One CFG-Generator to Rule Them All

James Koppel
MIT
Cambridge, MA, USA
jkoppel@mit.edu

Sreenidhi Nair
ByteAlly
Chennai, India
sreenidhi@byteally.com

Armando Solar-Lezama
MIT
Cambridge, MA, USA
asolar@csail.mit.edu

Abstract

This paper introduces two independent yet synergistic contributions related to control-flow graphs in programming tools. Our first contribution is a language-modular design for the construction of CFG generators. In this paper, we use this technique to build a complete CFG-generator for a small language with 17 distinct node types in only 19 lines of language-specific code. Our full implementation creates complete control-flow graphs for 5 industrial languages in only 1100 SLOC (averaging 122 language-specific SLOC), compared to 2400 for combined competitors. Our second contribution is a new program transformation primitive, CFG-based insertion, which provides the operation "insert statement S such that it always executes before expression E." This operation makes it possible to build 1-line program transformations that handle special cases across multiple languages. Both contributions have been implemented in Haskell in the Cubix framework, with support for C, Java, JavaScript, Lua, and Python. Together, the two contributions of this paper make it substantially easier to build high-fidelity CFG generators for many languages and use them in language-parametric program transformations.

ACM Reference Format:
James Koppel, Sreenidhi Nair, and Armando Solar-Lezama. 2018. One CFG-Generator to Rule Them All. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, Article 122, 13 pages. https://doi.org/10.1145/3276492

1 Introduction

New software becomes feasible when what once required implementing a complex procedure is now done by assembling building blocks. This is a paper about easing the implementations of tools that involve control-flow graphs, and we begin by posing a simple source-to-source transformation as a challenge: Some analysis has identified that a string variable `s` is unsanitized at a certain use-site. How would you build a tool that inserts the line `s = sanitize(s);` before that use-site, as near as possible?

This operation seems trivial when one imagines the common case where `s` sits within a one-line statement, and the transformation must merely insert the sanitization on the previous line. Yet complexity appears when one considers that the use of `s` may be awkwardly situated inside some construct wiith interesting control-flow, and may have multiple predecessors, separated by some distance. A common case is when `s` lies in the condition of a for-loop; then the sanitization must be inserted before the loop, at the end of the loop, and before every `continue` statement. Fig. 1 gives example input and output for this case in C. Similar problems arise with other control-flow constructs. When use is in the condition of a Python `elif`, for instance, as in Fig. 2, adding the sanitization will require splitting the `elif` into a nested conditional (Fig. 2b).

There are many existing techniques for program transformation, such as those surveyed in [22]. To a first approximation, all of them require the programmer to eventually specify tree rewrites, and would hence require giving many cases to work on these examples. But, using the techniques of this paper, as implemented in the Cubix framework, the following function performs this transformation on C, Python, and three other languages, correctly handling the special cases above.

```
insertSanitization targetNode var =
  dominatingPrepend targetNode
                    (assign (ident var)
                            (functionCall "sanitize"
                                          [ident var]))
```

Half of the magic of this snippet comes from the *incremental parametric syntax* introduced in a previous publication on the Cubix framework [13], which makes it possible for that `(assign ...)` expression[1] to actually expand into one of several language-specific variants, depending on the inferred type of each call-site. In this paper, we introduce the other half of the magic: the new source-to-source transformation paradigm of *control-flow based insertion*, invoked here through the `dominatingPrepend` function. The idea of CFG-based insertion is to provide a new primitive operation "Insert code A at points ensuring that it always runs before/after code B." Behind this lies a graph search and some language-specific operations for checking where the insertion is possible; these

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
Conference'17, July 2017, Washington, DC, USA
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/3276492

[1]The `(assign ...)` expression requires a few helper functions to compile verbatim, filling in default parameters. We leave all discussion of Cubix's parametric syntax to the original Cubix paper. [13]

```
for (int i = 0; i++ && !isStopCode(s); i++) {
  if (!containsCommand(s))) {
    s = nextInput();
    continue;
  }

  process(s);
  s = nextInput(s);
}
```

(a)

```
s = sanitize(s)
for (int i = 0; i++ && !isStopCode(s); i++) {
  if (!containsCommand(s))) {
    s = nextInput();
    s = sanitize(s)
    continue;
  }

  process(s);
  s = nextInput(s);
  s = sanitize(s)
}
```

