging monads as a notion for modeling computation [38] in a modular way that permits *gluing together* semantics of languages, though there were challenges that others discovered and addressed later. To express the semantics of a complex language, one starts with a monadic notion of basic computation and then, at each step, one applies a monad constructor to add one new feature to the language.

Wadler in [52] used this monadic approach to demonstrate in a very readable and accessible way how to construct a modular interpreter of a target language. In particular, he demonstrated how semantic features with side-effects in the target language (*impure features*) could be easily modeled in a purely functional interpreter using these monads. Wadler illustrated using monads to express features such as exceptions, output, call-by-name vs. call-by-value, and more.

Wadler's approach in [52] had much to recommend it, but required altering "by hand" the data structure definitions in order to accommodate the new language features being implemented. Additionally, Wadler's approach did not demonstrate how to compose together the various monads modeling the target language features. Steele set out in [46] to rectify these shortcomings, pursuing a vision of composable language elements as LEGO blocks or Tinkertoys, which could be used interchangeably and in combination to construct an interpreter with a richer set of features. He proposed the *pseudomonad* which is used to construct a *tower* or *stack* of monads in composition.

Liang et al. in [33] then showed that Steele's proposed pseudomonads failed to properly incorporate key features such as the store and the environment, as well as incurring challenges with the type system. They demonstrated that Steele's pseudomonads are really just a special kind of monad transformer first suggested by Moggi in [39]. Liang et al. then identified a number of shortcomings of all the previous efforts and offered new approaches to monad lifting to address those. Their work explores how rearranging the composition of monads in the monad stack can capture different semantics in the target language.

2.2. Abstract Interpretation

Interpreting a target program involves keeping track of the state of computation of that program as it runs. A dynamic interpreter need only keep track of the current state as the program runs on a single input. When more than one input is of interest, or no particular input, things get much more interesting and complex. In the limit, static analysis has the goal of keeping track of all possible computations of a program. This permits reasoning about not just what a program *did* do on a given input, but what it *could* do if given the right input.

One key challenge in static analysis is scalability. Keeping track of all possible state of computation across all possible inputs is impractical. However, by observing that not all minutiae of state is required for every mission question, one can imagine retaining only enough information from the static analysis to answer the specific mission question at hand. The question then becomes one of how to manage the process of retaining detail vs. removing detail from the state representation in such a way that one still has confidence that the analysis is reasoning about all possible executions, though less precisely. Informally, this means moving from analysis that may have been able to answer *yes* or *no* to a particular analysis question to *yes*, *no*, or *maybe* (this latter result arising from possible lack of precision).

The seminal paper in using abstract interpretation to statically analyze programs was published by Cousot and Cousot [16]. This paper introduced the notion of leveraging from mathematics a complete lattice to represent the program property of interest. The interpreter, as it calculates program states, also calculates the property of interest for each state in the program. The formal properties of the lattice enables the interpreter to merge states in a mathematically sound way, using higher lattice elements to capture summary information for a collection of values, removing precision in a controlled way and gaining scalability in return. A fixed-point algorithm is used, with the properties of the lattice essential to guaranteeing termination.

A key aspect of this approach is to construct abstractions that retain sufficient detail to answer the mission question at hand while eliding enough detail to make the tool scalable on real programs of interest. Some examples of such abstractions include numeric intervals [14], strings [29], objects [1], memory states and shapes [27], and more. In many cases, what is important to track is the relationship between different values in a program. For example, an array index and the buffer length. Examples of relational domains include the octagon domain which can track relationships of the form $\pm x \pm y \leq c$ [37], and general convex polyhedron [15].

2.3. Abstract Interpreters and Machines

There has been academic work over the last decade in devising ways to build flexible abstract interpreters that can easily be reconfigured to perform multiple styles of analysis on multiple programming languages, and that work is leveraged by the analyzers built in this project. This section summarizes key aspects of that work which are relevant to our research.

2.3.1. Abstracting Abstract Interpreters

Much of the work that is leveraged to implement interpreters in this project all stems from the 2010 paper *Abstracting Abstract Machines* (AAM) [49].

Lambda Calculus is the foundation for the vast majority of formalized programming models, and has a very small transition relation (used to determine which next states are reachable from a given current state) that defines its entire small-step operational semantic (see Section 6 for more explanation). However, this definition of the Lambda Calculus semantic makes heavy use of substitution and context matching, which are not easy to implement in an efficient manner. Luckily, there many different ways to reify this semantic, which are commonly referred to as *abstract machines*, and some of these reifications are very efficient to implement in code [21]. Many programming language interpreters (especially ones for functional languages) are modeled off of one of these abstract machines.

The core abstract machine concept that AAM is based off of is known as the *CEK* machine [22], which is short for *Control-Environment-Kontinuation Machine*. The *Control* part of the machine refers to the expression to be evaluated next, and behaves the same as it would in the core Lambda Calculus reduction relations.

The *Environment* part of the machine solves the efficiency problem of substitution in the Lambda Calculus-style semantic. Whenever a lambda-expression is applied, all bound names inside the expression must be substituted with whatever names/values the expression is applied to. Doing this directly requires a scan of the entire body of the lambda expression, and repeated applications/substitutions will require multiple scans over the same code, one for each substitution, and a given name may be substituted for multiple times before it is directly used [21].

The solution to this is to add an *Evaluation Environment* to the transition relation, which keeps a mapping of the current name-bindings, and allows the substitution to happen lazily. When a new substitution is required, the reduction relation just updates the evaluation environment with the variable name being substituted and its new value, and when the evaluator encounters a name it just looks up its value in the evaluation environment [21].

The *Continuation* ("Kontinuation") part of the abstract machine provides an efficient solution to the context-finding problem. In many Lambda Calculus-style semantics, a given outer form may require its inner forms to be evaluated before it can be evaluated. In this case an evaluator for this will iteratively split the form into an outer "Context" and an inner "Redex". For example, consider an expression tree built up of standard addition operators, $(a+(b+(c+d)))$. The rules of arithmetic require that $x_1 \leftarrow c+d$ is evaluated first, followed by $x_2 \leftarrow b+x_1$, followed by $a+x_2$. In this case, the first reduction has to split the expression twice, and the second reduction has to split the expression once to find their final redexes. This repeated set of searches doesn't scale very well, and can be solved by adding a *Continuation Stack* to the evaluator, which keeps track of the current state of nested splitting that the evaluator is currently in [21].

With these two changes, evaluation of Lambda Calculus-style semantics is quite efficient.

2.3.2. Abstracting the Abstract Machine

The AAM work takes this CEK machine and turns it into something that is suitable for performing Abstract Interpretation. The first step is to use a modification of the CEK machine, known as a *CESK* machine, which adds a persistent store *S* to the machine state. Name-binding and continuation stack are modified to store all values as pointers to data living within the data-store, *S*. This modification allows the store to become the "Single point of abstraction" [49] in which data/continuations can be modified/merged as the abstraction scheme sees fit. The final step that they do is add a time-stamp field to the CESK machine, enabling the analysis to track whatever context sensitivity is needed.

2.3.3. Defining Abstract Machine Semantics in terms of Monads

A follow-on paper, *Monadic Abstract Interpreters* [44], takes the same abstract machine formulation presented in [49] and interprets many of its operations in terms of monads. When using monads, the outer interface for an Abstract Machine is going to look very similar to the version without monads, however the main change is in how the internals of the machine are constructed. In the non-monadic version, the internals of the abstract machine were more-or-less monolithic, but the internals of the monadic machine are quite modular.

The monadic machine will implement things such as the name-binding, storage, non-determinism, etc. as monad transformers. These transformers will then be stacked up to generate the final interpreter implementation. It is important to note is that different orderings in the stack of the monad transformers will generate different styles of abstract interpreters [18].

2.3.4. Abstracting Definitional Interpreters

Abstracting Definitional Interpreters (ADI) [18] took the ideas presented in the previous two papers and applied them to the world of big-step semantics. While a small-step semantic, as presented above, relates a program/machine state to the next program/machine state, the *big-step semantics* relate a program/machine configuration to the final value/machine configuration, which arises when the program is finished evaluating. Because a big-step semantic collapses many individual small steps into a single big step, the semantic is usually implemented with a *recursive evaluator*. The ADI paper presents how to define an extensible abstract interpreter in terms of one of these recursive evaluator functions.

One of the main ideas that the ADI paper leverages is that of *open recursion* (see Section 9.1), in which a recursive function, instead of being hard-coded to recurse to itself, takes its recursive call as a parameter. This formulation allows for analyses to create arbitrary co-recursive function call chains. The paper then derives the required machinery to perform the kinds of analyses that were shown in the previous papers [18].

The ADI paper uses the same sort of monad-based construction as in [44]. The monadic construction is not necessary to perform abstract interpretation, however it adds the same benefits

that it did to the abstract machine formulation, in that it allows much greater modularity between the internal interpreter components and enables changing analysis strategies with minimal effort. This is important to our work because as the mission question changes, the strategies used to manage the process of abstraction and to control which information is retained with precision and which is elided will in general change as well.

The monadic approach also provides a stylistic advantage for the recursive evaluator approach. In the AAM approach, the machine state was represented by a 5-tuple and the transition relations related 5-tuples with 5-tuples. In the functional world, as is used by ADI, functions returning multiple values are much less common, and a function returning a 5-tuple of values is viewed as having particularly bad taste. The monadic construction allows for each evaluator function to return just a single value, with the rest of the values handled by the monads.

2.3.5. How it Applies to This Project

In this project, we built one of our main prototype interpreters using the recursive evaluator construction provided by ADI. Our open-recursive evaluator structure, in addition to what was presented in the paper, enables the construction of an interpreter for a given target language by recursive composition of smaller interpreters, each supporting a subset of the target language's semantic features, wrapping the combined result into a top-level dispatcher function. Our project was also able to combine languages that used separate evaluation environments by adding in a new monad that exposes the appropriate evaluation environment for a given interpreter while transparently threading through the rest of the environments. Details are presented in Section 9.

3. THE (DIS)UTILITY OF INTERMEDIATE REPRESENTATIONS

Modern compilers and runtime systems leverage intermediate representations to enable generic optimizations and code generation that work for a wide variety of source code languages. An intermediate representation (IR, sometimes also called an intermediate language or IL) is a programming language that inhabits an *intermediate* level of abstraction between a source language and a target language such as machine code. The intermediate language often retains only a subset of the features of the source language while adopting some aspects of the target language, hence the name *intermediate*. IRs are typically designed to ease transformations (e.g., optimizations) and other processing (e.g., translation to another intermediate or final representation).

IRs establish an interface between a *front-end* which translates a source language program into the intermediate representation and a *back-end* which executes or translates the intermediate representation. This separation enables a system to support multiple source languages (by providing multiple front-ends) and target execution environments (by providing multiple back-ends). Notable IRs used in this way include LLVM IR [31], JVM bytecode [34], and Common Intermediate Language [13].

It is tempting to use an IR to achieve language independence for static analyses. Ideally we could separately write (a) translators from source languages to the IR, (b) analysis components for the IR language features, and (c) mission application using these components. This could indeed replace some multiplicative cost factors with additive factors. However, we posit that this ideal is unachievable because the design of any IR fixes a level of abstraction which cannot be appropriate for all source languages, abstraction techniques, and mission questions. This challenge is recognized within the LLVM community: the Clang compiler includes analyses and transformations that are performed on the abstract syntax tree before emitting LLVM IR because the IR does not carry enough semantic information to support these operations.

We believe that any IR design will involve two challenges for static software analysis. First, any IR design will impose an undue burden on analysis by erasing explicit information about the source language semantics. When the IR semantics do not allow a trivial representation of the target language semantics, information is lost and the translation is referred to as a *lowering*. When the mission question relies on the information lost, a precision and/or scalability burden is imposed on the analysis to recover the information, which often cannot be done precisely and accurately.

