Program Analysis Without Repeating Yourself

James Koppel
Massachusetts Institute of Technology
jkoppel@mit.edu

Abstract

Traditional approaches towards implementing of dataflow analysis and other language software suffers from many fundamental limitations, typically requiring large amounts of work, with heavy duplication between conceptually-similar tools. We present a framework that allows extremely modular implementation of language software. Each target language is expressed as a collection of reusable, independent pieces. We give each piece a novel typing which allows them to be combined modularly while retaining the ability to be specialized to certain uses. Semantics are then written as interpreters which may be given multiple interpretations by supplying an evaluation context, and different evaluation contexts can be combined through use of monad transformers. The end result is that from a small handful of definitions, written as pure Java code, we can obtain a bevy of language software such as interpreters, compilers, and static analyzers, each of which can be specialized to multiple languages in multiple paradigms. Each language tool itself is composed of multiple modular pieces, which allows us to combine them into highly specialized compound tools such as superanalyzers, while retaining the power as if we had written them as a monolithic whole.

1. Introduction

It's exciting times at Automatic Code Solutions, as Richard's team is building a new product that's going to shock the marketplace: automatically making Go programs distributed. With all the infrastructure they've already built, creating it should be a piece of cake. And yet as he lists all the analysis they'd need to do, he realizes with a synching heart that every single one of them would be a new development. Who'd have thought, that, among the thousands of analyzers written by his company, no-one had bothered to write a context-sensitive escape analysis for Go which can eliminate spurious paths by string-constraint solving and understands the interface with SQL.

Today, building language software such as program analysis and transformation tools means large amounts of effort for a fairly narrow tool. The result is that most useful tools are not worth building. To understand the issues with typical approaches to designing a program to perform static analysis or some other language-processing task, consider the example of writing an analyzer to detect null-pointer exceptions in Java, and the particular task of implementing its ability to handle conditions of the form `if (x == null)` . Figure 1 gives an excerpt of the code that the SOOT framework [13] uses to do this, while Figure **??** gives the corresponding code for the ACCRUE framework [citation].

```
Value val=null;
if (left==NullConstant.v()) {
   if (right!=NullConstant.v()) {
      val = right;
   }
} else if (right==NullConstant.v()) {
   if (left!=NullConstant.v()) {
      val = left;
   }
}

if (val!=null && val instanceof Local) {
   if (eqExpr instanceof JEqExpr)
      handleEquality(val,outBranch);
   else if      ..
}
```

Figure 1. Simplified code from the NULLNESSANALYSIS of the SOOT [13] framework.

```
if ((Binary.EQ.equals(n.operator()) || Binary.NE.equals(
     n.operator()))) {
    if (n.left() instanceof NullLit || n.right()
         instanceof NullLit) {
        Expr e = (n.left() instanceof NullLit) ? n.right
             () : n.left();
         return comparisonToNull(e,
                                 Binary.EQ.equals(n.operator
                                      ()),
                                 dfIn,
                                 peer.succEdgeKeys());
    }
}
```

Figure 2. Code from the NOTNULLDATAFLOW analysis of the ACCRUE [citation needed] framework.

While there are a number of ways to restructure the code to improve these snippets, these examples highlight a fundamental facet about the way static analyzers are typically constructed which imposes significant limitations on their modularity and extensibility:

- While information is passed between nodes semantically, each transfer function itself is syntactic pattern matching. While simple, syntactic pattern matching requires manually enumerating large numbers of cases, giving rise to the next two problems. It

is also highly error-prone — especially undesirable in software which is supposed to be aiding program correctness!

- In order to make the number of cases small enough for complete syntactic pattern matching to be tractable, the code must first be preprocessed into an intermediate representation with a restricted grammar. This provides fundamental problems if we wish to use the analysis results to transform the program, such as for refactoring or program repair (e.g.: [8], [6], [7]) purposes. Indeed, existing program repair frameworks typically output their patches on preprocessed code, requiring the user to manually translate and apply them to the original program. [citation??] The ACCRUE framework does address this by using an expression-level flowgraph and building an expression evaluator atop it, similar to the approach we will take, but still resorts to pattern-matching at leaf expressions, and adds a new source of error because the framework is not flexible enough to include information about intermediate values in its type signatures , and thereby resorts to using a type-unsafe stack.
- Large numbers of transfer function must be written, potentially one per each syntactic construct. The semantics of the transfer function for if-statements for a nullness analyzer may be mechanically derived from the semantics of if-statements and the nullness-tracking abstraction, but here a large amount of code must be written manually.

This approach to static analysis contains additional problems which are less apparent in the example code, such as the following:

- Language software may only be written for one target language, with substantial modification required to work on other languages, even when they are vastly similar. The standard solution to this problem is to translate each language and version of the language into a uniform representation, an approach taken by large compiler infrastructure such as LLVM [citation], commercial multi-language tools such as Coverity [citation], and implicitly by any tool which operates on Java bytecode, such as the example Soot program. However, translating to a uniform representation intrinsically loses information, and hinders writing analyses specialized to languages that lack a construct — for instance, an analysis which is only valid in the absence of pointer arithmetic. (NOTE: Possibily include this tidbit: The company "Microprocessor Solutions" does language translation using a common universal representation. This produces translations of low quality; for their success stories, they allegedly wrote a custom direct translator which skips the IR. Semantic Designs avoids a universal representation because of this problem.)
- Language software may not take advantage of analyses written later to improve their results. For instance, consider the READWRITESETSPASS of the ACCRUE framework, which relies on a points-to analysis. If there is code that performs a write only if some condition holds, and this condition implies a variable does not point to something, the READWRITESETSPASS would need to be rewritten to take this information into account, even if there is already an analyzer that could provide this information. As another example, the Checker framework [10] contains analyzers for checking both nullness [citation] and typestate [citation], but will spuriously report errors if there is a pointer which is dereferenced, but only under a typestate in which it cannot be null. The Clang analyzer [citation] addresses this problem partially by providing a bounded path-sensitive framework, and allowing each analyzer to explicitly split states. However, analyzers still cannot share information except through what is built into the framework, and this restricts the expressivity of the framework.

In this work, we present a framework for writing language software which addresses all these problems. Our general approach is one of parametricity. In the old days, every single datatype required its own `sort` function for corresponding lists. We now have better abstractions that allow us to write a single `sort` function which can be automatically specialized to something equivalent to the hand-written ones (smart compilers can even take care of the performance difference!). Similarly, by thinking carefully about how we structure language software, we can write language software in a way that captures the essence of a semantic construct, and build components that can be automatically combined and specialized to perform innumerable tasks.

Overall, we seek to create language software which is modular in the following ways:

1. Modularity across languages: If we build tools for two languages which are 90% similar, they should be able to share 90% of the work.
2. Modularity across tools: Tools which require similar reasoning should be able to share code, even if things such as path-sensitivity are different. Tools should also be able to use facts discovered by analyzers without being explicitly built to do so.
3. Modularity across interpretation. There should be no code that could be mechanically derived from already-written code.

We present a framework which achieves a degree of modularity in all three dimensions. To achieve this, we take a number of novel departures from the typical construction of program analysis or compiler frameworks. While a few of these ideas have been previously seen in isolation in other contexts, many are new, and ours is the first to combine them into something that can achieve this level of modularity.

In total, our work makes the following contributions.