(b)

Figure 1. Sample C input (a) and output (b) for sanitization-transformation, targeting isStopCode(s).

```
if notificationReceived():
  handleNotification()
elif isCommand(s):
  process(s)
else:
  log("Skipped")
```

(a)

```
if notificationReceived():
  handleNotification()
else:
  s = sanitize(s)
  if isCommand(s):
    process(s)
  else:
    log("Skipped")
```

(b)

Figure 2. Sample Python input (a) and output (b) for sanitization-transformation, targeting isCommand(s).

together turn what would be a menagerie of casework into one declarative statement of intention.

While CFG-based insertion is a useful item in the transformation toolbox for any (imperative) language, its greater promise comes from its declarative ability to abstract over language-specific details, as illustrated here. As software engineering becomes increasingly polyglot, with engineers and systems fragmented across an ever-increasing set of languages, advanced tools will increasingly require multi-language support to be economically justifiable. The example above shows we are able to use CFGs to do transformation in a multi-language fashion, yet, to truly amortize the cost of tool-development across languages, one must also amortize the infrastructure. Can one also build the CFG-generators themselves in a multi-language fashion?

In the other half of this paper, we answer in the affirmative, presenting a new monadic decomposition of CFG-generators which allows them to be written in a language-modular fashion. The upshot of this is that we were able to construct complete CFG generators for 5 languages, passing an extensive test suite which includes CFG "challenge problems" such as Duff's device, in only 1100 total lines of code — and averaging 122 lines of language-specific code — compared to a combined 2400 lines of code in the best pre-existing comparison CFG-generators we found. We demonstrate this in miniature with a fully-worked tutorial: 19 lines of code for a full CFG-generator on a language with 17 distinct node types.

At first glance, sharing work between CFG-generators for many languages again sounds easy, as languages are often very similar. As many languages have while-loops, can we just write a generic "make CFG for while" function and invoke it from all the language-specific generators? But the problem is that similar languages can be subtly different. In this case, a naive "make CFG for while" function does not work due to the presence of *control effects*: loops in different languages may interact differently with break, continue, and goto statements within their bodies. Similar factors prevent code reuse between while- and foreach-loops, even though the control-flow structure is the same. And while many researchers have designed datatypes for representing ASTs in a language-modular fashion — most famously data types à la carte [20], taken to its extreme in the Cubix framework — we shall find that control-flow graphs, under their most common designs, require non-local informatiion to construct, demanding recursion patterns which break the separation between language fragments.

We answer with a fully-compositional design for CFG-generators. Our design uses monads to separate handling of control effects, and removes the need for non-local information. Under this design, it becomes quite straightforward to create a single, reusable "make CFG for while" function. Our design further allows several standard cases to be handled declaratively, including nodes with a default left-to-right evaluation order, nodes which do not take part in computation (e.g.: type declarations), and nodes which introduce a new control-flow contour (e.g.: lambdas). We explain these techniques with a tutorial in which we construct full CFG-generator for a small IMP language with 17 distinct node types; including loops, lambdas, and goto; in only 19 lnes of code.

We have implemented both the design for language-modular CFG-generators and CFG-based insertion in the CUBIX framework, where we respectively used them to implement the CFG-generators for all 5 of CUBIX's currently supported languages (C, Java, JavaScript, Lua, and Python), and to build two source-to-source transformations which successfully handled language corner-cases, as verified on compiler test suites. In summary, our contributions are:

1. A new monadic decomposition of CFG-generation, which permits CFG-generators to be built in a language-modular fashion, and hence for much less total labor than independent generators per language.
2. The new program-transformation operation of CFG-based insertion, a declarative mechanism for a common subtask of program transformations.

The remainder of the paper is organized as follows: §2 and §3 present our first contribution of building language-modular CFG-generators. §4 presents our second contribution, CFG-based insertion. The rest of the paper presents our implementation of both of these within CUBIX, experiments, related work, and conclusion.

Prerequisites The code in this paper is given in Haskell. Our exposition expects the reader to be familiar with Haskell notation, typeclasses, and the standard monads. We also make some use of type-level programming (type-level lists and type families), GADTs, rank-2 types, and explicit type parameters.