Second, if the IR design seeks to reduce or eliminate the need for lowering it must provide for a trivial encoding of source language features. Every target language semantic feature must be faithfully encoded by the front-end translator into the IR form of the program. Under this strategy, the IR will eventually accumulate features whose interplay is difficult to specify and even more

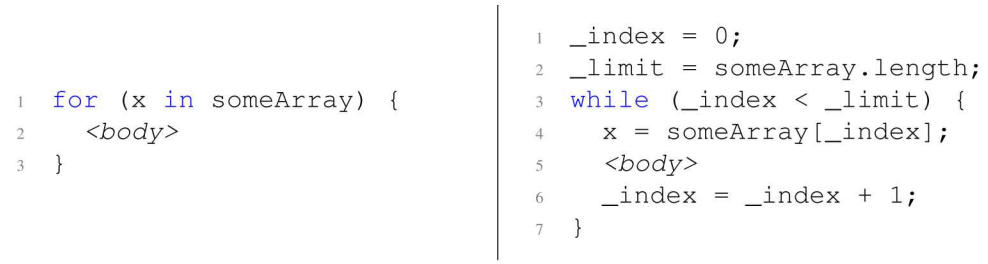


Figure 3-1. A for-each loop over an array and a possible lowering

difficult to analyze. If the IR employs annotations or other metadata to encode information lost during lowering or to control feature interaction, language independence is essentially lost—there is one intermediate *language* but with many *dialects* which each analysis must support.

We illustrate the information lost by lowering and the impact to analysis with an example. Many languages have a construct for iterating over values in an array or other collection type. Suppose the intermediate representation does not provide this feature directly and occurrences in the source program must be lowered to the IR. An example of this kind of loop and a possible lowering are depicted in Figure 3-1. The source language construct has several properties that are guaranteed by the semantics of the language, including

- each element of the array is accessed at most once,
- if the loop does not exit early (e.g. by a break statement or exception) each element of the array is accessed exactly once,
- no out-of-bounds access to the array is possible when computing x, and
- the trip count of the loop is equal to the number of elements in the array.

An analyzer that works with the original source language can assume all of these properties without any additional effort. In the lowered version, however, sophisticated analysis is needed to recover these properties. In particular, a relational analysis—one that captures the relationship between *_index* and *_limit*—is necessary to precisely analyze the loop, a considerable cost if the end-use analysis does not otherwise require that precision.

Even when these properties can be recovered by the analysis the lowering can still have a detrimental effect on precision and performance of the analysis. An analyzer may be able to use details of particular array abstractions to realize a more efficient analysis of the for-each loop than a loop involving explicit array indexing. When the loop is lowered, the opportunity to apply these kinds of optimizations is typically lost.

An intermediate representation can be viewed as a static binding of source language semantics to analyzable primitives. We have argued that there is no single best binding for all situations. Our approach sidesteps this limitation by allowing dynamic collaboration between the analysis components. An analysis can use language-specific components in combination with generic components to reduce implementation costs while retaining the precision and scalability advantages possible when working directly with the target language and its high-level semantics and implicit guarantees.

4. SEMANTIC COMPONENTS OF PROGRAMMING LANGUAGES

Abstracting Definitional Interpreters (ADI) [18] shows a design recipe for constructing an abstract interpretation analysis framework that is no more difficult than constructing a regular interpreter for a given programming language.[1] Once the extensible interpreter is built, an analysis is realized by supplying (a) implementations for the language's primitive operations, (b) recursive evaluators that compute a fixed point, and (c) a stack of monadic effects that support them. The bulk of the work in defining analyses for a new programming language comes with implementing the primitive operations. While there truly is a great diversity of programming languages, when one considers their semantics as opposed to their syntax they are all more alike than they are different. We can define generic, reusable analysis components that target these semantic similarities to reduce the cost of writing new mission application. Furthermore, well-designed interfaces for these components allow substituting alternative implementations of these analysis components. Each mission application can choose the components that make the best trade-offs between precision and scalability for the specific analysis task and characteristics of the program being analyzed.

In this section we highlight a few common language features and discuss how their semantics differ from language to language. For each feature, we consider generic analysis components that contribute to their implementations and how existing abstractions and analysis techniques fit into the decomposition. We also show that mission applications themselves may be a kind of analysis component that can be reused across programming languages.

4.1. Arrays

Languages that feature arrays share the semantics that reading from an index in the array yields the last value stored to that index. This forms the foundation for an array component interface. Many languages provide additional operations for arrays besides read and write; these operations should be part of the interface as well so that components can provide efficient precise implementations. Examples of these operations are slicing, splicing (as in Javascript), in-place reversal, bulk writes, and for-each loops. The array components will likely need to collaborate with other analysis components to, for example, evaluate and test indices—each implementation would declare the dependencies that must be provided when the component is instantiated.

Programming languages can and do differ in the capabilities of and restrictions on arrays and how they interact with other language features, but this does not preclude the reuse of array

[1] We estimate that the writing an interpreter in this style requires less effort than writing a translator from the language to an intermediate representation.

components across languages. Some of these differences can impose constraints on which array component implementations are appropriate while others merely affect how the analysis uses the component. For example, differences in how arrays are indexed (0-based, 1-based, or custom ranges as in Ada, etc.) need not be handled by the array component directly, instead the analysis can translate between the language semantics and the component interface.

The program analysis literature describes many abstraction techniques for arrays that could be embodied within an analysis component. Cousot et al. [17] describe one such array abstraction and discusses others. Implemented once, these components could be readily reused across a variety of analyses for different programming languages and mission applications.

We discuss a specific design for array components in section 8.

4.2. Objects

Programming languages express considerable variety in their object systems, but there are still some common themes: dynamic method dispatch on the runtime type of the receiver, either class- or prototype-based inheritance, and uniquely-identifiable object instances containing instance-specific field data. We suggest a number of analysis components based on these themes which can ease implementation of object analysis for many different mission questions and many different target languages. We also show that the object system itself can be used as a parameter to other components implementing general techniques.

Instance data storage can be delegated to shared analysis components that implement a range of abstraction strategies. For example, depending on the level of precision required, an analysis may require field-sensitive or field-insensitive implementations. In some languages the field set for instances are closed (limited to the fields declared in the instance's type) while in others it is open (fields are created on demand by assigning them a value). The latter, referred to as *open objects*, pose the greater challenge for analysis since instances of the same type may have different field sets, the details of which must be discovered during analysis. Generally, a storage implementation suitable for open objects can be used for closed objects, but having a fixed set of fields enables various analysis efficiencies. Therefore, we expect that component implementations would be specialized for one case or the other.

Since object instances commonly have a unique identity and a program can have multiple references to the same object, objects must be allocated in an abstract store. The semantics of such a store are widely applicable across many languages. The abstract store can also be provided by a shared analysis component implementing a specific strategy such as recency abstraction [3]. To increase precision of the abstract store, abstract garbage collection [36] can be employed. As in regular garbage collection, objects must be traversed to find the set reachable from garbage collection roots. Abstract garbage collection can be realized in a shared analysis component but it must make use of the object system to enumerate the values directly reachable by an instance. In this scenario, the object system of the analysis is itself an analysis component that is supplied as a parameter to the abstract garbage collection component.

4.3. Exceptions

The ADI technique allows for a particularly straightforward implementation of exception handling: they are simply another effect in the interpreter's monad stack. Different analysis strategies for exception handling can be provided as implementations of the exception handling effects.

4.4. Lock Discipline Mission Application

In some cases, mission applications themselves can also be conceived as a kind of analysis component. As described in Section 1.2, numerous target languages share identical or nearly identical semantics of locks. A mission application designed to detect misuse of locks would be constructed from an analysis component that provides an abstraction for mutex locks (e.g., creation/initialization, lock, and unlock) and target-language-specific operations on that abstraction. Such an analysis would raise alerts when proper lock discipline has been violated. Other analyses for different purposes and for different target languages may also incidentally require tracking lock state at varying degrees of fidelity. In such cases, the same components used to create a lock discipline mission application can be reused for other mission applications, as can many of components used to analyze the new target language for different mission applications.

5. ALGEBRAIC THEORIES AND STATIC SOFTWARE ANALYSIS

Like many approaches to software abstraction, key design elements of our approach are loosely based on the concept of an *Algebraic Theory*. This section will provide a brief background on the relevant aspects of algebraic theories and how they apply to modeling the behaviors of the domains that we wish to develop. The precise definition of an algebraic theory can be somewhat subtle and is beyond our scope here, though the curious reader is referred to Bill Lawvere's thesis [32].

A *Single-Sorted Algebraic Theory* consists of a type ("Sort") T along with functions f, where $\forall n \in \mathbb{N}$, a function f of arity n is of type $T^n \to T$, will map an n-tuple of T-elements $T \times T \times \ldots T \to T$. It is also possible to have nullary functions $T^n \to 1$ and n-ary functions that map from a tuple to a tuple $\exists n, m.\ f : T^n \to T^m$. However, any function mapping to a tuple of outputs must be factorizable to a tuple of functions, each mapping to a single output: $\forall n \in \mathbb{N}.\ \forall m \in \mathbb{N} > 2.\ \forall f : T^n \to T^m,\ \exists f_1, f_2, \ldots, f_m.\ f(t^n) = \langle f_1(t^n), f_2(t^n), \ldots, f_m(t^n) \rangle$

The core operations that are required to be part of every algebraic theory are the projection operations, $\pi_{i,j} : T^i \to T$, which return the j-th element of a tuple, and all tuplings of these projections operations, which form sub-tuple operations.

Everything presented up until this point is boilerplate, which is required by every algebraic theory. Past, this boilerplate, each algebraic theory will define (a) additional operations, and (b) universally-quantified consistency rules that prescribe how these additional operations behave. It is these additional operations which make an algebraic theory unique.

Given some logical system M, we say that M *models* our algebraic theory A, $M \models A$, if we are able to map the appropriate types and operations of M into A such that all consistency rules of A are satisfied. This is what goes on when one designs software abstractions. The abstraction is the algebraic theory Abs and then implementations are simply models of the theory: $\mathsf{Impl} \models \mathsf{Abs}$.

5.1. Simple Example

One of the simplest examples of an algebraic theory is that of a *Monoid*. A monoid is a semi-group with an identity element. This means that, in addition to the boilerplate, there are two additional operations present.

1. Identity $I : 1 \to T$, which generates the identity element
2. Multiplication $(*) : T \times T \to T$

These operations satisfy the following consistency rules

- Associativity $\forall t_n \in T, (t_1 * (t_2 * t_3)) = ((t_1 * t_2) * t_3)$
- Identity $\forall t \in T, t * I = t \wedge I * t = t$

There are many kinds of domains that model the theory of a Monoid: integers with addition, integers with multiplication, floating point numbers with these operations, multiplication of square matrices, string concatenation, data aggregation, etc. Any software component that is shown to be correct using the algebraic theory of monoids can be instantiated to use any one of these models, and its correctness is guaranteed.

5.2. A More Complex Example: Key-Value Stores

To model key-value stores, we need to extend the notion of a single-sorted algebra to that of a *multi-sorted algebra*. The main difference here is that the algebraic theory now defines multiple types ("sorts") for the functions to operate on instead of a single one as before.

We can describe a typical *Key-Value Store* abstraction using a multi-sorted algebraic theory that has three sorts:

Store the type of the storage

Key the type of the key

Value the type of the value

And two operations `set` and `get`:

`set` The set operator has a signature of `set` : Store $\times$ Key $\times$ Value $\rightarrow$ Store. It takes a Store, Key, Value, and returns a new store in which maps the new key to the new value. If there was a different value belonging to the same key in the input Store, it is overwritten.

`get` The `get` operator requires that we first decide what to do when it is called for a key that doesn't exist in the Store. There are multiple valid options here, and two of them are:

1. Leave the behavior of `get` undefined. With this choice, `get` becomes a *partial function* (a function that is not defined for all inputs), so this algebraic theory becomes an *essentially algebraic theory* [40], in which partial functions are allowed. The signature of `get` is Store $\times$ Key $\rightarrow$ Value. For many software analysis tasks, this is a great approach because it doesn't commit to any particular error handling scheme. Concrete key-value stores that throw exceptions when a key isn't found also model this theory since an exception path is taken, which means that the *get* function never returns in this case, thus is undefined.