- Parametric interpretation: a way of writing interpreters abstractly which allows them to be specialized into anything from flowgraph generation to symbolic execution.
- Parametric syntax: A way of expressing languages as a combination of syntax fragments, and allowing tools to be expressed on the fragments and combined into tools specialized for a single language
- A novel typing of program terms which allows compile-time constraints on which procedures may be executed on which languages [NOTE: is this all it does?]
- A method for unifying AST nodes and flowgraph nodes of different sorts into a single representation without losing information
- Modular monadic program analysis: A way of structuring analyzers using monad transformers which allows them to share information without being engineered to do so

In order to achieve the above, we also make two additional contributions, discussed in the appendix:

- Monads without lambdas: A way of allowing framework users to write monadic code without needing to write sequential code as nested lambdas
- Proof-term embeddings: A method for embedding sophisticated type systems in languages with first-order type systems by using type-level proof terms

2. Parametric Interpretation

NOTE: This section is going to be more informal than other sections because I realized when writing it that there were decisions I made

that were critical in getting the main result of this section whose importance I hadn't noticed before.

Also, the biggest results of this section were scooped by the unpublished preprint "Abstracting Definitional Interpreters" by Darais et al. Read that paper for much of the ideas and development.

We will still present our version, which has the added benefit of type safety. The big ideas are:

- Low-level parametric abstraction lets us configure every aspect of the execution
- Upgraded type system (or, less charitably, dynamic casts) allow us to generalize the interface of abstract domains, letting us use things that aren't lattices as properties
- Using monads to parameterize control flow helps to reconcile that concrete interpreters are deterministic, while abstract interpreters can be nondeterministic

Here are the components that are framework provides to realize these ideas:

- The values used in evaluation are all "abstract values" operated on through their interface. For instance, instead of using the built-in integers, we use an `AbstractInt` type and operations defined on that type, which may be instantiated with types representing concrete integers, intervals, etc. Note that, unlike as in abstract interpretation frameworks, abstract values do not need to be lattices. In particular, they may be concrete values. Lifting this restriction allows us to unify interpreter and analyzer implementation. However, this prevents us from interchanging different possible abstract value types independently. For instance, if we have a branching construct, then, if we introduce an abstract boolean type which has "unknown" as a possible value, this creates a "long-distance" constraint that all types which could be the result type of the conditional must support the "join" operation. To handle these long-distance constraints, we perform a dynamic analysis step where each node of a language is evaluated with TODO: figure out exactly what abstract values they need to be evaluated with and what guarantees they can make. This dynamic analysis step produces a certificate $CtxCert < C, L >$ that all long-distance constraints in an (eval context, language) pairing have been satisfied, which is required to actually evaluate the node. See Appendix A for more discussion of the use of type-level certificates.
- The $EvalContext$ encapsulates all parameters used when evaluating a node.
  - First, it contains a set of abstract value types, denoted by $AbstractValue < C, S >$, where C is the eval context type, and S is the sort. So, for instance, the abstract integer type associated with eval context C is denoted $AbstractValue < C, Integer >$; these may then be converted to an unknown $AbstractInteger$ type, so that calling code may access the abstract integer operations.
  - Second, following modular monadic semantics [9], it provides a monad and a set of operations on that monad. So, for instance, any $StatefulEvalContext$ must provide put and get operations. These naturally must act on some hidden environment parameter, which is encapsulated in the associated monad. To hide the monad choice from the language implementer, we denote that monad with the $ContextVal$ type. So, for instance, a monadic value carrying an abstract integer in the monad associated with eval context C would be denoted $ContextVal < C, Integer >$; the impementation of C would use an unsafe assumption that no third-party created a class implementing $ContextVal < C, S >$, and would then cast values of that type into its internal monadic representation. The operations provided by the EvalContext will typically all return a monadic value. Computations involving monadic values must be explicitly sequenced using $bind$. This is especially important for abstracting AST nodes that implement branching control flow, as it allows us to meaningfully "take both branches."
  - Third, it provides a late-binding of recursive eval calls. All calls to eval should be threaded by invoking the eval context's $eval$ method on a node, rather than calling the node's $eval$ method directly. For example, the flowgraph-generating eval context, described in detail in Section 5, will insert an edge upon every evaluation call.
  - As an addendum, note that all recursive $eval$ calls return a $ContextVal$ monadic value, which can only be used by invoking their $bind$ method and wrapping the remainder of the computation in a lambda. For convenience, each evaluation context also provides a "Sequential Evaluator", which automatically sequences the computation by using the monads-without-lambdas technique of Appendix B

A simplified signature for the evaluation method of an AST node is as follows. We explain in Section 4 that each AST node takes as a type parameter an $EvalContext$ type representing the onle $EvalContext$ type that the node may be evaluated in. We indicate that type below with the type variable C. $CtxtCert < C, L >$ represents the type-level proof that the context and language are compatible; see Section 3 for an explanation of the L language parameter.

```
public ContextVal<C,S> eval(C c, CtxCert<C,L> cert);
```

With these evaluation contexts, we have essentially parameterized out the data that evaluation operates on, the operations they use to support their computation, and what happens in-between evaluating expressions. We can then essentially think of the implementation of an AST node as choosing how to combine togetehr these various building blocks; picking different abstractions for those building blocks results in a different language tool. In Section 5, we give a detailed example of how these elements combine to turn $eval$ functions into flow-graph generators.

2.1 Future Work

We need fully flesh out the story of what makes this different from older versions of abstract interpretation: is there already something that can turn an interpreter into a CFG builder? We also need to explore the "Monadic Abstract Interpretation" work of David Darais and understand the connections. Also, one major limitation of the current implementation is that an evaluation context is given each abstract value type parameter independently. However, they connect: to implement AbstractInt's comparison operators, you need it to be parametric over abstract booleans. And it can't be too parametric: you can't return a definite answer for $<$ if both sides are unknown, so you can't combine ConcreteBoolean with everything. Maybe I want something like `interface AbstractInt<I exteneds AbstractInt<I, C>, C extends EvalContext<C>& HasAbstractInt<I>>`; then can use `AbstractValue<C, Boolean>` to refer to said parameters.

3. Parametric Syntax

If two laguages are 70% similar, then we should be able to share 70% of the work when writing a tool for each. The 70% that is shared must therefore operate on a common datatype.

The classic approach to this problem is to translate each language to a common representation (citation? examples?), and then operate

on the shared representation. This, however, hopelessly loses information: each tool cannot assume the absence of a construct, and transformation becomes impossible. Our dilemma is that we need both the ability to write one piece of code that operates on multiple languages, and yet have that code be specialized to each.

Our solution is to decompose each language into a set of fragments, and write portions of an analyzer or transformer for each fragment. Our approach is directly based off the compositional data types of Bahr and Hvitved [1]. Our main contribution is to show how to adapt their approach to languages without higher kinds.

Compositional data types are themselves an extension of the older sum-of-products representation of abstract syntax, perhaps best known from Wouter Swierstra's seminal "Data Types à La Carte" paper [12]. We briefly explain the sum-of-products representation here.

- Each AST node type is a product type, i.e.: a record of fields. However, the AST node type constructor takes a type parameter `e` for all recursive positions in the product. So, for instance, an assignment node take the form `Assign Name e`, where `e` is a type parameter. This allows the RHS of the assignment to be any RHS from the language, where the language is later determined
- Each AST node in the language is summed together. We also have the choice of first summing different types into fragments, and then summing fragments together into a language signature. So, a fragment for arithmetic might take the form `Arith e = AddExpr e + MultExpr e`, and entire language might then be written `LangSig e = Arith e + Assign e` . Note that we keep the type parameter for recursive holes and propagate it to each summand.
- The language signature is closed using a type-level fixpoint: `Lang = Fix LangSig`, where `Fix f = f (Fix f)`.