2 CFG Generation: Not So Easy

Though our ultimate goal is to build language-modular CFG-generators, modularity problems appear in CFG generators even for one language. In this, we illustrate the problems of naive definitions CFG generators. In a tutorial reconstruction, we proceed to refine such a naive generator into a new, compositional design, in preparation for generalizing this design in §3.

2.1 A Naive CFG-Generator

Consider this datatype for a simple imperative language:

```
type Var = String
type Label = Int

data Exp = Add Exp Exp | Lt Exp Exp | VarExp Var

data Stmt = Assign Var Exp
          | If Exp LabStmt LabStmt
          | While Exp Stmt
          | Block [LabStmt]
```

```
type LabStmt = (Label, Stmt)

type Graph -- definition not shown
addEdge :: Graph -> Label -> Label -> Graph
connect :: Label -> Label -> State Graph ()

optConnect :: Maybe Label -> Label -> State Graph ()
optConnect l1 (Just l2) = connect l1 l
optConnect l1 Nothing   = return ()
```

Disregarding the extra complexity to construct basic-blocks, a traditional statement-level CFG-generator would create one node per statement, as a recursive traversal. We construct an implementation below, and then discuss why its design does not readily extend to a language-generic implementation.

First we design the recursion. Let us think about what information must be passed down and up the stack in such a traversal. There will be an edge between the last statement in each branch of an if-statement, and the first statement after the if. Some part of the code must have access to the nodes of both. This can be accomplished either by passing down the node after the `if`, or by passing up the set of nodes which may be the last thing executed. In our implementation, we choose the former: the recursion passes down the the next node that runs after the current term.

Below is code for the CFG generator. As the language is small, the code is short, yet contains a high density of special cases.

```
genCfg :: Maybe Label -> LabStmt -> State Graph Label
genCfg next (l, Assign _ _)  = do optConnect l next
                                  return next
genCfg next (l, If _ s1 s2)  = do l1 <- genCfg next s1
                                  l2 <- genCfg next s2
                                  connect l l1
                                  connect l l2
                                  return l
genCfg next (l, While _ s)   = do l' <- genCfg (Just l) s
                                  connect l l'
                                  optConnect l next
                                  return l
genCfg next (l, Block ss)    = do ml' <- genCfgBlock next ss
                                  case ml' of
                                    Just l' -> connect l l'
                                    Nothing -> optConnect l next

                                  return l

genCfgBlock :: Maybe Label -> [LabStmt]
                           -> State Graph (Maybe Label)
genCfgBlock next []       = return Nothing
genCfgBlock next (s:ss)   = do l  <- genCfgBlock next ss
                               l' <- genCfg (Just l) s
                               return l'
```

For a language so small, any problems that could exist in this code would be trivial inconveniences. Yet, viewed through a magnifying glass, it is already possible to spot two issues that may blossom into barriers as the language grows, or when trying to make this code language-modular.

The first is that this code is not *compositional.* This means that the CFG for one node is not a function of the CFGs for its child nodes. Indeed, some lack of modularity is already present: thinking carefully about the reason for each line, the handling of next nodes is derived from the design of Block nodes (e.g.: it would be different if the cons-lists of Block were replaced with snoc-lists or a binary Seq node).

The second is that the implementation of each case assumess this language has no indirect control-flow. Upon the introduction of new nodes with indirect control-flow, every case of the CFG-generator would need to be rewritten to thread extra state throughout the generator.

In the following sections, we construct a new architecture that addresses all these problems.

2.2 The Case for Enter/Exit

In the previous section, we determined that the genCfg function needed to pass around non-local information because it may sometimes need to draw an edge connecting an AST node and its (great-)grandparent. This is also the reason genCfg is not compositional. To make it compositional, we must change the CFG design. Our proposal: create two CFG nodes per AST node, representing its entry and exit.