2. Add a new sort, Error, to the algebraic theory, which will be returned if the lookup key is not found. The signature is `get` : Store $\times$ Key $\rightarrow$ Value $+$ Error. This theory is modeled by many different key-value store structures, mainly in strongly-typed ML-style languages that have easy-to-use variant types.

For the rest of this example, we will use the partial-function theory.

The behavior of a Key-Value store can be summed up with a single consistency rule.

$$\forall S \in \mathsf{Store},\ k,k' \in \mathsf{Key},\ v \in \mathsf{Value}.$$
$$\texttt{read}(\texttt{write}(S,k,v),k') = \begin{cases} v & \text{if } k' = k \\ \texttt{read}(S,k') & \text{if } k' \neq k \end{cases}$$

This rule can be decomposed into multiple parts:

1. **Read-After-Write:**
   If a store is written to with a key and value, reading that key from the updated store will subsequently return that value.
2. **Independence of Writes:**
   If a store is written to with a given key, reading from the updated store with other keys will return the original values stored at those keys.
3. **Inductive Structure:**
   This property inductively holds for arbitrary sequences of reads/writes.

6. MODELING SEMANTICS IN INTERPRETER DESIGN

The main way to express the operational semantics of a system is through the use of expressions and reduction relations. In the vast majority of cases, expressions are represented by an inductively-defined type, giving them a tree-like structure. Reduction relations precisely specify the steps that a computational system will use to transform the program to its output.

One way of specifying these reduction relations is with *small-step semantics*, in which each reduction relation specifies one atomic unit of computation that the system can perform. In this case, the execution of a program consists of chaining a sequence of reduction steps that transform the input expression to an expression that is in *normal form*, in which there are no longer any reductions that can be performed [21].

Another way of specifying these reduction relations is with *big-step semantics*. Instead of the reduction relation just specifying a single step in the computation for a given input form, Big-step reduction relations directly specify the evolution from an input form to its normal form. Instead of things happening in multiple *small* steps, everything happens in one *big* (recursive) step. Each of these approaches to semantic modeling have their benefits and drawbacks, and each of these approaches gives rise to different types of practical machinery for interpreting a language.

6.1. Practical Machinery for Small-Step Semantics

The core component of an interpreter that implements small-step semantics is a syntax tree to syntax tree transformation. Each reduction will transform the input syntax tree into another syntax tree that is one step closer to termination. Each of these transformations is accomplished by splitting the input into two parts: the reducable expression (called the *redex*), which is the portion of the expression that the reduction rule will apply substitution to, and the context, which is the portion of the expression that surrounds the redex and receives no substitution in the given step. For example, there are three reduction relations defined for lambda calculus:

1. α reduction, which renames a bound identifier
2. β reduction, which "applies" a lambda expression to a value by substituting the value in for each bound occurrence of the parameter identifier in the body of the lambda expression
3. η reduction, which collapses an indirect application of a lambda expression to a direct one.

There are two main drawbacks that come up when attempting to directly implement small-step reduction relations [21]:

1. Splitting the expression into context and redex takes time that is proportional to the size of the entire program.

2. Substituting values in for identifiers also takes time that is proportional to the size of the program.

Luckily there are straight-forward solutions to both of these problems, transforming the time complexity of each from $O(n)$ to $O(1)$.

To speed up the splitting of the context/redex for an expression C, the solution is to add a *continuation stack* (K) to the interpreter. We then change our reduction relations to operate on the entirety of C. If, at a step, the outermost part of C is something that should be part of the context, then the reduction relation peels of the outermost part of C, inserts a *(hole)* into the outer expression, where the inner expression once was, pushes the outer expression, with its hole, onto the continuation stack, K, and then proceeds to reduce the inner expression. This stack-pushing continues until the expression C is something that is directly reducible to a value. At this point, C is reduced to a value V, and the reduction relation, seeing that V is no longer reducible, will pop the top expression off of stack, K, put V into the *(hole)* and then reduce that expression.

To eliminate the computational cost of identifier substitution, we use an *evaluation environment*. The evaluation environment is an extra piece of data (in addition to the expression C) that the reduction relation will use to operate. This evaluation environment consists of an identifier-to-value mapping for any identifiers that are currently bound to values. In lambda calculus, instead of the reduction relation directly substituting each of the bound ids every time it performs a beta-reduction, the reduction relation will not immediately perform the substitution, but store a mapping from the id to its value in the evaluation environment. After that, evaluation will continue as normal. Whenever the reduction relation encounters an identifier, it looks for the identifier in the evaluation environment and substitutes the value in for the id at the time of use.

Joining these two fixes yields a reduction relation that reduces from a tuple (C, E, K) to another tuple (C, E, K) and is thus named the *CEK machine*. The CEK machine is a standard implementation for interpreters of lambda-calculus style languages.

Felleisen and Friedman [23] extended the CEK machine by adding a store, S. This machine is, not surprisingly, called the CESK machine. Van Horn et al. use this machine model as the basis for developing a reconfigurable abstract interpreter [49].

6.2. Practical Machinery for Big-Step Semantics

One of the practical ways that big-step semantics is implemented in a concrete interpreter is using recursive *evaluator functions*. To do this we build a set of functions whose co-recursive structure matches the inductive structure of the language syntax. Given an expression C, that evaluates to value V, containing sub-expressions A and B, the evaluator $EV()$, when applied to C will call $EV(A)$ and $EV(B)$ in the process of computing $EV(C)$. One thing to note when using an evaluator function is that it restricts our big-step semantics to only use functions instead of arbitrary relations. For many practical cases, this is just fine since many concrete languages of interest have deterministic reduction relations. In the case of non-determinism, a simple way to get around this limitation is to have each evaluation function return a list of possible values and then introduce a simple monadic composition to join everything together.

Using big-step semantics to define abstract interpreters requires more explicit instrumentation, due to the fact that big-step semantics combine multiple reduction steps into a single step. Many analyses may do additional processing between each step in the reduction. This makes it easy to instrument small-step semantics for abstract interpretation, because its granularity is at a single step. For big-step semantics we have to break apart the interpreter and add instrumentation between each of the individual steps. However, explicitly doing this breaks the abstraction model of big step semantics, so it creates a bit of a design puzzle.

There are a couple things that need to be done to take a recursive evaluator function and turn it into an analyzer. First, a way is needed to intercept the function's recursive calls and replace them with ones that are instrumented for the analysis, to do things such as widening, etc. This can be done by changing the evaluator function to use *open recursion*. For example, given a recursive function defined by

```
(define (f arg1 arg2 ...)
  body)
```

in which `f` is called somewhere inside of `body`, we can replace the recursive call with an argument-supplied function. With this, the function looks like

```
(define (f rec-call arg1 arg2 ...)
  body)
```

where all of the direct recursive calls to `f` in the first form are replaced by calls to `rec-call` here. To recover the first interpreter from the second, we need to use a *Y-combinator*, which computes the recursive fixed-point of a function. Our Y-combinator, called "fix" is defined as

```
(define ((fix g) arg1 arg2 ...) ((g (fix g)) arg1 arg2 ...))
```

and then `(fix f)` will recover the first version of the function.

In addition to adding instrumentation in between function calls, we also need to correctly propagate interpretation effects within a single evaluation function. This can be done by using *monadic semantics* [18]. This creates is a generic big-step semantic which can be instantiated with a monad of our choice, and allows reconfigurable analyses via the choice of monad.

Mixing open recursion, monadic semantics, and swappable operator domains, creates a very powerful schema for implementing abstract interpreters using big-step semantics. This is the approach taken in the ADI [18] paper, and has been successfully used in the Ibis abstract interpretation framework, which is actively being developed here at Sandia.

6.3. PLT Redex Tool

One powerful tool to work with designing semantics is *PLT Redex* [21], which is a library that is bundled with the Racket language. PLT Redex is built around defining mini languages of expressions and relations on those expressions. Relations can either be general n-ary relations, or specialized binary reduction relations. In the case of reduction relations, PLT Redex provides a number of useful tools to run and simulate these relations. PLT Redex doesn't make any implicit

assumptions about what the semantics are, so it forces the designer to be completely explicit about how their system should operate. This forced explicitness quickly exposes a lot of bugs and edge cases in the design of semantic domains.

At the beginning stages of semantic modeling, we used PLT Redex and found that it did a good job of exposing design issues with our domains. Due to its utility, one idea that we had was to use these PLT Redex designs directly in our experimental interpreters. However, we ran into a limitation of the PLT Redex tool, that it does not support first-class use of its language definitions, which means that the semantic definitions of PLT Redex couldn't be passed around as data values. This was something that we needed to do for generically combining domains, so we abandoned the idea of defining our semantic domains in terms of PLT Redex. PLT Redex is still a very useful tool, however it is not designed for the task that we were looking to accomplish.

7. DOMAIN INTERFACES AND DEPENDENCIES

We use the algebraic theories described in Section 5 to model our domains. Algebraic theories have a decent correspondence to the module systems provided by a number of different meta-languages we might use to implement our mission application (e.g. OCaml), however most languages do not provide good ways to model and verify the laws between the functional terms in an algebraic theory. The algebraic theory used in designing a given domain is expressed through its *interface*.

Domains are allowed to arbitrarily depend on other domains, and this can occur in two different ways. The first is a direct dependence, in which domain A depends directly on a specific domain B. Another, much more common, form is indirect dependence in which domain A depends on a domain that models a given algebraic theory, S. Any domain B, which models algebraic theory S, can be used to instantiate domain A. In the OCaml language, this corresponds to a *functor module* A, which depends on another module with signature S.

```
1 module type S = sig ... end
2
3 module A (B : S) = struct ... end
```

Indirect dependence of modules is what allows for easy abstraction of a particular domain. Given a concrete domain C and one of its abstractions A, in which the interfaces of both of these domains model the algebraic theory S, transforming domain C to its abstraction A is achieved by simply replacing the given module C with the module A in the interpreter. Since both of the modules model the same algebraic theory, S, swapping one domain out for the other is very straight-forward.

Another kind of abstraction operation might involve changing out a set of concrete domains $C_1, C_2, \ldots$ for a set of abstract domains $A_1, A_2, \ldots$. As long as the set of algebraic theories $S_1, S_2, \ldots$ modeled by $C_1, C_2, \ldots$ is a subset of the algebraic theories modeled by $A_1, A_2, \ldots$, then swapping out the set of concrete domains for the set of abstract domains is straight-forward.

The standard way in which dependencies between libraries are managed is via a transitive dependency tree (module A depends on module B, which depends on module C, etc). Anyone familiar with a utility such as GNU Make, has dealt with this type of dependency structure. In addition to standard trees, there may be cases of circular dependencies in these graphs. This is certainly the case with domain composition. For example, the lattice domain of an inductively defined data structure is going to depend on the lattice domains of its individual components. Since the structure is inductively defined, the domains of at least some of its components are going to depend on the domain of the entire structure itself. This gives us a circular dependency between two parametric domains. Although it is a little more complex, these circular cases can be implemented without a problem.