The end result is that we have broken the syntax of a language into multiple fragments which can be combined at will. This relies in a fundamental way on the type-level fixpoint `Fix`, which has kind $(* \to *) \to *$, and thus cannot be implemented in a language without higher kinds.

As they say, every problem in computer science can be solved with an extra level of indirection, which is exactly the approach we take. There are two components to our solution. First, for every AST node type $A : * \to *$, we define an associated tag $t_A : *$, and a mechanism $+ : * \to * \to *$ for summing them. We must then define a "reifier" constructor R such that, if $ts = t_{A_1} + \cdots + t_{A_n}$, then $R\,ts \cong A_1 ts + \cdots + A_n ts$. We can then write all recursive positions in a type constructor using R, and leave open the exact set of terms which it may represent by parameterizing the tag list.

We show how a conventional type defined using explicit type-level fixpoints can be expressed using the R constructor. A recursive type representing a list of integers can be written using explicit type-level fixpoints as follows:

$$\begin{aligned} IntListF\,e &= 1 + int \times e \\ IntList &= FixIntListF \end{aligned}$$

To transform this into a version using the R constructor and no use of higher kinds, we replace the type parameter representing recursive instances, e, with a type parameter for the tag associated with recursive instances, t, and replace all uses of e with $R < t >$. We can then replace the use of a type-level fixpoint by simply picking a tag that corresponds to the unfixed constructor.

$$\begin{aligned} IntListF\,t &= 1 + int \times R\,t \\ IntList &= IntListF\,t_{IntListF} \end{aligned}$$

Constructing the R constructor is straightforward in a language with open sums, i.e.: most languages with class-based inheritance. Every constructor we may possibly want in a recursive position is defined to inherit from the R type and accept a tag list ts. However, for each such constructor A, we ensure that the type $A\,ts$ is inhabited if and only if $t_A \in ts$. We can do this by requiring that constructing an instance of $A\,ts$ requires a type-level proof that $t_A \in ts$, which we can do using the techniques of Appendix A. Associating a tag with each type constructor is trivial. In Java, this becomes particularly convenient thanks to Java's support for raw types: every type constructor of kind $* \to *$ may also be used at kind $*$.

So, a language $ArithLang$ with addition, multiplication, and assignment may be expressed as follows:

$$\begin{aligned} R\,ts &= Add\,ts + Mult\,ts + Assign\,ts + Other\,ts \\ Add\,ts &= R\,ts \times R\,ts \texttt{ if } t_{Add} \texttt{ in ts} \\ Add\,ts &= \texttt{void otherwise} \\ Mult\,ts &= R\,ts \times R\,ts \texttt{ if } t_{Mult} \texttt{ in ts} \\ Mult\,ts &= \texttt{void otherwise} \\ Assign\,ts &= Name\,ts \times R\,ts \texttt{ if } t_{Assign} \texttt{ in ts} \\ Assign\,ts &= \texttt{void otherwise} \\ ArithLangExp &= R(t_{Add} + t_{Mult} + t_{Assign}) \end{aligned}$$

In summary, this is how we represent terms in our framework:

- Each AST node implements a common $Term$ interface.
- Every AST node takes a type parameter L tagging that it is to be interpreted as an expression of a language associated with L. Each AST node's children must also be tagged with L.
- A language is expressed as a type-level list of AST node types. In order to construct an AST node tagged with a language, we must present type-level proof that the AST node is present in that language. We do this using the techniques of Appendix A.

3.1 Dealing with Multiple Sorts

Not every term can fit into every spot of a syntax tree. The syntax definition above allows us to insert assignment statements into addition nodes. We need an additional mechanism to specify the sort of every recursive occurrence.

Bahr and Hvitved offer a solution to this problem which is quite straightforward to adapt to our setting. Every type constructor is given an additional tag indicating its sort, and every occurrence is tagged with the sort of terms that may appear there. While, in the original setting, this requires an upgrade from ADTs to GADTs, here we only need the ability to inherit from a partially-applied type constructor, which can be done using ordinary Java inheritance. We simply add an extra type parameter to our $Term$ constructor indicating the sort of the desired term.

As an example, here is how we can declare an assignment node type of sort "Statement", whose right child must be of sort "IntExp".

```
public class Assign<L> extends Term<L, Statement> {
  private Name lhs;
  private Term<L, IntExp> rhs;
  ...
}
```

This setup allows us to define the syntax of a language in a manner fully isomorphic to traditional ADT-based approaches, while enjoying modularity between languages.

4. Typing Expression/Context Compatibility

In Section 2, we introduced the idea of evaluation contexts as the means by which we can derive tools from a language's semantics by parameterizing key details of processing an expression. In Section 3, we showed how we get modularity between languages by expressing each language as a collection of fragments. This poses a problem: some languages can only be run under some evaluation contexts. A language with continuations cannot be run in an interpreter that doesn't support continuations, and a language with memory allocation requires that any analysis provide some model for memory allocation. We now show the final ingredient of how we type programs. We now show how we use the type system to allow terms to impose constraints on what evaluation contexts they may be run in.

Our end goal is that we wish the type system to enforce that a node can be evaluated in a given evaluation context if and only if the evaluation context satisfies the constraints imposed by that language. We also wish to reuse each node in different languages which have different constraints.

One approach is to simply ignore the problem, and trust the programmer to only use appropriate evaluation contexts. However, this fails if this constraint is actually needed in the implementation. The $eval$ function for an $Assignment$ node must be able to call the $putVar$ operation on its evaluation context. Unless we resort to dynamic casts, this requires constraining the evaluation context type to something that supports this operation.

Our approach is to associate a constraint with each node type. These combine together to give a constraint for an entire expression. A language serves as a static guarantee of what node types may appear in an expression, and hence allows us to overapproximate the constraints that may appear in an expression. Any evaluation context which satisfies the constraints of a language may then be used to evaluate an expression from that language.

Below, we explain the components of this typing:

- Each AST node takes a type parameter C, and may only be evaluated in an evaluation context of type C. For the remainder of his section, we will pretend this is the only type parameter of a node, and write `Node<C>`, and refer to C as the associated evaluation context type. (Note that this is not actually a restriction due to our use of polymorphism.)
- Each AST node is also associated with a constraint U, and instantiating it to `NodeType<C>` requires $C \subseteq U < C >$.
- The associated evaluation context type of a node must match that of its children, meaning we can meaningfully speak of the associated evaluation context type of the entire tree. The result is that, if the nodes in an AST have constraints $U_1, \ldots, U_n$, then the associated evaluation context type C must satisfy $C \subseteq U_1 \sqcap \cdots \sqcap U_n$.
- We then give pick a type $D \subseteq U_1 \sqcap \cdots \sqcap U_n$, and give the overall expression the type $\forall C \subseteq D < C >, RootNodeType < C >$, where the notation $\forall C \subseteq D < C >$ denotes F-bounded polymorphism [2].

To gain some intuition for this scheme, simply note the information flow. We need to ensure compatibility between each node in a term and an evaluation context it will be run in. Each node has a constraint indicating that an evaluation context it's run in must support certain operations. The associated evaluation context type of the term must satisfy the constraint of every node. We generalize this with F-bounded polymorphism to any applicable evaluation context type. The constraints flow from the individual nodes, into the associated evaluation context type of the term, into the constraint placed on the outer F-bounded generalization, where a comparison is made to the final evaluation context we wish to use.

The end result is that we have a term that, after instantiating its type parameter, may be run in an evaluation context if and only if that evaluation context satisfies the collective constraints of every node in that term.

In order to implement this in Java, we again use the techniques of Appendix **??**, creating a type for the F-bounded abstraction of another type, and encoding the typing rules as proof terms.