With this change, each AST node's CFG fragment depends only on its children. No extra information is passed down in recursion. And, as an added bonus, the special casing has also been eliminated, for it stemmed from decisions over where to handle the non-local information. Here are the first three cases:

```
makeEnterExit :: State Graph (Label, Label)
-- Implementation not shown

genCfg :: LabStmt -> State Graph (Label, Label)
genCfg (_, Assign _ _) = do (enter, exit) <- makeEnterExit
                            connect enter exit
                            return (enter, exit)
genCfg (_, If _ s1 s2) = do (enter, exit) <- makeEnterExit
                            (enter1, exit1) <- genCfg s1
                            (enter2, exit2) <- genCfg s2
                            connect enter enter1
                            connect enter enter2
                            connect exit1 exit
                            connect exit2 exit
                            return (enter, exit)
genCfg (_, While _ s)  = do (enter, exit) <- makeEnterExit
                            (enterBody, exitBody) <- genCfg s
                            connect enter enterBody
                            connect exitBody enter
                            connect enter exit
                            return (enter, exit)
```

While the increase in the number of CFG edges has correspondingly increased the size of the code, the amount of information has decreased thanks to the higher symmetry. Gone are the branches; instead, each case reads as a list of the edges corresponding to the current node.

However, the casework needed to deal with empty blocks has not gone away — it's gotten worse! Continuing, the case for Block's looks like this:

```
genCfg (_, Block ss) = do
  (enter, exit) <- makeEnterExit
  mEnterExitSS <- genCfgBlock ss
  case mEnterExitSS of
    Nothing                -> return ()
    Just (enterSS, exitSS) -> do connect enter enterSS
                                 connect exitSS exit

  return (enter, exit)

genCfgBlock :: [LabStmt] -> State Graph (Maybe (Label, Label))
-- Implementation not shown
```

We shall find that dealing with possibly-empty returned nodes is such a common situation that it merits its own primitive, and all the casework can be encapsulated in a new operation for combining a possibly-empty pair of nodes.

```
type EnterExitPair = Maybe (Label, Label)

combineEnterExit :: EnterExitPair -> EnterExitPair
                                  -> State Graph EnterExitPair
combineEnterExit Nothing          p2             = return p2
combineEnterExit p1               Nothing        = return p1
combineEnterExit (Just (l1, l2)) (Just (l3, l4)) = do
    connect l2 l3
    return (Just (l1, l4))
```

With this new primitive, branching is eliminated. Here is the new code for Block:

```
genCfg (_, Block ss) = do
  (enter, exit) <- makeEnterExit
  eepSS <- genCfgBlock
  x <- combineEnterExit (Just (enter, enter)) eepSS
  combineEnterExitPair x (Just (exit, exit))

genCfgBlock :: [LabStmt] -> State Graph EnterExitPair
genCfgBlock ss = fold (\s mEepRest -> do
                           eepS <- genCfg s
                           eepRest <- mEepRest
                           combineEnterExit eepS eepRest)
                      Nothing
                      ss
```

2.3 Monadic Deferral

Having made CFG-generation compositional, we chase the next milestone of modularity: how to make each case of genCfg into an independent function? Doing so would make it possible, for instance, to share CFG-generation code across all languages with while-loops.

```
genCfgWhile :: (Label, Label) −> State Graph EnterExitPair

genCfgPython :: PyTerm −> State Graph EnterExitPair
genCfgPython (PyWhile _ s) = do sEnterExit <− genCfgPython s
                                genCfgWhile sEnterExit
−− ... other cases

genCfgC :: CTerm −> State Graph (Label, Label)
genCfgC (CWhile _ s) = do sEnterExit <− genCfgC s
                          genCfgWhile sEnterExit
−− ... other cases
```

This refactoring involves hoisting the recursive genCfg call out of the case for While. This works for the simple language we have used thus far. It does not work when genCfg may have non-commutative effects. Such effects in the CFG-generator occur when there are control-effects in the code. Let us add break and continue to our language:

```
data Stmt = ...
          | Break
          | Continue
```

Generating a CFG for these new nodes requires tracking the break and continue targets. This requires adding a new parameter to CFG generation, either explicitly, or by rolling it into the state. The latter is clearly the more modular option. We thus extend the state of the CFG-generator to also include a stack of break/continue targets, and update the signatures of other functions accordingly.

```
data CfgGenState = CfgGenState { graph :: Graph
                               , loopStack :: [(Label, Label)] }

type CfgGen a = State CfgGgenState a

combineEnterExit :: EnterExitPair −> EnterExitPair
                                  −> CfgGen EnterExitPair
connect :: Label −> Label −> CfgGen ()
pushLoop :: Label −> Label −> CfgGen ()
popLoop :: CfgGen ()
```