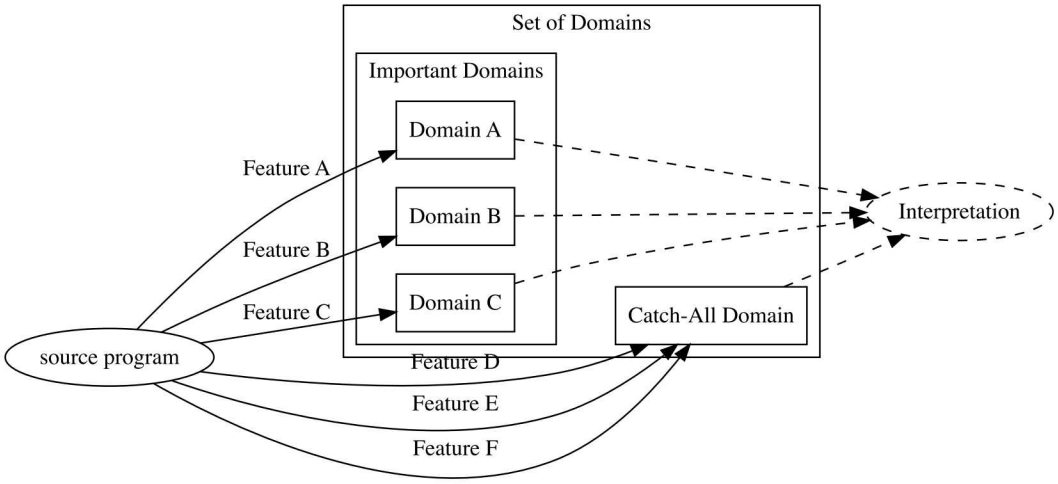


Figure 7-1. catch-all domain

7.1. The Catch-All Domain

One of the main drivers for doing a language-independent analysis is to preserve as much of the semantic information as possible from the original target languages so that it can be utilized by the analysis. Taking this philosophy to the extreme, one could decide to precisely model every feature of every language. The problem with this approach is that there may be an unbounded number of domains, which defeats the purpose of doing language-independent analysis. For a particular analysis, there is certain semantic information that must be precisely modeled, and there is other information that the analysis does not require. It is rare to find an analysis that requires high-fidelity models of *every* language component, and this fact presents a practical way to solve the "unbounded number of domains" problem.

Given an analysis that requires precise modeling of semantic language features F_a, F_b and F_c, modeled by domains D_a, D_b and D_c, we construct our analyzer using the domains D_a, D_b, D_c plus one additional domain, the *catch-all domain*. The catch-all domain is essentially a low-level component that discards all high-level structures. From any target language, we define a transformation from the entire language to this catch-all domain. Then, when we build the analysis, we first transform all structures handled by domains D_a, D_b, D_c into the proper forms. After that, we transform all of the leftovers to the catch-all domain, which preserves behavior, but removes all of the high-level structure. This information loss is only acceptable because the information is unneeded by the analysis.

Since anything can be interpreted by the catch-all domain, we only need to directly instantiate the handful of domains that the given analysis cares about. The catch-all domain will handle the rest.

In Figure 7-1, our analysis needs precise models of features *A*, *B* and *C* for the target language,

but doesn't care about the structure of features D, E, and F. Because of this, we will build an interpreter using four domains, a specific domain for features A, B, and C, and then the catch-all domain will handle all other features present in the language D, E, and F.

8. THE ARRAY DOMAINS

This section considers in detail one particular semantic language feature that is common across many languages – *arrays*. Here we describe the semantics of arrays and identify the various lower level semantic features out of which arrays are constructed.

8.1. Arrays

In the most general sense, arrays are just a specialization of key-value storage to the case in which the key types form a total order. The main consistency condition that key-value stores need to satisfy is known as the "Read-Over-Write" rule, and is summarized in [48] as

$$\begin{aligned} &\forall a : \mathsf{Array}.\forall i, j : \mathsf{Index}.\forall v : \mathsf{Value}. \\ &i = j \implies \texttt{read}(\texttt{write}(a,i,v),j) = v) \wedge \\ &i \neq j \implies \texttt{read}(\texttt{write}(a,i,v),j) = \texttt{read}(a,j) \end{aligned} \tag{8.1}$$

This says that: given that array a' is the result of writing value v to array a at index i, any read from a' at the same index will return the value v, while a read of a' at any other index will return whatever element was originally at that index in array a.

Many languages have the notion of a limited range of allowed indices for arrays. That is, each array has a certain size, and one is allowed to read and write to elements that lie within the size range, but usually some sort of error behavior is triggered during reads/writes to indices outside of that range. Moreover, that behavior is going to change based on the particular programming language used. Because of that, we modify the `read` and `write` operations so that they can return configurable error behavior. So the signatures are now

$$\begin{aligned} \texttt{read} &: \mathsf{Array} \times \mathsf{Index} \to \mathsf{Value} + \mathsf{ConfigurableError} \\ \texttt{write} &: \mathsf{Array} \times \mathsf{Index} \times \mathsf{Value} \to \mathsf{Array} + \mathsf{ConfigurableError} \end{aligned}$$

Two other operations that come up relatively frequently in many programming languages is array allocation and deallocation. Because of this, we add two more operations to our theory

- `array-alloc`$(size) \to a$, where a is a new array of size *size*
- `array-dealloc`$(a) \to a'$, where a' is the deallocated array specifier

The consistency rules relating these functions are as follows

- the read/write consistency conditions as mentioned above
- for the valid index set I of array a, $\forall i \notin I.\ \texttt{read}(a,i) =$ configurable error behavior $\wedge\, \texttt{write}(a,i,-) =$ configurable error behavior
- $\forall a$ s.t. $a = \texttt{dealloc}(a').\, \texttt{read}(a,-) =$ configurable error behavior $\wedge$ $\texttt{write}(a,-,-) =$ configurable error behavior

The general construction of the array domain is as follows. Each array is modeled as a mutable key-value store, in which each element of the array is given a unique storage identifier. This storage identifier does not, necessarily, correspond to the element's index in the array. This is because there may simultaneously exist multiple slices of an array floating around in a program, and each element of the data store may correspond to a different index in each of those array slices. To accommodate this difference in indexing, each array reference is equipped with an indexing function, which will take a given index and transform it into its appropriate storage-identifier. This mechanism allows for easy, in-place, modification of an array's data-store via multiple array references that use different indexing schemes.

The array domain has a single datum that it exports

- `array-ref` : storage-ref $\times$ indexing-function

storage-ref is a mutable reference to the array's data store, which is a key-value data store where each datum is assigned a unique storage id. The `indexing-function` transforms the array index into the storage identifier for that particular element.

8.2. Decomposing Array Operations into Smaller Domains

Sample programs using arrays also depend on other semantic features, including numerics and arithmetic, key-value storage, name binding, and mutable references. By explicitly calling out these lower-level features out of which arrays are constructed, we can maximize code reuse for other language features (e.g., objects, exceptions, etc.).

These four array operations are relatively high-level and can easily be decomposed into operations on smaller domains. Here, we will tour the semantics of these domains and see how these domains can be assembled to build our full array domain.

8.2.1. Arithmetic Domain

This is standard arithmetic that you would find in nearly every language with addition, subtraction, multiplication, and division, with the proper laws about inverses, distributivity, and division by zero.

Each of the four arithmetic operations follow the signature *binop*, which is:

$$\texttt{binop} : \mathsf{Index} \times \mathsf{Index} \to \mathsf{Index}$$

In this particular model, the division-by-zero case is left undefined. The current implementation of the system uses the arithmetic operators of the host language (Racket/Scheme), so division-by-zero will be handled as an exception in that language.

8.2.2. Key-Value Storage

This domain provides key-value storage and lookup as would be found in some sort of hash structure. The operations are:

- `kv-new`
  Create a new empty store.
- `kv-get` : KV-store $\times$ List[Key] $\rightarrow$ List[Value] + Error
  Return a list of values populated by the data in the store corresponding to the key arguments. Returns error values if the keys are not present in the store.
- `kv-set` : Store $\times$ Key $\times$ Value $\rightarrow$ Store
  Returns a store that is the input store, but with the *Key* set to the *Value*.

This domain follows the same "Read-Over-Write" consistency rule 8.1 that the full array abstraction needs to follow.

8.2.3. Lambda and Name-Binding Domain

This domain provides standard name-binding, lambda expression, and function application operations as would be found in lambda calculus. This domain has a standard lambda expression as a data type

- Lambda $\triangleq$ Arg $\ldots \times$ Body

and has the following operations:

- `lam-apply` : Lambda $\times$ Arg$\ldots \rightarrow T$
  Applies a lambda expression, of type $S_1 \times S_2 \times \ldots \times S_n \rightarrow T$, to the given arguments, of type $S_1, S_2, \ldots, S_n$, to produce an output of type T.
- `lam-let` : (Key $\times$ Value)$\ldots \times$ Body $\rightarrow T$
  This is the standard "let binding" expression for languages. Given that the Body has free variables corresponding to all of the Key values provided, it will bind the corresponding *Value* values to each of these keys and then evaluate the *Body* with these name bindings. Note, that this is accomplishing the same objective as *lam-apply*, however it provides syntax that is preferable in a number of situations. In the current implementation, this form just desugars to a `lam-apply`.

- `lam-var` : Key $\rightarrow$ Value
  This returns the value that is currently bound to the Key. If Key is not bound, the behavior is undefined. The current implementation in Racket/Scheme uses a native hash under the hood, so it will throw a language exception if an unbound name is accessed.

Bindings visible within nested forms

$$\text{let } exp = \text{ the expression } \texttt{lam-let}((\ldots v_1 = x_1 \ldots), \texttt{lam-let}((\ldots v_2 = x_2 \ldots), \texttt{lam-var}(v_1)))$$

$$\text{then } exp = \begin{cases} x_1 \text{ if } v_1 \neq v_2 \\ x_2 \text{ if } v_1 = v_2 \end{cases}$$

In prose, this rule says that a name-binding form that happens within another name-binding form can have two behaviors

1. If the inner name-binding form does not bind a name bound by the outer form, then a value lookup (using `lam-var`) will return the value bound by the outer form.
2. If the inner name-binding form binds a name that was already bound by the outer form, the inner binding *shadows* the outer binding, and the value lookup will return the new binding for the scope of the inner body. Outside the scope of the inner body, the name is still bound to the outer value.

For a practical implementation, this domain needs an additional *Evaluation Environment*, in addition to the standard input expressions. The environment will contain some sort of persistent key-value lookup (whose specifics can be chosen by the implementation). When a new *lam-let* form is encountered, the implementation will add the new bindings to the environment. Once a *lam-let* form is exited, the implementation will revert the evaluation environment back to its previous state, before the new bindings were added.

8.2.4. Mutable Reference Domain

This domain provides mutable value storage and references to that storage. Unlike the standard value semantics, any modification of a stored value from any one of its references is immediately visible from all of the other references.

There is one data type provided by this domain:

- Ref $\triangleq$ Id
  A reference to the mutable data, which is identified through a unique identifier.

There are three different operations provided by this domain:

- `makeref` : Expr $\rightarrow$ Ref
  Evaluates *Expr* and creates a new mutable reference out of its value.

- `deref` : Ref-Expr $\rightarrow$ Value
  Given *Ref-Expr*, that evaluates to a *Ref* value, evaluates *Ref-Expr* and returns the value stored at this reference.
- `assign` : Ref-Expr $\times$ Value-Expr $\rightarrow$ 1
  Assigns the result of evaluating *Value-Exp* to the reference generated by evaluating Ref-Expr.

This domain also has an evaluation environment that will be passed through the evaluator, and this environment is a key-value store, mapping each internal reference ID to its data.

Adding the mutable reference domain to the set of languages brings in some additional subtleties that need to be addressed. Up until this point, each operation was *referentially transparent*. A *referentially transparent* expression/operation can be replaced, in a program, by its result and the behavior of the program will not change. Stated a different way, for a referentially transparent operation, the entirety of its effect on the program is contained in its return value. Referential transparency is a property that can greatly simplify program structure and is often desirable. It enables expressions to be evaluated in any order, as long as the values needed to evaluate an expression are available by the time of its evaluation. In this case, one can think about the evaluation of each function in terms of a partial order in the time domain, in which, for a function $f(e_1, e_2, \ldots, e_n)$ $\forall i.\ e_i \sqsubseteq f$, where $\sqsubseteq$ means *is evaluated before*. All that f requires is that the values $v_1, v_2, \ldots, v_n$ generated by evaluating its argument expressions $e_1, e_2, \ldots, e_n$ are available by the time it is evaluated. The structure of this partial order corresponds very closely with the structure of the syntax tree, so it is very easy to generate and reason about.