4.1 Future Work

We still don't have a good story for how to incorporate *negative* constraints into the type system, such as for a simple dataflow-graph generator which only works for languages without mutable state. We can do various encodings, such as explicitly writing certain properties of a language as a type-level list, and requiring a certificate that an evaluation context has been checked against each member of that list in order to run it on terms of that language. However, this has the drawback that it is an externally-posed constraint: it is simple to write the analysis without requiring the lack of a property. Conversely, the constraints we speak of above are intrinsic: one cannot write an $Assignment$ AST node that does not require its evaluation contexts to support the $putVar$ operation or equivalent.

5. Example: Flow graph generation

We now present a full example of a language tool developed in our framework: a control-flow graph generator. We aim to create an evaluation context called $FlowGraphContext$ such that, when a term is evaluated in the $FlowGraphContext$ using its normal evaluation rules, it will produce a control flow graph instead of being interpreted. Our control-flow graph is based on a single rule: every expression evaluated has a control-flow edge from the last evaluated expression.

For our first attempt, we have the context maintain a graph, and track the last evaluated expression. Whenever we evaluate an expression, we create an edge from the last expression to the current one, and update the last evaluated expression. The last evaluated expression is initialized to a special $RootNode$ to indicate entry to the procedure. The $ContextVal$ for the $FlowGraphContext$ is simply the identity monad. The other operations are trivial.

```
public class FlowGraphContext extends EvalContext
 implements HasAbstractBoolean<UnknownBoolean>,
      HasAbstractInteger<TrivialInt> {
  private Graph graph = new Graph();
  private Exp<?, ?> prevExp = new RootExp();

  ...

  public <S> ContextVal<FlowGraphContext, S> eval(Exp<
       FlowGraphContext, S> exp) {
    graph.addEdge(prevExp.getLabel(), exp.getLabel());
    prevExp = exp;
    return exp.eval(this);
  }

  private static class FlowGraphContextVal<S> extends
ContextVal<FlowGraphContext, S> { ... }
}
```

This works beautifully for straight line code. However, if each node is only evaluated once, this will never create multiple outgoing edges from a single node. Clearly, something more is needed if we wish to handle conditionals.

5.1 Conditionals

When evaluating either branch of a conditional, the last evaluated expression will be the same for both: the condition. In order to capture this, we will need to move the state of the previously evaluated expression into the monad using a state monad construction. Essentially, the current state containing the last evaluated expression is then copied to both branches, which can add the appropriate edges.

What about after evaluating the branches? The statement following the if will need incoming edges from both branches. Somehow, we need a way for their to be multiple "last" evaluated expressions.

Let us observe how we implement conditionals in our framework. Conditionals are implemented using the following $eval$ method, based off the $branch$ method of the $AbstractBoolean$ interface.

```
public ContextVal<C,S> eval(C c) {
  c.eval(cond).bind((b) -> b.branch(
    () -> c.eval(thenBranch),
    () -> c.eval(elseBranch)
  ));
}
```

The $FlowGraphContext$ uses $UnknownBoolean$ for its $AbstractBoolean$ type. The $UnknownBoolean$ will execute both branches, and joins the results of both, requiring the results to implement the $Joinable$ interface.

The nondeterminism monad provides exactly what we are looking for. Running $eval$ results in a function, which, when given the last evaluated expression, results in a collection of possible resulting states, each with their own value for the new last-evaluated expression. Joining two such results simply combines the possible resulting states. Now, when running the $eval$ function of the If construct, we get edges from the condition to the first node of both branches, and a collection of states indicating that the last node of either branch could be the last-evaluated expression. When evaluating the next node, the monadic bind operation of the nondeterminism monad will execute it twice, creating edges from both possible values of the last-evaluated expression.

Most commonly, nondeterminism is implemented by accumulating results in a list. In our case, this results in path-sensitive semantics, where the state splits at every conditional, and never recombines. We want identical states to collapse back down, which we can do by using a set instead of a list.

Additionally, remember that we need to accumulate all results into a graph. Our final type of the $eval$: Exp$[S]$ $\rightarrow$ ContextVal[FlowGraphContext, S] thus expands into the following:

$$\text{Exp}[S] \rightarrow \text{Exp} \rightarrow \text{Set}[\text{Exp} \times \text{AbstractValue}[S]] \times \text{Graph}$$

Using monad transformers, this would be expressed as

$$\text{Exp}[S] \rightarrow \text{StateT Label (WriterT } Graph) \text{ AbstractValue}[S]$$

However, because updates to the graph are associative-commutative and live outside the nondeterministic component, it is actually equivalent to move the graph out of the monad and into the mutable state of the $FlowGraphContext$, meaning all graph updates are written directly into the final result, instead of into lots of mini-graphs which are combined into the final graph. This can be seen as a trivial application of the techniques of Appendix B.

5.2 Loops

It's not obvious from the above how this scheme can handle loops, which involves creating cycles in the graph. To understand how we deal with this, it's helpful to look at the actual interpreter code for a while loop. Our interpreter for while loops is implemented roughly as follows. Our actual implementation eliminates the inner bind using the techniques of Appendix B.

```
public ContextVal<C,S> eval(C c) {
  return c.eval(cond).bind((b) -> b.branch(
    () -> c.eval(body).bind((_) -> c.eval(this)),
    () -> c.makeVoid()
  ));
}
```

Similar to `if`'s, we see that the state splits in two when we evaluate the condition. In the first branch, the previous state will be `cond`, and we create an edge into the `body`, and then from the last constituent of `body` back into the `while` node when we make the recursive call, creating the desired back-edge. The problem is that we then evaluate the current `while` again, endlessly traversing and creating the same edges.

To fix this, we cache all visited nodes, and, if we visit a node again, instead of evaluating the node again, we return an empty set of states, essentially taking the interpretation that this path will loop forever. We probably could get a much better story for this, generalize it, and frame it as a fixpoint operation. This is quite close to, but probably worse than, the approach taken in the unpublished work of Darais et al that scooped this. (citation needed)

The end result is that evaluating the `while` node creates the expected edges, and results in a single state in which $cond$ is the previous expression; i.e.: while loops always evaluate their condition last before exiting. The next expression evaluated will hence get a single incoming edge from the condition of the while loop.

5.2.1 Related Work

Initially, we thought this problem was reducible to monadic fixpoints [5]. However, we soon discovered our need for a fixpoint is very different from that of [5]. In particular, their notion of fixpoint, when applied to the state monad, requires that the value be computable without referencing the state. This is in general not true in our setting: e.g.: when interpreting a variable node, the value (the result of interpretation) will depend on the state (the environment).

5.3 Invoking the Generator

To invoke the generator, we must first create a term using the typing described in Section 4, specialize it to the `FlowGraphContext` type, create a flow graph context, and then invoke it on the term.

We give our example here in a hypothetical extension to Java augmented with first-class F-bounded polymorphism. Our actual implementations embeds this into standard Java 8 using the techniques of Appendix A. We also simplify away some of the extra parameters and type parameters related to parametric syntax and language-context compatibility.