With these additions, it is no longer possible to write genCfgWhile with the given signature. genCfgWhile must use pushLoop and popLoop to communicate the break/continue targets to invocations of genCfgC/genCfgPython on nodes in the loop body. However, the CFGs for these nodes are generated before genCfgWhile is even called. In this new language, the CFG fragment for a node is no longer a function of the CFG fragments for its subnodes. genCfg is no longer compositional. Instead, the CFG fragment for a node also depends on the control effects of its children. And so, by accepting a monadic value for the CFGs of its children, genCfgWhile can control the state passed to the recursive calls to genCfg, and again becomes compositional:

```
genCfgWhile :: CfgGen (Label, Label) −> CfgGen (Label, Label)
genCfgWhile mBodyEnterExit = do
  (enter, exit) <− makeEnterExit
  pushLoop enter exit
  (bodyEnter, bodyExit) <− mBodyEnterExit
  popLoop
  connect enter bodyEnter
  connect bodyExit exit
  connect enter exit
  return (enter, exit)

genCfgBreak :: CfgGen (Label, Label)
genCfgBreak = do
  (enter, exit) <− makeEnterExit
  (breakTarget, continueTarget): rest <− gets loopStack
  connect enter breakTarget
  return (enter, exit)

genCfgContinue :: CfgGen (Label, Label)
genCfgContinue = do
  (enter, exit) <− makeEnterExit
  (breakTarget, continueTarget): rest <− gets loopStack
  connect enter continueTarget
  return (enter, exit)
```

It is now straightforward to define a language-specific genCfg function which defers to these cases.

```
genCfg :: Stmt −> CfgGen (Label, Label)
genCfg (While _ s) = genCfgWhile (genCfg s)
genCfg Break       = genCfgBreak
genCfg Continue    = genCfgContinue
−− cases for Assign, If, Block not shown
```

2.4 Finer-Grained CFGs

Control-flow graphs are best known from their use in compilers, where the definition "a CFG is a directed graph of basic blocks" has become sacrosanct. We have already departed from this somewhat by not collapsing consecutive statements into basic blocks, and by using two nodes per statement.

Yet smaller units also have control-flow (e.g.: f() + g() evaluates f() before g()), and we argue that, for static analysis and transformation tools, finer-grained control-flow graphs are a better choice. Indeed, many static analysis frameworks, such as IncA [21] and Polyglot [15], already use expression-level CFGs. We can argue this point at length, but, for the setting of this paper, the following observation resolves the issue decisively: in the examples of §1, the CFG-based inserter must find the predecessors of the *condition* of the loop and conditional. It is thus **not possible** to build the CFG-based inserter without a finer-grained CFG.

In the remainder of this paper, all control-flow graphs will be expression-level.

3 Language-Modular CFG Generation

Having discovered and eliminated the bottlenecks to modularity, we can now aggressively factor out and automate common parts. In this section, we present the remainder of our approach to CFG-generation, and use it to create a complete CFG-generator for a language with 17 node types in only 19 lines of language-specific code.

3.1 Background: Unfixed Data Types, With Sorts

In this subsection, we develop the modular data types used by the generic-programming machinery in the rest of the section, particularly: unfixed data types, which make structured recursion schemes possible; and sorted (unfixed) data types, which allow language engineers to direct the generic cases.

The modular data types we present here are simpler than the actual representation used in Cubix. Cubix's defining feature is that it is possible to give different languages overlapping syntactic definitions. But interestingly, for this application, having languages share syntax increases code reuse only marginally over making the cases be separable into functions. We can thus omit discussion of splitting data types into fragments, simplifying our presentation.

The definitions presented here are a diminished variant of multi-sorted data types à la carte, the style of modular datatypes used by Cubix. Ordinary data types à la carte are explained in Swierstra [20]; multi-sorted ones are described in Bahr and Hvitved [1] and Koppel et al. [13].