The `assign` operation is *not* referentially transparent because it has a system effect that is not contained in its return value, it modifies something in the reference environment. `assign` actually returns a value with no information, something of the *Void* type.

As soon as non-referentially transparent operations enter into the language, the order of evaluation for expressions becomes much more important and the partial order described above becomes much more densely populated, and no-longer corresponds to the structure of the syntax tree. Instead of attempting to define the partial ordering minimally in order to capture the non-referentially transparent dependencies, many programming models just approximate it with a *total order*, which gives a complete order of evaluation for every form in the programming language.

For example, using the expression $f(e_1, e_2, \ldots, e_n)$, a programming model that uses only referentially transparent forms (such as the Haskell language) need not define a precise order to evaluate the expressions $e_1, e_2, \ldots, e_n$ because referential transparency prescribes that they have no effect on each other. However, in a language with side-effectual expressions, the order of evaluation of $e_1, e_2, \ldots, e_n$ *does* matter, because the evaluation of any of those expressions could cause system side-effects that may change the evaluation of the other expressions. Because of this, the programming model must give a precise order in which to evaluate the expressions in this case.

In our implementation, we have left most of this behavior formally unspecified, instead allowing the programming model to inherit its properties from the meta-language (in this case

Racket/Scheme). In the case of the Racket language, function arguments are evaluated from left to right, which reflects the behavior of many target languages of interest to our work.

8.2.5. Language-Specific Array Operations

In our work, we found three main primitives that could be used to describe a wide-range of language-specific array behavior: indexing, slicing, and error types. We focus here on comparing the semantics of these primitives in C and Python.

In C, arrays of size N are indexed in the range $[0, N-1]$. Any attempt to index the array outside this range will give undefined behavior according to the language standard. We can *slice* C arrays by creating a pointer to some element, with index $M \in [0, N-1]$ in the middle of the array. Indexing this pointer as an array, valid indexing is in the range $[-M, N-M-1]$ and any attempt to index outside of that range will yield undefined behavior.

In Python, an array of size N can be indexed in the range $[-N, N-1]$, in which non-negative indices index from the front of the array and negative indices will index backwards from the back of the array. Any attempt to index the array outside of this range will generate an exception. Python arrays can be sliced using a slice triple $A : B : C$, which will generate a view of the array containing indices $i \in [A, B)$ s.t. $(i - A) \bmod C = 0$ for the case in which A and B are non-negative indices (informally, C representing the stride of the slice). In the case that A or B are negative, they are essentially converted to their corresponding positive indices and slicing is done as described above. If C is a negative number, the slicing is performed in the same way, but "backwards", and A must be a higher index than B, according to the rules of Python array indexing. The elements in this negative slice will be in the opposite order from that of the original array. For valid indices A, B, if any of these conditions on A, B, C are violated, the slicing operation will just return an empty slice.

Generically, any given target language array theory would need to provide the following:

- One or more error types, corresponding to different error behavior in the language.
- `array-default-index-fn` : $\mathsf{Size} \to (\mathsf{Index} \to \mathsf{Storage\text{-}ID})$
  Generates the standard indexing function, which will transform index values into unique storage identifiers for the elements in the array store.
- `array-slice` : $\mathsf{Array} \times \mathsf{Slice\text{-}Tuple} \to \mathsf{Array} + \mathsf{Error}$
  Generates a new array reference whose indexing function is the requested slice of the original array. The array elements will still have their same unique storage identifiers, however the mapping from array indices to storage identifiers will change with the slice.

Each of these language-specific array domains may also depend on one or more error types, which are incorporated into the global syntax tree.

8.3. Currently Unsupported: Size-Changing Operations

There are a number of languages (Python & Perl included), that have built-in array operations that change the size of an array, such as `push` and `pop`. Though our implementation doesn't currently support these operations, this section considers how these operations could be added.

The underlying storage mechanism is easily extensible to operations that modify the length of the array, since each element is just stored under a unique identifier. However, the design needs to ensure that there are no collisions between storage identifiers for any two distinct elements for all arrays that reference the same underlying storage. This is a property that a naive identifier assignment strategy may fail to satisfy.

Another issue that would need to be addressed is the fact that push/pop operations modify more about the array than just its contents. The current design only uses mutable references to model the actual data that lives in the array. If the structure of the language is such that changes to an array's index set only need to be valid for the current lexical scope, then these changes can just be accomplished by rebinding the array identifier to the updated array object. However, if these changes need to be valid outside of the current lexical scope, then another layer of indirection needs to be added, and the array objects themselves must be stored in mutable references as well, instead of being directly bound to by identifiers.

8.4. Constructing the Array Domain

Finally, we demonstrate how to assemble the complete array processing domain from these individual domains.

The array uses the *Key-Value Storage* domain as its way to store individual array elements via their unique storage identifiers. It uses the *Arithmetic* domain for any of the mathematics necessary to perform indexing and slicing calculations. The array domain doesn't use the lambda/name-biding domain directly in its operations, however this domain is very useful for analyzing non-trivial programs that use arrays. The name-binding domain also requires the use of an evaluation environment, so programs that use arrays will use this environment. The array domain uses the *Mutable Reference* domain to link together multiple views/slices of the same array. In many languages, views/slices of an array all modify the same underlying data, and the mutable reference domain allows exactly this kind of behavior. The mutable reference domain also uses an evaluation environment, which stores the actual modifiable values. The *Language-Specific* array operations provide the language-specific indexing and slicing operations for the arrays. Finally, we use a *Value* domain to provides the necessary monadic "return" function for working with our particular monadic interpreter implementation.

8.4.1. Example: Read Function

This is a sample implementation of the `read` function on arrays, which combines operations from the underlying domains.

```
['(read ,array-exp ,index-exp)
     ;; get the array reference
     (let*-values ([(arr-ref _ pass1) (rec-call array-exp env pass)]
                   [(index _ pass2) (rec-number? index-exp env pass1)])
       (match arr-ref
         ['(array-ref ,kv-ref ,ix-trans)

          ;; dereference the kv storage and transform the index
          (let-values ([(kv-store _ pass3) (rec-call '(deref (value ,kv-ref)) null pass2)]
                       [(tr-ix) (ix-trans index)])

            ;; if we have an indexing error, then return that
            ;; error. Otherwise, pull out the corresponding element
            ;; from the kv store
            (match tr-ix
              [(index-error errorval) (values errorval null pass3)]

              ; call recursively and pull out the retrieved element from the
              ; list
              [_ (let-values ([(vlist _ pass4)
                               (rec-call '(kv-get ,kv-store (,tr-ix)) null pass3)])
                   (values (car vlist) null pass4))]))]))]
```

This will perform the following steps:

1. Recursively call the general evaluator to generate the array reference value.
2. Recursively call the general evaluator to generate the index value.
3. Call into the mutable reference evaluator to dereference the storage ref contained in the array ref, returning the data associated with that reference.
4. Apply the index transformer, which was stored inside the array reference, to generate the storage id for the given array element.
5. Call into the key-value evaluator to retrieve the array element data stored under the id returned by the transformer.

8.4.2. Sample: Write Function

This function uses the operations from the underlying domains to write a value to an array and then evaluate code after that write has taken effect

```
['(with-write (,(? symbol? name) ,index-exp ,new-value-exp)
                ,@(list others ...))
     (let*-values ([(arr-ref _ pass1) (rec-call '(lam-var ,name) null pass)]
                   [(index _ pass3) (rec-number? index-exp null pass1)])

       (match arr-ref
         ['(array-ref ,kv-ref ,ix-trans)
```

```
8           (let-values ([(kv-stor _ pass4) (rec-call `(deref (value ,kv-ref))
    null pass3)]
9                        [(tr-ix) (ix-trans index)])
10
11            ;; check whether we have an index error
12            (match tr-ix
13              [(index-error errorval) (values errorval null pass4)]
14              [_
15               ;; first, set the value of the array
16               (let*-values ([(modarr _ pass5)
17                              (rec-call `(kv-set
18                                          ,kv-stor
19                                          ((,index ,new-value-exp)))
20                                        null pass4)]
21
22                             ;; second, install the storage in the reference
23                             [(_1 _2 pass6)
24                             (rec-call `(assign (value ,kv-ref) (value ,
    modarr)) null pass5)]
25
26                             ;; third, call the rest of the
27                             ;; statements with the modified array
28                             [(retval _ pass7)
29                              (rec-call others null pass6)]
30
31                             ;; last, reinstall the unmodified array
32                             [(_1 _2 pass8)
33                              (rec-call `(assign
34                                          (value ,kv-ref)
35                                          (value ,kv-stor))
36                                        null pass7)])
37
38                 ;; return the retval with the most recent pass
39                 (values retval null pass8))])))])))]
```

`with-write` performs the following operations:

1. Recursively call the general evaluator to generate the array reference value.
2. Recursively call the general evaluator to generate the index value.
3. Call the mutable reference evaluator to dereference the array data. store.
4. Pull the indexing function out of the array and applies it to the supplied index to generate the storage id.
5. Call the key-value evaluator to set the storage id to the new value. This call will then recursively call the general evaluator to evaluate the expression that generates the new value to store.

9. RECURSIVE INTERPRETER DESIGN

In order to use the domains described in Section 8 to implement a mission application, we need to construct an abstract interpreter to statically "run" the target application, calculating at each program point all possible abstract values needed by the mission application. This section first addresses issues of interpreter design affecting modularity needed to enable construction of multiple mission applications from a common set of components.

9.1. How open-recursion works

Implementing an interpreter according to the big-step semantics of a language typically involves a recursively defined evaluator function. For example, consider a language defined over the following syntax tree:

- `(intval a)` represents a literal integer value
- `(int-add a b)` represents integer addition
- `(int-sub a b)` represents integer subtraction

It is then easy to represent complex expressions by arbitrarily nesting these three forms, for example:

```
1 (int-add (int-sub (int-val 5) (int-val 2)) (int-val 7))
```

would represent the math expression $(5-2)+7$. To evaluate an expression such as this, one of the simplest ways to go about it is to use a recursively-defined evaluation function that has the same recursive structure as the syntax tree. In this case, the evaluator is defined as:

```
1 (define (int-eval expr)
2   (match expr
3     [('int-add a b) (+ (int-eval a) (int-eval b))]
4     [('int-sub a b) (* (int-eval a) (int-eval b))]
5     [('intval a) a]))
```

`intval` is the base case of the recursion, and it will return the number wrapped in the `intval` domain. Note that both the addition and subtraction cases (lines 3 and 4) take expression trees as arguments. Because of this, they need to recursively call the evaluator to generate concrete values from these expression trees before they are able to do their own arithmetic.

In its current form, the `int-eval` evaluator can only be used for the language containing exactly the three forms `intval`, `int-add`, and `int-sub`. However, it is possible to modify this evaluator so that it can be combined with other evaluators to process larger languages. Instead of

recursing directly on `int-eval`, the recursive call itself becomes a parameter. This modification produces an evaluator of the form

```
(define ((int-eval rec-call) expr)
  (match expr
    [('int-add a b) (+ (rec-call a) (rec-call b))]
    [('int-sub a b) (* (rec-call a) (rec-call b))]
    [('intval a) a]))
```

This style of function definition is commonly referred to as *open recursion*.

To recover the original interpreter from the open-recursive version, one needs to use a function called a *fixed-point combinator*, which is defined as.

```
(define ((fix f) x) ((f (fix f)) x))
```

The first argument of `fix` is an open-recursive function whose first argument is its recursive call and second argument is the standard data value. `fix` then calls f on datum x with `fix x` as the recursive call. What this does is provide f with a unbounded chain of recursive self-calls to use until it hits its base case. This definition works because the application of `fix` in the function body is done lazily, so it will only generate calls as deep as is required by f.