We first must create the term. The term is given a type indicating that it may be evaluated in any evaluation context that satisfies the necessary conditions. Our example term contains an $Assign$ node, which requires that its argument implement $StatefulEvalContext$.

```
(∀C ⊆ StatefulEvalContext. Exp<C>) term =
Generalize(S ⇒ new Assign<S>(...));
```

We then must specialize the term to run in the $FlowGraphContext$. Doing so requires a type-level proof that $FlowGraphContext$ extends $StatefulEvalContext$.

```
Exp<FlowGraphContext> flowGraphTerm =
  term.instantiate(Extends.<FlowGraphContext,
       StatefulEvalContext>make());
```

We are now free to evaluate the term under a $FlowGraphContext$:

```
return new FlowGraphContext().eval(flowGraphTerm).
     getGraph();
```

6. Next Steps

6.1 Having multiple abstract types together in an interpretation

In Section 2, we showed how parameterizing the abstract values of an interpretation helps to allow us to instantiate an interpreter as one of many tools. Unfortunately, there is a whole in our presentation: while we independently varied each abstract type, in reality the abstract domains are intertwined. We may be able to implement abstract booleans and abstract integers separately to implement operators such as `branch` and `add`, but, to implement `compare`, we need them tied together. Some will be incompatible, which makes the choices non-independent. For example, we can only use $ConcreteBoolean$'s if all other domains are precise enough to return a definite answer.

While Opal [4] also allows for modular mixing-and-matching of the abstract domains for different program types, each operator returns an instance of the $DomainValue$ type, a tagged sum of each abstract type. Meanwhile, for returning the result of comparisons, no customization is possible: it may only return a value in three-valued logic. This is appropriate for its use doing abstract interpretation of Java bytecode, when every result will be immediately placed onto an untyped stack, but it is insufficient for being able to reinterpret a concrete interpretation, as we do here.

6.2 Dataflow Computation Via Open Recursion

How do we implement a dataflow analysis in our framework? This requires that we evaluate nodes out of order, perhaps multiple times. It would seem at first blush that we can't do this while keeping computation with the $eval$ functions of the various nodes: the $eval$ functions invariably call $eval$ on their children, which call $eval$ on their children, etc, seemingly prohibiting any kind of external control over scheduling computations.

In fact, this is not so, because nodes do not directly $eval$ their children; rather, they request that the evaluation context do so, allowing us to supply a late-binding for the recursive $eval$ invocation. The evaluation context is free to then serve cached results and schedule and reschedule computation of a node. This effectively separates computation of each node, allowing us to run them as a dataflow analysis.

Note that not every desirable transfer function can be cleanly expressed as a specialization of a node's $eval$ function; for that, we will need to provide the ability to supply a different transfer function. However, many "congruence" rules in a dataflow analysis – ones that merely pass along or restructure information, or deal with branching – are easily expressible as such, so we foresee substantial savings in implementation cost from being able to express them so.

Note that our representation allows the children of any node to be set to anything which implements the Exp interface, is tagged with the correct sort, and belongs to the same language – and we are free to augment languages with extra nodes for analysis. We can thence achieve the same effect by setting the children of each node to an $IndirectExp$ object peered with an actual node of the program. The $eval$ function of the $IndirectExp$ object can look up the results from eval'ing the node in some data structure.

6.3 Monadic Combination of Analyses

Typical implementations of dataflow analysis are inherently unmodular. Think back to our examples of the two nullness interpreter needing to handle statements like `if (x == null)` . We can see from their need to special case these statements that they might not be able to handle more complicated conditions involving correlations of several variables that imply lack of nullness. In Section 1, we also gave an example of the inability for a third nullness checker to use information given by a typestate checker to refine its results. Ideally, these individual reasoning components – of analyzing implications between conditions, of tracking the nullness of a variable, of tracking the typestate of an object – could be designed separately, and then combined together to create hybrid analyses of arbitrary precision.

This lack of modularity is fundament to the way they are designed. A dataflow analysis can be seen as a mapping from each program point p to a transfer function f_p : State $\rightarrow$ State. This decomposition is inherently unmodular: the type State $\rightarrow$ State is invariant in the $State$, meaning that any modifications to the state will break the transfer function.

Our suggestion is to instead write transfer functions using a type Constraints m $\Rightarrow$ State $\rightarrow m$ State, where m is some monad. Essentially, this indicates that the analysis can operate in some undetermined monad as long as it provides certain operations. For instance, a nullness analyzer may demand operations for querying and setting the nullness of a variable, but leaves unspecified the precise monad, allowing a monad providing these operations to be combined using monad transformers with ones providing operations for other analyzers, similar to the technique of modular monadic semantics [9] for writing modular interpreters.

As an end result, we hope to be able to produce analyzers in a manner similar to the following:

```
Analyzer a = new Analyzer();
a.addDataflowState(new NullnessLattice());
a.overrideTransferFunction(new BooleanConstantProp());
```

The idea is that the $BooleanConstantProp$ object will be able to automatically case split on predicates such as $x == null$. For most predicates, like $x == 1$, both branches will result in the same state; however, for predicates like $x == null$, the nullness lattice will be able to track this property, and we will get behavior exactly like an implementation of a nullness analyzer that special-cases these conditions.

This is probably related to the approach taken in [3] and [11]. I still need to investigate these works to understand it.

7. Related Work

– tagless final – mms – data types a la carte, compositional data types – Polyglot has expression-level CFGs – Lerner, Grover, Chambers – calculating correct compilers – Gulwani's method – David Darais's work – One of the things ezyang sent me

A. Enhancing Java's Types with Proof Terms

The Java type system is far more powerful than most, both computer scientists and practioners, realize. Its Hindley-Milner type system with class-based subtyping allows for encoding numerous constraints, including ones which would be difficult to encode in a type system with arbitrary first-order polymorphism like System F.

Nonetheless, there are many useful things it can't express directly. Higher kinds are missing, and with it, the ability to express things like monad transformers. So is higher-rank polymorphism.

However, the presence of type variables and type constructors is a master key, for it gives us an ability that allows us to overcome these shortcomings: the ability to define arbitrary type-level terms, and their rules for composition. Using this, we can encode proof rules for arbitrarily-complicated type systems, and enforce far more interesting constraints, at the price of having to manually construct the derivations.

A.1 Simulating Higher Kinds

For our first extension, we show how to simulate higher kinds.

Java allows defining type constructors. For example, the $List$ type constructor takes a type E and returns $List < E >$, the type

of lists of E. However, we cannot have a type variable for a type constructor. We cannot express a type $F < E >$, where both F and E are variables.

However, let us consider implementing a type system with type-level applications. To do so, we would create a AST node $TApp : Type \rightarrow Type \rightarrow Type$. Node that this has the exact same signature as a type-level pair constructor, $TPair : Type \rightarrow Type \rightarrow Type$. From the meta-languange's perspective, there is no syntactic difference between an ordinary type variable and one representing a type construtor.

We can hence exploit the syntactic isomorphism between type-level application and type-level pairs to create first-class type-level application in Java.

We can create type-level pairs simply by creating any type constructor with two arguments; the pattern-matching in Java's type system allows for destructuring. To allow this to represent type-level application, we must allow the first argument to represent a constructor of a higher kind. While Java's kind system does not permit this directly, we can create a constructor of kind $Type$ which is isomorphic to a higher-kinded constructor.

More concretely, we represent an application $F < X >$ by defining a type constructor App and representing it by $App < F', X >$, where F' is some type which is in one-to-one correspondence with F. In Java, we can simply use the type constructor's corresponding raw type. So, the type application $List < String >$ could be represented as $App < List, String >$.

Now, to invoke a function defined in terms of $App < F, X >$, we must be able to convert between the isomorphic $F < X >$ and $App < F, X >$ types. We clearly cannot write a function of type $\forall F, X.F < X > \rightarrow App < F, X >$ because we cannot even write $F < X >$.