Evaluating `(fix int-eval)` will yield the original integer arithmetic interpreter from the open-recursive version.

9.2. Combining Two languages

Imagine that there is a similarly-defined interpreter for floating point numbers

```
(define ((float-eval rec-call) expr)
  (match expr
    [('float-add a b) (+ (rec-call a) (rec-call b))]
    [('float-sub a b) (* (rec-call a) (rec-call b))]
    [('floatval a) a]))
```

How would one compose this with the integer interpreter to create an interpreter that performs addition and subtraction on both `ints` and `floats`? The open-recursive definition of these interpreters makes this composition very easy to do.

The first step is to distinguish which forms belong to the `int` interpreter and which ones belong to the `float` interpreter. To do this, one creates a predicate function for each interpreter that will return `true` when the given form matches something handled by the interpreter and `false` otherwise. These predicate functions look like

```
(define (int-form? expr)
  (match expr
    [('int-add _ _) #t]
    [('int-sub _ _) #t]
    [('intval _) #t]
    [_ #f]))

```

```
(define (float-form? expr)
  (match expr
    [('float-add _ _) #t]
    [('float-sub _ _) #t]
    [('floatval _) #t]
    [_ #f]))
```

Next, these predicate functions are used by a top-level dispatcher function, which determines which of the evaluators should handle the given form. The dispatcher looks something like this

```
(define ((top-eval rec-call) expr)
  (match expr
    [(? int-form?) ((int-eval rec-call) expr)]
    [(? float-form?) ((float-eval rec-call) expr)]))
```

The dispatcher function also uses open-recursion, which means that it could, subsequently, be combined with another evaluator function. To recover the closed-form interpreter, one just has to put it into the fixed-point combinator.

```
(define final-eval (fix top-eval))
```

and the result is an evaluator that will perform addition and subtraction on both `ints` and `floats`.

The important thing to note is that in `top-eval` the recursive call argument, `rec-call`, is passed as the recursive call to both `int-eval` and `float-eval`. Since `int-eval` and `float-eval` are called directly from `top-eval`, these calls are effectively chained together, and passing the `rec-call` argument from `top-eval` into `int-eval` and `float-eval` creates a recursive loop out of the chain.

Without taking the fixed-point of `top-eval`, the call graph is shown in figure 9-1, in which the recursive calls are still dangling, since it hasn't been supplied as an argument to the function yet.

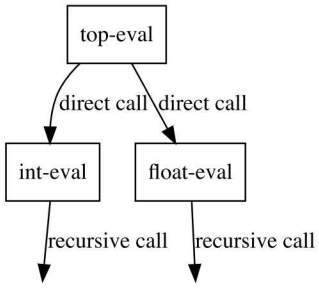


Figure 9-1. `top-eval` with recursion still open

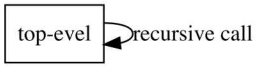


Figure 9-2. `top-eval` in fixed-point combinator

Once `top-eval` is put into the fixed-point combinator, it will recursively call itself, and the call chain will follow figure 9-2.

And zooming in, the internal call structure of the `top-evel` fixed-point is shown in figure 9-3, in which each recursive call to `top-eval` will dispatch to either `int-eval` or `float-eval` depending on which sub-interpreter predicate matches.

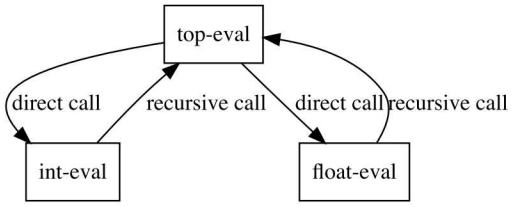


Figure 9-3. Inner view of `top-eval`

9.3. Why Evaluation Environments are Useful

Operations such as arithmetic are very efficient to implement in a direct manner, in which one simply substitutes the result of evaluating an expression for the expression itself. However, there are other types of operations which are less suited for this kind of pure-substitution approach. Consider the operation of name-binding (see Section 8.2.3), in which one takes a value and attaches a name to it, which then can be referenced from anywhere within the (possibly extensive) code within scope of the binding. For example, the code

```
(let ([x 5] [y 6])
  (+ x y))
```

binds `x` to equal 5 and `y` to equal 6 in the body of the expression, which means that this code will evaluate to 11.

Using the pure-substitution approach, a `let` form is evaluated by substituting the bound value for any free occurrences of the identifier in the body, and then evaluating the substituted body. This approach requires an additional scan over the entire body of the `let` form to perform the substitution. In the example above, this doesn't result in much additional work, but this approach can easily become very slow once the body of the `let` form (i.e., scope of the binding) gets large.

In addition, for every nested binding, the evaluator needs to repeatedly re-scan the same code to perform required substitutions.

There is an easy way around this problem, and that is to augment the evaluator with an *evaluation environment*. This is an extra piece of state that the evaluator recursively passes to itself in addition to the expression tree. With the environment, the signature of a closed evaluator will look like `(eval expression environment)`. In the name-binding case, the evaluation environment is a key-value store (see Section 8.2.2) that maps bound names to their values. Whenever the evaluator encounters a `let` form, it updates the key-value store to contain the new bindings. Whenever it encounters a name which must be evaluated, it looks up the corresponding value in the environment. When the evaluator exits a `let` form, it will just revert the key-value store to its old state, from before the `let` form. With this approach, the evaluator only has to do a single pass over the code instead of making a substitution pass over the code every time it encounters a `let` form.

Another domain that greatly benefits from the use of an evaluation environment is mutable references (see Section 8.2.4). For example, let *a* equal the mutable reference to storage location *stor*. The evaluator can set a new mutable reference, *b*, equal to *a*, which means that *b* will also reference the storage location *stor*. Any change to the value via the reference *a* will be automatically updated when read from reference *b* and visa-versa. Implementing this behavior via pure substitution is possible, but is difficult and inefficient. This scheme, using an evaluation environment, is much more efficient to implement.

9.4. Extending Evaluator composition to include Environments

Below are evaluator stubs for the previously-mentioned name binding and mutable reference domains described in Section 8.

```
(define ((eval-namebind rec-call) expr env)
  (match expr
    [('let etc...)
      ;do something ...
      (rec-call new-expr new-env)
      ;do something ...
      ]))

(define ((eval-ref rec-call) expr env)
  (match expr
    [('ref-set etc...)
    ;do something with call (rec-call new-expr new-env)
    ]
    [('ref-get etc...)
    ;do something with call (rec-call new-expr new-env)
    ]
    [('ref-new etc...)
    ;do something with call (rec-call new-expr new-env)
    ]))
```

Each of these evaluators takes both an expression, *expr*, and an environment *env*. In addition, each recursive call made by these evaluators will also include an evaluation environment, which may be modified from the original `env` argument to reflect updated information. For example, a `let` form will add any new name bindings to the environment and then recursively call the evaluator on the body expression.

Recovering a standard evaluator from one of these open-recursive evaluators that take an additional environment uses the same type of Fixed-Point Combinator as before, but with an additional argument for the environment.

```
(define ((fix f) expr env) ((f (fix f)) expr env))
```

9.5. Chaining Evaluators with Different Environments

For two open-recursive evaluators that use the same environment, composing them using a top-level dispatcher is relatively straightforward. Each function, including the dispatcher will pass the evaluation environment around, in addition to the expression tree. However, many cases arise in which one wants to compose evaluators that take different evaluation environments, and that can be done using the idea of *lenses*, which was originally presented in [25], and has been the subject of much interest during the past decade.

The mathematical formulation of lenses can get quite deep, so all that is presented here is a short, intuitive, motivation to the concept. Imagine a data structure S, with fields $S.a$ and $S.b$ and there exists two client methods, F_a and F_b, that desire to access and/or modify the fields $S.a$ and $S.b$ respectively. However, there exist more fields in structure S, and the precise locations for fields $S.a$ and $S.b$ within S are not known a-priori. In addition, one may also desire the client methods F_a and F_b to work with structure T, with fields $T.a$ and $T.b$, which contain the same kind type data as $S.a$ and $S.b$, but might be located in a different part of the structure and require different kinds of access. The desire is to create some sort of generalization of data access that F_a and F_b can use so they can be used on a many different data structures.

A solution to this problem is to hide the internal structure of S from both F_a and F_b and then to provide, for each field a, b, two accessor functions `set` and `get` such that

$$\begin{aligned} set &: S \times \mathsf{value} \rightarrow S \\ get &: S \rightarrow \mathsf{value} \end{aligned}$$

With this formulation, neither F_a and F_b need to know the specifics of how to access their respective fields, they can just use the accessor functions.

This idea allows for the combination of interpreters that use different evaluation environments. The environment for the top-level dispatch function will be a tuple of all individual environments required by each of the interpreters.

The next problem that comes up is that each of the individual interpreters expects its own individual environment, yet they need to be able to recursively call back into the dispatcher

function, which requires the complete tuple of evaluation environments. To facilitate this, a third argument, *pass*, is added to the evaluator functions. This polymorphic argument is used to pass all other environments through the evaluator. The evaluator essentially ignores the *pass* argument, except to pass it to, and receive it from, any recursive calls that it makes.

With these additional parameters added, every evaluator function will have the standard signature

$$\texttt{eval} : \mathsf{expression} \times \mathsf{environment} \times \mathsf{pass} \rightarrow \mathsf{expression} \times \mathsf{environment} \times \mathsf{pass}$$

The last piece of this puzzle is the lifting and unlifting of individual evaluation environments out of the tuples. The top-level dispatcher needs to, somehow, unpack a particular sub-interpreter's environment from the tuple before calling it and then pack the updated environment back into the tuple after a recursive call or return back into the dispatcher. These operations are done by the functions `lift` and `unlift`.

Here is example code for both `lift` and `unlift`

```
(define (lift id env pass)
  (let ([newenv (hash-ref env id)]
        [restenv (hash-remove env id)])
    (values newenv (hash-set restenv 'outside pass))))

(define (unlift id env pass)
  (let ([newpass (hash-ref pass 'outside)]
        [newenv (hash-remove pass 'outside)])
    (values (hash-set newenv id env) newpass)))
```

In these two functions, each evaluation environment is assigned a unique id, and the tuple of environments is implemented as a Scheme hash, in which each environment is stored under its unique id. The `lift` function will pull out the requested environment from the hash and store the old `pass` parameter in the environment hash under the id `'outside`. It then returns the extracted environment as the new "environment" parameter and returns the old environment, with the new `'outside` mapping, as the new "pass" parameter. Unlift does the exact opposite, replacing `env` into `pass`, pulling the old `pass` out of the old environment, and then returning the replaced environment hash as "environment" and the extracted `pass` as "pass".

Figure 9-4, is the modification of Figure 9-3 to include the lifting and unlifting of environments. Instead of working with evaluators for integer and floating point expressions, this figure uses an evaluator for mutable reference and an evaluator for name-binding, both of which use their own separate evaluation environments. Instead of of top-level dispatcher calling the individual evaluators directly, it will first call the appropriate environment lifter and then call into the evaluator. For a recursive call back to the top dispatcher, the individual evaluators will first call the appropriate unlifter and then call back into the top-level dispatcher. The individual evaluators don't know that they need to call the unlifters, so this is accomplished by pre-composing the unlifter onto the top-level dispatcher and then passing that composed function into its respective evaluator, as the recursive call.

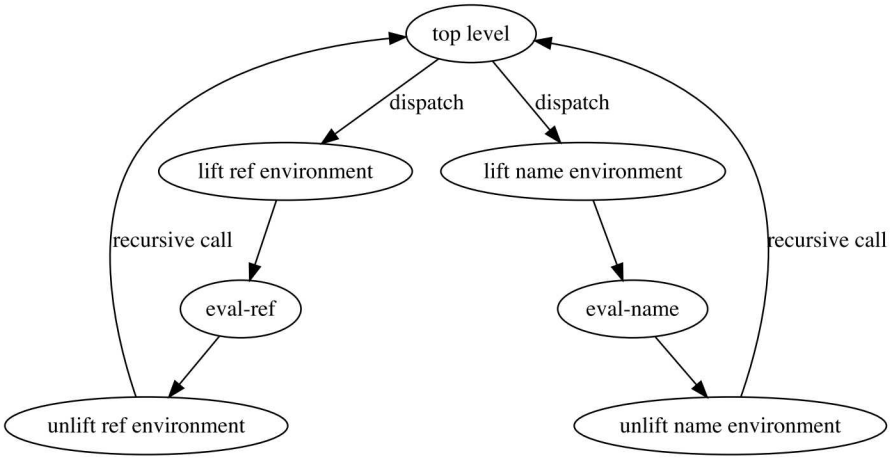


Figure 9-4. Recursion with Environment Lifting

10. A MULTI-TARGET-LANGUAGE UNINITIALIZED VARIABLE ANALYSIS

This section describes an experiment conducted to validate our hypothesis of being able to construct cross-language mission applications by leveraging a set of semantic features and a modular abstract interpreter. This section describes in detail a multi-target-language analysis using the components and structures described in the sections above.

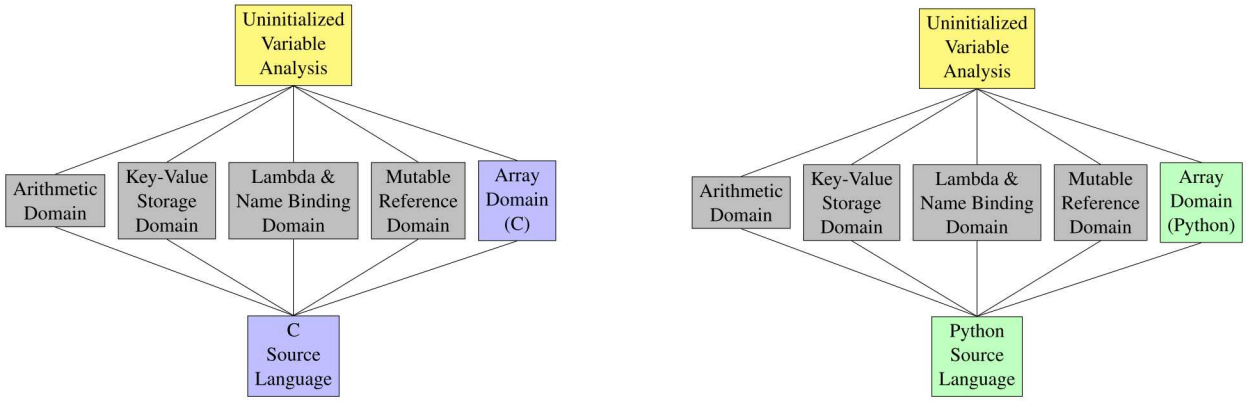
In abstract interpretation, the results of the analysis is itself captured as an abstract value calculated along with all the other program values. In our experiment, we perform a simple *uninitialized value analysis* with arrays. In an uninitialized value analysis, the initialization state of each variable is tracked using an abstract domain, and uses of potentially uninitialized values are detected.

The primary test program that we used to validate the capability of the composed array domain was a simple *uninitialized value analysis* with arrays. To simplify implementation, we did not implement a front-end parser; rather the input to our tools is an S-expression representation of the target program semantic structure.

10.1. Base Experiment with Arrays

The main template of the target program for this analysis is in Listing 10.1.

The language is fairly primitive, so a step-by-step explanation is in order. Starting at line 15, the program allocates an array of size 3 and then initializes each of its elements to -1 (which is a



(a) Domain structure for C semantics **(b) Domain structure for Python semantics**

Figure 10-1. Structure of domains for the uninitialized variable analysis

Listing 10.1 Starting code for the array analysis.

```
(define test-prog
  '(lam-let ((truth-value (value #t)))
   (lam-let

    ; after the control has merged back together, this is what we're
    ; going to do to start moving things up the lattice
    ((rest (value
            (lam-lambda (arr)
                        (lam-let ((a (+ (array-get arr (value 0)) (value 0)))
                                  (b (+ (array-get arr (value 1)) (value 0)))
                                  (c (+ (array-get arr (value 2)) (value 0))))
                          (lam-var XXX))))))

   ; set the array to uninitialized (negative) values
    (with-array-alloc (arr (value 3))
      (with-array-set (arr (value 0) (value -1))
        (with-array-set (arr (value 1) (value -1))
          (with-array-set (arr (value 2) (value -1))
           (if-then-else
            (lam-var truth-value)

            ; true case, set elements 1 & 2 to positive
            (with-array-set (arr (value 1) (value 1))
              (with-array-set (arr (value 2) (value 1))
                (lam-apply (lam-var rest) ((lam-var arr)))))

            ; false case, set elements 0 & 2 to positive
            (with-array-set (arr (value 0) (value 1))
              (with-array-set (arr (value 2) (value 1))
                (lam-apply (lam-var rest) ((lam-var arr))))))))))))
```

surrogate for an uninitialized value in this program). Next, on line 19 the program branches using an `if-then-else` form, and branches on `truth-value` which is defined at the very beginning of the program on line 2. On the `true` branch array elements 1 & 2 are initialized to contain a value of 1 (lines 23 and 24), and on the false branch array elements 0 & 2 are initialized to 1 (lines 28 and 29). At the end of each branch, lines 25 and 30 each pass control to the final stage of the program, defined on line 7, which pulls each value out of the array, runs each value through a binary operation (addition of 0) and then returns one of the results (`a`, `b`, or `c`, whatever is provided for `XXX` on line 12)[1].

In a concrete interpreter, the program behaves as follows:

- If `truth-value` is set to `true` on line 2, then the final values `b` and `c` are fully initialized, but value `a` depends on uninitialized data.
- If `truth-value` is set to `false` on line 2, then the final values `a` and `c` are fully initialized, but value `b` depends on uninitialized data.

The purpose of the analysis is to determine whether any value manipulations (such as arithmetic) in the program `may` depend on uninitialized values (represented here by negative values). To perform an all-paths analysis we need to switch to an abstract domain designed to track this information. This is easly implemented as a two-element lattice with values "definitely not negative" and "possibly negative" in which definitely not negative $\leq$ possibly negative in the lattice.

Running the example with this domain, and defining the value bound to `truth-value` to be unknown (e.g., $\perp$ in an abstract interpretation domain lattice), the results are as follows:

- `a` is "possibly negative" since it depends on the value at array index 0, which is undefined in the true branch, but defined in the false branch.
- `b` is also "possibly negative" since it depends on the value at array index 1, which is defined in the true branch, but undefined in the false branch.
- `c` is "definitely not negative", since it depends on a value that was defined in *both* the true and false branches.

10.2. Extending the Experiment with Array Slicing

The sample program in the previous section used array semantics that hold true for multiple languages, in particular C and Python. In this section, we extend the experiment with a semantic feature that differs between these two languages, and demonstrate how the vast majority of our interpreter infrastructure and abstract domains are reused across target-languages.

We slightly modify the above program to derive the program in Listing 10.2.

[1] Although one could imagine an uninitialized variable analysis which ignores identity operations (e.g., $+0$, $*1$, etc.). Our current mission application defines such identity operations as errors

Listing 10.2 Extending the base array analysis code with an array slicing operation.

```
(define test-prog
  '(lam-let ((truth-value (value unknown)))
   (lam-let

    ; after the control has merged back together, this is what we're
    ; going to do to start moving things up the lattice
    ((rest (value
            (lam-lambda (arr-arg)
                        (lam-let ((a (+ (array-get arr-arg (value 0)) (value 0)
    ))
                                  (b (+ (array-get arr-arg (value 1)) (value 0)
    )))
                          (lam-var XXX))))))

  ; set the array to uninitialized (negative) values
    (with-array-alloc (arr (value 3))
      (with-array-set (arr (value 0) (value -1))
        (with-array-set (arr (value 1) (value -1))
          (with-array-set (arr (value 2) (value -1))
            (with-array-cslice arr-slc (slice arr 1)
              (if-then-else
                (lam-var truth-value)

                ; true case, set elements 0 & 1 to positive
                (with-array-set (arr-slc (value 0) (value 1))
                  (with-array-set (arr-slc (value 1) (value 1))
                    (lam-apply (lam-var rest) ((lam-var arr)))))

                ; false case, set elements 0 & -1 to positive
                (with-array-set (arr-slc (value -1) (value 1))
                  (with-array-set (arr-slc (value 1) (value 1))
                    (lam-apply (lam-var rest) ((lam-var arr))))))))))))))
```

The main difference between this program and the previous is that it creates a slice of the original array on line 18 with the `(with-array-cslice ...)` form, and then operates on the slice from that point on. In C, *array slicing* is performed by defining a pointer to the middle of the array and then indexing that pointer as an array. In this language, `(with-array-cslice arr offset)` will return a "pointer" to array `arr` at offset `offset` This means that all indices of the array slice map to a constant offset from indices of the original array.

Running the same uninitialized value analysis on this target application using C semantics for array slicing yields the following results:

- `a` is "definitely not negative" because it depends on the value at array index 0, which is set in both branches.
- `b` is "possibly negative" because it depends on the value at array index 1, which is only set in the false branch.
- the false branch also sets the value at array index -1, which maps to index 0 of the original array. This is a valid operation, but does not affect the output of the program.

In order to apply this same analysis to Python, we first note that array slicing semantics in Python are different than in C. Instead of just a single offset, a slice in Python is given by a triple `arr[start:stop:stride]`, and it behaves as explained in previous sections. The form provided by the Python-specific array domain is `(with-array-pyslice slice-name (slice orig-array start end step))`. We can "port" our sample program to Python by merely swapping out line 18 from `(with-array-cslice ...)` to `(with-array-pyslice arr-slc (slice arr 1 3 1))`. This is the same program as above, but now using Python semantics. We then can run the same analysis with our same interpreter, except for substituting the C array domain for the Python array domain, and get the following results:

- `a` is "definitely not negative" because it depends on the value at array index 0, which is set in both branches.
- `b` is also "definitely not negative", and this is due to the fact that Python indexing semantics are different than those of C. Negative indices in Python index backwards from the end of the array, so index 1 and -1 refer to the same element of the slice. Because of this, the value at index 1 in the slice gets set in both branches, yielding a value of "definitely not negative".

It is worth reiterating that to accomplish this, we only need to swap out a single domain, replacing the C-specific array functionality with the Python-specific array functionality, and the analysis runs as before, now aware of Python array specifics. All other domains work for both languages, and the interpreter framework is the same in all cases.

11. FUTURE WORK

There is much research yet to be done in the area of language-independent static software analysis. This work focused primarily on a semantic-driven view of language analysis and a particular line of modular interpreter structures.

In the area of modeling languages as sets of parameterized semantic features which share a large amount of commonality, there is more work to be done. Our work evaluated a small set of features (objects, arrays, and exceptions) for a small set of languages (Javascript, Java, C, Python), though we provide only a few point demonstrations of possibility and did not exhaustively explore even this narrow scope. Additional work is needed to include additional language features, to evaluate the extent to which real-world languages can be completely modeled this way, and to what extent outlier semantics may mitigate the generality of the approach. Further, the target languages we evaluated come from the imperative family. We did not evaluate the extent to which functional or relational languages may be amendable to this approach.

In this work, we implemented a minimal set of domains in order to conduct our key experiments. This minimal approach enabled us to sidestep a wide range of research issues involved in the design and implementation of a useful library of semantics domains, along the lines of the Apron [28] library but extended beyond numerical abstractions.

In the area of modular interpreter structures, there is also more work to be done. While this work demonstrated that a monadic-style definitional interpreter was suitable to this task, static analysis of real-world programs requires very careful design to achieve reasonable scalability in terms of time and space. There are numerous tradeoffs in this area which would need to explored to make real mission applications practical.