Our first attempt is to simply write one pair of functions $lift : F < X > \rightarrow App < F, X >$ and $proj : App < F, X > \rightarrow F < X >$ for each possible F. So, for instance, we would write $liftList : List < X > \rightarrow App < List, X >$ and $projList : App < List, X > \rightarrow List < X >$. Note that writing the latter requires performing a typecast, which is type-safe as long as the rest of the system was implemented correctly. Clearly, this is quite cumbersome as the number of type constructors of interest grows large. Furthermore, each cast is a "backdoor" to the system; we desire to keep the number of such backdoors small and centralized, which is difficult if each user needs to add a new one for each constructor.

While we cannot escape the need to create something specific for each type-constructor, we can improve on this somewhat by using proof terms, described below. We first define a new type constructor $TypeIso : Type \rightarrow Type \rightarrow Type$. Using the prenex polymorphism available in Java, for each F of interest, we define a term $\forall X.TypeIso < F, X >$. For example, the definition for $List$ is given as follows:

```
public static <X> TypeIso<List<X>, App<List, X>> list
    () {
  return new TypeIso<>();
}
```

We can then simply define functions

$$liftIso : \forall U, F, X.TypeIso < U, App < F, X >> \rightarrow U \rightarrow App < F, X >$$

and

$$projIso : \forall U, F, X.TypeIso < U, App < F, X >> \rightarrow App < F, X > \rightarrow U$$

Now, we need only define a single parametric $TypeIso$ term for each constructor of interest. While we still must provide a backdoor to allow the construction of a type-isomorphism proof term between arbitary types, these are much easier to check, and each module can keep all such definitions in one place. As first-class proof terms, we can also manipulate these, and transform them using symmetry and transitivity laws.

This example illustrates the general interplay between generic machinery, which can shuffle around higher-kinded types as simple variables, and its specific uses, where type-specific terms are needed. While our extended type system is generic, at the end of the day, we must use it in an ordinary Java program, and that part must be specific.

(Random theoretical sidenote: Note that we are requiring the existence of an isomorphism between a base type and functions on that base type. This is possible because we are in a constructive setting. There's probably not anything interesting to say about this.)

A.1.1 Example: Monad Transformers

Conventional wisdom states that creating type-safe monad transformers in Java is impossible because of its lack of higher-order kinds. However, we've shown above that we can simulate higher-order kinds in Java. We show now that we can also create type-safe monad transformers.

A monad is a type constructor $M : Type \rightarrow Type$ and an interface containing two methods: $return : \forall X.X \rightarrow M < X >$ and $bind : \forall X, Y.M < X > \rightarrow (X \rightarrow M < Y >) \rightarrow M < Y >$. The encoding into Java is straightforward:

```
public interface Monad<M> {
    public <E> App<M, E> ret(E e);
    public <E,F> App<M,F> bind(App<M, E> m, Function<E,
         App<M, F>> f);
}
```

This allows us to define subinterfaces for certain monadic operations. For example, here is the interface for a state monad with state S and constructor F:

```
public interface MonadState<S, F> {
    App<F,S> get();
    App<F, Void> put(S s);
}
```

A monad transformer is a type constructor $T : (Type \rightarrow Type) \rightarrow Type \rightarrow Type$ with an associated method $lift : \forall X : Type, M : Type \rightarrow Type.\text{Monad}M \Rightarrow M < X > \rightarrow T < M >< X >$. The encoding into Java becomes straightforward using our App constructor. Note that we need to explicitly pass in a $Monad$ instance so that it can invoke the relevant $return$ and $bind$ functions.

```
public interface MonadTrans<T extends MonadTrans<T, F>,
     F> {
    public <A, M extends Monad<G>, G> App<App<F, G>, A>
           lift(M monadInst, App<G, A> m);
}
```

It is now straightforward to define monad transformer instances with an appropriate $bind$ method, such as the state monad transformer. However, the main part of implementing monad transformers is not the $bind$ method, but the corresponding monad instances. For specific monad/transformer pairs, we will need to be able to be able to construct a Monad instance for the result of lifting a certain monad through a certain monad transformer. To do that, we create a new monad instance which takes a base monad as a parameter. For example, here we show that the state monad transformer applied to any monad is a monad.

```
public class StateMonadTransformerApp<S, M extends Monad
     <F>, F>
  extends Monad<App<App<StateVal, S>, F>>
```

```
  implements MonadState<S, App<App<StateVal, S>, F>> {

  public StateMonadTransformerApp(M mInst) { ... }

  ...
}
```

Using the monad transformer is now straightforward.

```
StateMonadTransformer<Integer> trans = new
     StateMonadTransformer<>();
StateMonadTransformerApp<Integer, ListMonad, List>
     mStateL =
  new StateMonadTransformerApp<>(new ListMonad());
List<Integer> posses = new ArrayList<>();
posses.add(2);
posses.add(3);

StateVal<Integer, List, Integer> sv =
  App.appToStateVal(mStateL.bind(trans.lift(new
       ListMonad(), App.listToApp(posses)), a ->
       mStateL.bind(mStateL.put(a), b ->
                    mStateL.bind(mStateL.get(), c -> {
                                 System.out.println(c);
                                 return mStateL.ret(c);
                               }
                     )
       )));

sv.getTransFn().apply(0);
// Prints
// 2
// 3
```

A.2 Proof Terms

The techniques of the previous section allow us to embed higher kinds in Java by noting that, syntactically, they are simply trees of type variables, which we can express in Java. Now we generalize this idea to embed a wide array of type systems in Java.

The idea is simply. Writing all the variables in a typing rule as XS, a typing rule takes the general form

```
F(XS)
----------- (inf-G)
G(XS)
```

where $F, G : \hat{Type} \to Type$ is some expression containing the variables in XS. We will transform this into a Java term: it is only possible to construct a term of type $G < XS >$ if we can supply a term of type $F < XS >$.

```
public class G<XS> {
  private G() {}

  public static <XS> G<XS> infG(F<XS> f) {
    return new G();
  }
}
```

Note that Java allows us to create a term of any type using $null$. To make this type system sound, we will need to use a nullness type system to forbid this. [use and cite example]

For instance, the introduction rule for existential types:

```
Gamma, x |- F<x>
--------
Gamma |- exists x. F<X>
```

can be encoded as such:

```
public class Exists<F> {
  private Exists() {}
  public static <X, F> make(Function<X, App<F,X>> f) {
    return new Exists();
  }
}
```

Here, we have only encoded the type system; it is a straighforward extension to also encode the computational interpretation of existentials, existential packages. Note also that here we have given the presentation of existentials in a point-free style, purely in terms of type-level functions; fully using this system requires also encoding the rules of substitution as proof terms. Yes, that means that using it will require you to manually construct a tree with a node for every step of the substitution. This technique makes these embeddings possible, not easy.

Note also above that we have removed the need to model the environment by using Java's environment (NOTE: say something about higher-order abstract syntax here).

Some rules may be defined in terms of a fresh variable. We can hijack Java's existing universal parametricity to mandate this. For example, consider the universal introduction rule:

```
Gamma, x |- F<x>   x not free in Gamma
-----------
Gamma |- forall x. F<x>
```

We translate it as follows:

```
public interface ForallIntro<F> {
  public <X> F<X> create();
}

public class Forall<F> {
  private Forall() {}

  public static Forall<F> make(ForallIntro<F> intro) {
    return new Forall();
  }
}
```

A.3 Translating Between Intrinsic and Reified Constraints

To fully interoperate with Java, it is not enough to simply have unqualified type variables. Java allows for subclass constraints to be placed on type variables

First, it is simple to create a piece of type-level evidence that one type variable extends another.

```
public class Extends<U,V> {

  private Extends() {}

  public static <U extends V, V> make() {
    return new Extends<U,V>();
  }

  public V upcast(U u) {
    return (V)u;
  }
}
```

This first-class evidence has many uses. For starters, we can overcome the limitation in the Java type system that, while we may require that a type variable extend multiple concrete classes and interfaces, e.g.: by writing `X extends Collection&Comparable`, we may only impose a single variable constraint – `X extends U` is legal, but `X extends U&Comparable` or `X extends U&V` are not. See Section 4 for a good example of using these proof terms of subclass constraints.

What if we have a method foo that has a `X extends V` constraint, and we have a term of type $Extends < U, V >$? How can we call foo at type U?

Hastily written explanation of how to do this: The user creates a method whose signature is parameterized on $< WextendsV >$. We can invoke this generically; Java will simply synthesize a new variable of that type at the callsite because it is otherwise unconstrained. If the result is in terms of this new variable W, we can convert it back to being in terms of U as follows: convert it to an App, invoke a special method which infers necessary type equalities from a combination of $Extends$ proof terms and built-in Java $extends$ constraints to perform a cast, and then convert it away from an app.

A.4 First-class F-Bounded Polymorphism

There is one final step in creating first-class terms for Java's type system. Java does not merely allow type variables to have constraints of the form `U extends V`; it allows F-bounds like `U extends F<U>`.

Unfortunately, unlike for normal subclass constraints, it is impossible to synthesize a variable with constraint `X extends F<X>` in a manner which is generic in F. Here, we cannot escape Java's lack of higher kinds: Any such mechanism must have a fully-applied type constructor passed in, which leaves no way to extract the variable. One attempt of ours was to parameterize a constructor onX, G, F and require $Extends < X, G >$ and $TypeIso < G, App < F, X >>$; this fails because there is no way to then force X to be a free parameter.

Ideally, we would like to be able to create an interface like the following:

```
public interface FBoundCreator<G,F> {
  public <X extends F<X>> App<G,X> create();
}
```

While we cannot create this interface and have it be enforced by the Java type system because we cannot write down $F < X >$, we can nonetheless create and enforce this interface by turning to reflection. We can simply use reflection to look up the F and G parameters on the instance of $FBoundCreator$, and then check for the existence of a $create$ method with the desired signature. Note that, while Java execution does have type erasure, by default type parameters of method signatures and classes are still accessible at runtime by reflection.

The rules for F-bounded generalization and instantiation are then simple. First, the $FBound < X, F >$ class is type-level proof that `X extends F<X>`.

```
public class FBound<X,F> {
  private FBound() {}

  public static <X,F,A> FBound<X,F> make(Extends<X, A> c,
       TypeIso<A,
App<F,X>> iso) {
    return new FBound();
  }
}
```

We now define the generalization and instantiation rules:

```
public class FBoundedUniversal<F,T> {
  private App<T,?> value;
  private FBoundedUniversal(App<T,?> value) {
    this.value = value;
  }

  public static <F,T> FBoundedUniversal<F,T>
generalize(FBoundCreator<F,T> creator) {
    return new FBoundedUniversal(creator.make());
  }

  public <X> App<T,X> instantiate(FBound<X,F> bound) {
    return (App<T,X>)value;
  }
}
```

A.5 Further Work

We need to craft a more general story about the exact situations when this is useful beyond the present context. I'd like to have a good story about embedding different kinds of DSLs into Java with these techniques. One example that comes to mind is creating typestate by embedding it into first-order logic using a resource semantics, and then embedding the first-order logic into Java using proof terms.

We also still need to develop its use in our current context (e.g.: actually see through the thing about proofs of language membership) and learn how cumbersome it is to program with these.

I also think we can develop a metaprogram which translates nearly-arbitrary type systems into a Java encoding.

A.6 Related Work

Someone previously used the same App trick for simulating higher-order kinds, and implemented much of the Haskell standard library in Java: https://github.com/DanielGronau/highj

B. Monads without Lambdas

As explained previously, a critical ingredient in our ability to modularly combine semantics, whether that's getting analyzers to automatically use each other's results, or writing an `eval` function in a way that's agnostic to what kinds of effects are used in the rest of the program, is to rely on monads to sequence operations, and parameterize the computation by some unknown monad. Here's what a monadic implementation of the `eval` function for the SEQ AST node, which simply runs one statement after another, might look like in Java 8.

```
public <C extends EvalContext<C>> AbstractValue<C, Void>
      eval(C c) {
  return ctx.eval(stmt1).bind((_) ->
      ctx.eval(stmt2).bind((_) ->
        ctx.makeVoid()));
}
```

`stmt1` may be a statement containing nondeterminism, continuations, or indirect control flow. We may need to execute `stmt2` multiple times when the rest of the program has a different state, or we may not want to execute it at all. Or when constructing a control-flow graph, and we need to know what the last executed statement is when examining `stmt2` so we can create a proper control-flow edge, `stmt1` may contain branches and therefore have multiple possible "last" statements. It would seem that writing sequential statements as nested lambdas, where each successive computation is suspended so that the `bind` method is free to run it multiple times or not at all, is necessary to achieve this modularity. In fact, this is not the case, and we can instead write the code in the manner below so that it is equivalent to the code above for arbitrary monads.

```
public <C extends EvalContext<C>> AbstractValue<C, Void>
evalSequentially(SequentialEvaluator<C> seq) {
  seq.eval(stmt1);
  seq.eval(stmt2);
  return seq.getCtx().makeVoid();
}
```

At first, this might seem impossible: it appears that `stmt2` is always executed exactly once. However, note the following two facts:

- The `EvalContext` invokes the `eval` and `evalSequentially` methods of the AST node, and can choose to evaluate it 0 or more times, and store state associated with its execution

- The `SequentialEvaluator` may share state with the `EvalContext` and may update the state with each call to `seq.eval()`. Each call to `seq.eval()` may give an answer which depends on said state, and may throw an exception to directly return control to the `EvalContext`, which invoked `evalSequentially`.

These two facts allow the `EvalContext` to "observe" the behavior of the `eval` / `evalSequentially` method, and to control its execution. Over the next two sections, we will present two constructions for `seq.eval` such that `seq.eval(s); <rest of computation>` is equivalent to `ctx.eval(s).bind((_) -> { <rest of computation> });` for both the nondeterminsm and continuation monads. These constructions both rely on having mutable state and exceptions in the language. Note that we could remove the dependence on exceptions by setting a flag to ignore the results of the rest of the computation, and returning a dummy value to the computation.

B.1 Lambda-free nondeterminism

The main idea of the lambda-free nondeterminism construction is that every sequence of nondeterministic choices in the `evalSequentially` method determines a path through its execution. The traditional implementation of nondeterminism accumulates a list of all choices at every branch point and then executes the remainder of the execution with each possible choice. Ours instead repeatedly executes the entire `evalSequentially` method, picking a different sequence of nondeterminstic choices each time. To enable this, it maintains a cache of the result of all subcomputations, and uses exceptions to skip the remainder of the computation in case it encounters a nullary choice. Note that this construction assumes that the implementation of `evalSequentially` contains no side effects.

Code for the construction is given in Figure B.1. NOTE: There is an error in the code, in that it doesn't depict that recursive EVALOUTER calls occur in a new sequential evaluator with new $stack$ and $evalledStmts$ state. I don't really know how to properly typeset this, given the impedance mismatch between the procedural algorithmicx package and the actual object-oriented algorithm. Similarly, it does not contain a catch block for the thrown exception.

Evaluation of a node begins in $evalOuter$. $evalOuter$ initializes a new intermediate-result cache for evaluating this node. It then runs the node's $evalSequentially$ method in a loop. Each evaluation explores one path through the tree of nondeterminstic choices. It combines the results from each branch, and then returns the collected results once the branches have been exhausted. Note that, while our presentation below shows results being collected in a list, we can also collect results in a different data structure such as a set, which can collapse occasions where we received the same result from different branches.