Additionally, we did not demonstrate in this work that our modular interpreter structure and the semantic domains can be used to easily take advantage of new analysis strategies and techniques from the academic literature.

12. CONCLUSION

This research activity was an initial exploration into a large, challenging multi-faceted area of language-independent static software analysis. In this project we explored two very specific facets about which we are comfortable drawing conclusions. First, we explored the hypothesis that languages can be conceptualized as a set of common semantic elements, instantiated with different parameters for different languages. The value of such a set of semantic elements would be to facilitate code reuse of similar features across different languages. We conclude that this conceptualization does work in a great many cases, noting that it does not have to work perfectly in all cases to be of utility in an ecosystem of cross-language static analysis tools. By focusing on the semantics of the language features rather than the syntax and on individual elements of semantics rather than each language as a indivisible whole, commonality across languages can more readily be identified.

Second, we explored methods of constructing analysis tools to take advantage of this semantic feature conceptualization, though our explorations were limited to abstract interpretation. While other analysis structures may also provide some benefits, we conclude that the *domains* of abstract interpretation are a natural and effective construct for capturing these semantic similarities across languages. The broadly-applicable structure of the abstract interpreter coupled with the abstract domains instantiated for a given language's semantic features creates the ability to write an analysis that can run across multiple languages. Though we only implemented a very limited number of semantic features and languages, our experiences lead us to full expect that this approach will scale to many other language features and many additional languages. However, in order to fully realize this vision, additional research must be done to determine exactly which semantic language features to target, in what way they should be parameterized to maximally represent as many languages as possible, and how those semantics may compose.

References

[1] Vincenzo Arceri, Michele Pasqua, and Isabella Mastroeni. “An Abstract Domain for Objects in Dynamic Programming Languages”. In: *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part II*. Ed. by Emil Sekerinski et al. Vol. 12233. Lecture Notes in Computer Science. Springer, 2019, pp. 136–151. DOI: 10.1007/978-3-030-54997-8_9. URL: https://doi.org/10.1007/978-3-030-54997-8_9.

[2] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[3] Gogul Balakrishnan and Thomas W. Reps. “Recency-Abstraction for Heap-Allocated Storage”. In: *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*. Ed. by Kwangkeun Yi. Vol. 4134. Lecture Notes in Computer Science. Springer, 2006, pp. 221–239. DOI: 10.1007/11823230_15. URL: https://doi.org/10.1007/11823230_15.

[4] I. Balbaert and S.S. Laurent. *Programming Crystal: Create High-Performance, Safe, Concurrent Apps*. The pragmatic programmers. Pragmatic Bookshelf, 2019. ISBN: 9781680502862. URL: https://books.google.com/books?id=8bMnvAEACAAJ.

[5] *Ballerina*. https://ballerina.io/. Accessed: 2020-08-10.

[6] Kenny Ballou. *Learning Elixir*. Packt Publishing, 2016. ISBN: 1785881744.

[7] Bill Baloglu. “How to find and fix software vulnerabilities with coverity static analysis”. In: *IEEE Cybersecurity Development, SecDev 2016, Boston, MA, USA, November 3-4, 2016*. IEEE Computer Society, 2016, p. 153. DOI: 10.1109/SecDev.2016.041. URL: https://doi.org/10.1109/SecDev.2016.041.

[8] Jeff Bezanson et al. “Julia: A fast dynamic language for technical computing”. In: *arXiv preprint arXiv:1209.5145* (2012).

[9] Gavin Bierman, Martin Abadi, and Mads Torgersen. “Understanding typescript”. In: *European Conference on Object-Oriented Programming*. Springer. 2014, pp. 257–281.

[10] Gilad Bracha. *The Dart Programming Language*. 1st ed. Addison-Wesley Professional, 2015. ISBN: 0321927702.

[11] Cristiano Calcagno et al. “Moving Fast with Software Verification”. In: *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*. Ed. by Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Vol. 9058. Lecture Notes in Computer Science. Springer, 2015, pp. 3–11. DOI: 10.1007/978-3-319-17524-9_1. URL: https://doi.org/10.1007/978-3-319-17524-9_1.

[12] Sang Kil Cha et al. “Unleashing Mayhem on Binary Code”. In: *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 2012, pp. 380–394. DOI: 10.1109/SP.2012.31. URL: https://doi.org/10.1109/SP.2012.31.

[13] *Common Language Infrastructure (CLI)*. 6th ed. ECMA 335. Ecma International. June 2012. URL: http://www.ecma-international.org/publications/standards/Ecma-335.htm (visited on 09/07/2020).

[14] P. Cousot and R. Cousot. "Static determination of dynamic properties of programs". In: *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France, 1976, pp. 106–130.

[15] P. Cousot and N. Halbwachs. "Automatic discovery of linear restraints among variables of a program". In: *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Tucson, Arizona: ACM Press, New York, NY, 1978, pp. 84–97.

[16] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973. URL: https://doi.org/10.1145/512950.512973.

[17] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. "A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 105–118. ISBN: 9781450304900. DOI: 10.1145/1926385.1926399. URL: https://doi.org/10.1145/1926385.1926399.

[18] David Darais et al. "Abstracting definitional interpreters (functional pearl)". In: *Proceedings of the ACM on Programming Languages* 1.ICFP (2017), pp. 1–25.

[19] Paul Deitel and Harvey Deitel. *Swift for Programmers*. 1st ed. USA: Prentice Hall Press, 2015. ISBN: 0134021363.

[20] Alan A.A. Donovan and Brian W. Kernighan. *The Go Programming Language*. 1st ed. Addison-Wesley Professional, 2015. ISBN: 0134190440.

[21] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex*. Mit Press, 2009.

[22] Matthias Felleisen and Daniel P. Friedman. "Control operators, the SECD-machine, and the λ-calculus". In: *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*. Ed. by Martin Wirsing. North-Holland, 1987, pp. 193–222.

[23] Mattias Felleisen and D. P. Friedman. "A Calculus for Assignments in Higher-Order Languages". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. Munich, West Germany: Association for Computing Machinery, 1987, p. 314. ISBN: 0897912152. DOI: 10.1145/41625.41654. URL: https://doi.org/10.1145/41625.41654.

[24] David Flanagan. *JavaScript: the definitive guide*. O'Reilly Media, Inc., 2006.

[25] J. Nathan Foster et al. "Combinators for bidirectional tree transformations". In: *ACM Transactions on Programming Languages and Systems* 29.3 (May 2007), p. 17. DOI: 10.1145/1232420.1232424. URL: https://doi.org/10.1145%5C%2F1232420.1232424.

[26] *Hack Documentation*. https://docs.hhvm.com/hack/. Accessed: 2020-08-10.

[27] Hugo Illous, Matthieu Lemerre, and Xavier Rival. "A Relational Shape Abstract Domain". In: *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*. Ed. by Clark W. Barrett, Misty Davies, and Temesghen Kahsai. Vol. 10227. Lecture Notes in Computer Science. 2017, pp. 212–229. DOI: 10.1007/978-3-319-57288-8_15. URL: https://doi.org/10.1007/978-3-319-57288-8_15.

[28] Bertrand Jeannet and Antoine Miné. "Apron: A Library of Numerical Abstract Domains for Static Analysis". In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 661–667. DOI: 10.1007/978-3-642-02658-4_52. URL: https://doi.org/10.1007/978-3-642-02658-4%5C_52.

[29] Vineeth Kashyap et al. "JSAI: a static analysis platform for JavaScript". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. Ed. by Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey. ACM, 2014, pp. 121–132. DOI: 10.1145/2635868.2635904. URL: https://doi.org/10.1145/2635868.2635904.

[30] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd ed. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[31] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88. DOI: 10.1109/CGO.2004.1281665. URL: https://doi.org/10.1109/CGO.2004.1281665.

[32] F William Lawvere. "Functorial semantics of algebraic theories". In: *Proceedings of the National Academy of Sciences of the United States of America* 50.5 (1963), p. 869.

[33] Sheng Liang, Paul Hudak, and Mark P. Jones. "Monad Transformers and Modular Interpreters". In: *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 333–343. DOI: 10.1145/199448.199528. URL: https://doi.org/10.1145/199448.199528.

[34] Tim Lindholm et al. *The Java® Virtual Machine Specification*. Java SE 14 Edition. Feb. 20, 2020. URL: https://docs.oracle.com/javase/specs/jvms/se14/html/index.html.

[35] Nicholas D Matsakis and Felix S Klock II. “The rust language”. In: *ACM SIGAda Ada Letters*. Vol. 34. 3. ACM. 2014, pp. 103–104.

[36] Matthew Might and Olin Shivers. “Improving Flow Analyses via ΓCFA: Abstract Garbage Collection and Counting”. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006)*. Portland, Oregon, Sept. 2006.

[37] Antoine Miné. “The octagon abstract domain”. In: *High. Order Symb. Comput.* 19.1 (2006), pp. 31–100. DOI: 10.1007/s10990-006-8609-1. URL: https://doi.org/10.1007/s10990-006-8609-1.

[38] E. Moggi. “Computational Lambda-Calculus and Monads”. In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. Pacific Grove, California, USA: IEEE Press, 1989, pp. 14–23. ISBN: 0818619546.

[39] Eugene Moggi. *An Abstract View of Programming Languages*. Tech. rep. University of Edinburgh, 1990.

[40] NCatLab. *Essentially Algebraic Theories*. 2019. URL: https://ncatlab.org/nlab/show/essentially+algebraic+theory (visited on 04/06/2020).

[41] *Nim Documentation*. https://nim-lang.org/documentation.html. Accessed: 2020-08-10.

[42] Benjamin C. Pierce. *A Taste of Category Theory for Computer Scientists*. Tech. rep. CMU-CS-88-203. Pittsburgh: Computer Science Dept, Carnegie Mellon University, 1988.

[43] A. Sedunov. *Kotlin In-Depth [Vol-I]: A Comprehensive Guide to Modern Multi-Paradigm Language*. BPB PUBN, 2020. ISBN: 9789389328585. URL: https://books.google.com/books?id=MVDNDwAAQBAJ.

[44] Ilya Sergey et al. “Monadic abstract interpreters”. In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 2013, pp. 399–410.

[45] Yan Shoshitaishvili et al. “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016, pp. 138–157. DOI: 10.1109/SP.2016.17. URL: https://doi.org/10.1109/SP.2016.17.

[46] Guy L. Steele Jr. “Building Interpreters by Composing Monads”. In: *Conference Record of POPL’94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*. Ed. by Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin. ACM Press, 1994, pp. 472–492. DOI: 10.1145/174675.178068. URL: https://doi.org/10.1145/174675.178068.

[47] Bjarne Stroustrup. *The C++ Programming Language, First Edition*. Addison-Wesley, 1986. ISBN: 0-201-12078-X.

[48] A. Stump et al. “A decision procedure for an extensional theory of arrays”. In: *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE Comput. Soc. DOI: 10.1109/LICS.2001.932480. URL: https://doi.org/10.1109/LICS.2001.932480.

[49] David Van Horn and Matthew Might. “Abstracting abstract machines”. In: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. 2010, pp. 51–62.

[50] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[51] David Vitek. “Auditing Code for Security Vulnerabilities with CodeSonar”. In: *IEEE Cybersecurity Development, SecDev 2016, Boston, MA, USA, November 3-4, 2016*. IEEE Computer Society, 2016, p. 154. DOI: 10.1109/SecDev.2016.042. URL: https://doi.org/10.1109/SecDev.2016.042.

[52] Philip Wadler. “The Essence of Functional Programming”. In: *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*. Ed. by Ravi Sethi. ACM Press, 1992, pp. 1–14. DOI: 10.1145/143165.143169. URL: https://doi.org/10.1145/143165.143169.

DISTRIBUTION

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop
1	D. Chavez, LDRD Office	1911	0359

Email—Internal

Name	Org.	Sandia Email Address
Technical Library	01177	libref@sandia.gov

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.