Within an invocation to a node's $evalSequentially$ method, it may invoke the $SequentialEvaluator.eval$ method on many subnodes. The $SequentialEvaluator.eval$ method first checks to see if it has cached results for evaluating this subnode. If not, it evaluates the subnode, caches the result, and pushes a label onto the stack to track the order in which nodes were evaluated. It then returns the first of the cached results to the caller, throwing an exception if there is none.

Note: Should probably say something about how $evalOuter$ pops things off the stack to iterate through all branches.

B.2 Lambda-free continuations

The main idea of lambda-free continuations is, by observing the recursive calls to $SequentialEvaluator.eval$, we can explicitly maintain a stack tracing the previous returns and the position in the code. To invoke a continuation, we simply throw an exception to rewind to the beginning of the computation, replay it up to the relevant $callCC$, and return the value passed to the continuation. Figure B.2 gives the code for the lambda-free continuation instruction.

```
1: stack ← Stack()
2: evalledNodes ← Map()
3: function SEQUENTIALEVALUATOR.EVAL(node)
4:     if not (evalledNodes contains (numBinds, node)) then
5:         evalledNodes[(numBinds, node)] ← EVALOUTER(node)
6:         stack.push((numBinds, node))
7:     end if
8:     choices ← evalledNodes[(numBinds, node)]
9:     numBinds ← numBinds + 1
10:    if choices is empty then
11:        throw NoChoicesException
12:    else
13:        return choices.first()
14:    end if
15: end function
16:
17: function EVALOUTER(node)
18:    result ← []
19:    repeat
20:        for all 1 do r in NODE.EVALSEQUENTIALLY(this)
21:            APPEND(result, r)
22:        end for
23:        last ← stack.top()
24:        evalledNodes[last].removeFirst()
25:        if evalledNodes[last] is empty then
26:            evalledNodes.remove(last)
27:            stack.pop()
28:        end if
29:    until stack is empty
30: end function
```

Figure 3. Code for the lambda-free nondeterminism construction

The construction of lambda-free continuations is similar to that for lambda-free nondeterminism, in that it caches the result of evaluating subnodes, and relies on re-executing the computation for its effects. Unlike the nondeterminism construction, it does not re-execute a computation unless a continuation is invoked, and it re-executes the computation from the beginning, rather than from the current node.

We begin by invoking $evalTop$ on the root node, which immediately invoke's the node's $evalSequentially$ operator, which may in turn call $SequentialEvaluator.eval$ on its subnodes. Each such call places an $EnterFrame$ on the stack to record that it has entered that node, and then calls that node's $evalSequentially$ method. When the method returns, it rewinds the stack up to the point of invocation, and places a $ResultFrame$ on the stack to record the result of evaluating that node.

To create a continuation, we simply copy the current state of the stack and store it in a $Continuation$ object. To perform a $callCC$, we simply call the supplied method with the current continuation, and return the result.

The interesting part is when we invoke a continuation. To invoke a continuation, we throw an exception to pass control back to $evalTop$ along with the value passed to the computation. We then begin replaying the computation. To do this, we copy the continuation's stored stack into a queue, and re-evaluate the root node. Now, whenever we encounter a recursive call to $SequentialEvaluator.eval$, we take the first frame from the queue: if it's a $ResultFrame$, we return the stored result; else,

we invoke the node's $evalSequentially$ method. Each time, we move a frame from the queue to the stack, reconstructing our steps. Finally, when we exhaust the queue and reach the relevant $callCC$ invocation, we return the value passed to the continuation.

This algorithm could be optimized by checkpointing each recursive call, so that, when we invoke a continuation, we resume computation at the parent node, rather than at the root node.

```
 1: stack ← Stack()
 2: queue ← Queue()
 3: contVal ← nil
 4:
 5: function EVALTOP(node)
 6:     node.evalSequentially(this)
 7:     catch ContinuationException(contStack, val)
 8:     stack ← Stack()
 9:     queue ← STACKTOQUEUE(contStack)
10:     contVal ← val
11:     return EVALTOP(node)
12: end function
13:
14: function SEQUENTIALEVALUATOR.EVAL(node)
15:     if queue is empty then
16:         PUSH(stack, EnterFrame)
17:         result ← node.evalSequentially(this)
18:         repeat
19:             frame ← POP(stack)
20:         until frame == EnterFrame
21:         PUSH(stack, ResultFrame(result))
22:         return result
23:     else
24:         nextFrame ← DEQUEUE(queue)
25:         PUSH(stack, nextFrame)
26:         if nextFrame == EnterFrame then
27:             return node.evalSequentially(this)
28:         else
29:             (ResultFrame val) ← nextFrame
30:             return val
31:         end if
32:     end if
33: end function
34:
35: function CALLCC(f)
36:     if contVal ≠ nil ∧ queue is empty then
37:         result ← contVal
38:         contVal ← nil
39:         queue ← Queue()
40:         return result
41:     else
42:         return f(Continuation(COPY(stack)))
43:     end if
44: end function
45:
46: function CALLCONTINUATION((Continuation stackCopy), val)
47:     throw ContinuationException(stackCopy, val)
48: end function
```

Figure 4. Code for lambda-free continuation construction

B.3 Future Work

We have above given constructions for the nondeterminism and continuation monads, and it is simple to give constructions for the reader, writer, and state monads. It is known that arbitrary monads can be simulated using continuations [citation]. There are a few big threads of work remaining. First, we need to optimize the continuation construction to resume computation from the parent node instead of the root.Second, we need to show how to mechanically derive the monad-without-lambda constructions from the monad definitions and related them to the axioms of the relevant monads.Third, we need to develop a mechanism to extend these constructions to monad transformers. Fourth, we would like to find exactly what conditions are needed to use these constructions and find other applications of this technique beyond the setting of building frameworks for language software.

Acknowledgments

I would like to thank my advisor Armando Solar-Lezama for his constant feedback and guidance and Jean Yang for encouraging to pursue this project when I was feeling pressure to do something with more immediate benefits. I would also like to thank Stephen Chong, Ziv Scully, Oleg Kiselyov, Edward Z. Yang, and my labmates in the Computer-Aided Programming group for discussions about various ideas in this paper.

References

[1] Patrick Bahr and Tom Hvitved. Compositional Data Types. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*, pages 83–94. ACM, 2011.

[2] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C Mitchell. F-Bounded Polymorphism for Object-Oriented Programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280. ACM, 1989.

[3] David Darais, Matthew Might, and David Van Horn. Galois Transformers and Modular Abstract Interpreters: Reusable Metatheory for Program Analysis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 552–571. ACM, 2015.

[4] Michael Eichberg and Ben Hermann. A Software Product Line for Static Analyses: The OPAL Framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6. ACM, 2014.

[5] Levent Erkök. *Value Recursion in Monadic Computations*. PhD thesis, Oregon Health and Science University, 2002.

[6] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. Repairing Programs with Semantic Code Search (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 295–306. IEEE, 2015.

[7] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic Patch Generation Learned from Human-Written Patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[8] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A Systematic Study of Automated Program Repair: Fixing 55 Out of 105 Bugs for $8 Each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[9] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM, 1995.

[10] Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212. ACM, 2008.

[11] Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. Monadic Abstract Interpreters. In *ACM SIGPLAN Notices*, volume 48, pages 399–410. ACM, 2013.

[12] Wouter Swierstra. Data Types à La Carte. *Journal of Functional Programming*, 18(04):423–436, 2008.

[13] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a Java Bytecode Optimization Framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.