Unfixed Data Types Recursive data types are self-entangled data types, and entangled data types means entangled code. It has been well-known that recursive datatypes hinder code reuse. The idea of unfixed data types is to *delay the recursion* as long as possible, allowing most functions to be written on non-recursive data types. This operates by replacing all recursive instances in a data type definition with some type variable e, which can be read "the type of child nodes, to be determined later." The resulting definition is a *signature* describing the space of allowed nodes, but not yet the type of trees comprised of these nodes.

```
data TermSig e = Assign Var Exp
               | If Exp e e
               | While Exp e
```

The original datatype is recovered by taking the type-level fixpoint of TermSig ("tying the recursive knot").

```
data Fix f = In (f (Fix f))

type Term = Fix TermSig
```

The Term type above is now isomorphic to the following more conventional definition:

```
data Term = Assign Var Exp
          | If Exp Term Term
          | While Exp term
```

An immediate advantage of unfixed data types is that they enable *structured recursion schemes*. We stated in §2 that the new CFG-generator design is compositional, meaning that the CFG of a node is a function of those of its subterms. The *catamorphism* recursion scheme formalizes this. A recursive function like genCfg, implemented as a catamorphism, is built out of a function of type TermSig (m (Node, Node)) → m (Node, Node), for some appropriate monad m. This is called an *algebra* of the TermSig functor with *carrier* m (Node, Node). The input to this function is a term where each child node has been replaced by a value of type CfgGen (CfgNode, CfgNode) (i.e.: a command that, when run, produces a CFG and its enter/exit nodes). The catamorphism construction combinator cata then lifts this algebra into a recursive function over an entire term.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

-- Functor instance for TermSig omitted (would be auto-generated)

cata :: (Functor f) => (f a -> a) -> Fix f -> a
cata f (In t) = f (fmap (cata f) t)
```

As a simple example of a catamorphism, consider this function to compute the number of statements in a term. Notice how there are no explicit recursive calls; instead, sizeF inputs a "pre-digested" term, in which each child has been replaced by its size.

```
sizeF :: TermSig Int -> Int
sizeF (Assign _ _) = 1
sizeF ( If _ n1 n2) = 1 + n1 + n2
sizeF (While _ n)  = 1 + n

size :: Term -> Int
size = cata sizeF
```

We shall demonstrate a modular genCfg built in this fashion.[2]

Sorts The definitions of Term and TermSig above break the recursion in statements, but still treat expressions normally. With that definition, one can write a statement-level CFG-generator as a catamorphism. Yet, as discussed in §2.4, we desire an expression-level CFG-generator. The next step is to extend unfixed data-types to multi-sorted terms.

One approach is to place expression constructors like Add and VarExp as alternate constructors of Term, putting them on equal footing with If and While. The obvious downside is that this type permits ill-sorted terms. The other approach is to generalize the operation of unfixing a recursive datatype to the unfixing of mutually-recursive datatypes. We choose the latter, which has the added benefit that it makes the sort of a term available for use in generic programming.

The code below constructs Term to be a family of types, one per each sort. ExpL and StmtL are type-level tags. The types Term ExpL and Term StmtL represent terms of sort expression and statement, respectively. The TermSig constructor similarly takes a parameter for the family of types of subterms of each sort; Fig. 3 gives a pictoral representation of a similar type

[2]In the actual implementation, we use a recursion scheme which also permits cases to inspect the children themselves. This is rarely used.

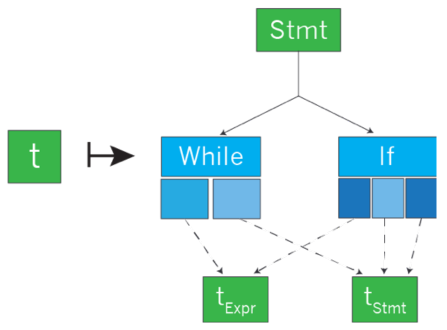

Figure 3

constructor. Whereas the previous definition of TermSig had kind $* \to *$, a type constructor which takes the type of child nodes and gives the type of terms with those children, the new definition has kind $(* \to *) \to (* \to *)$, a type constructor which takes a family of types of child nodes, one for each sort, and returns a family of types of terms, one for each sort.

```
data ExpL
data StmtL

data TermSig e  i  where
  Add     ::  e  ExpL -> e  ExpL -> TermSig e  ExpL
  Lt      ::  e  ExpL -> e  ExpL -> TermSig e  ExpL
  VarExp  ::  Var                -> TermSig e  ExpL

  Assign  ::  Var     -> e  ExpL                 -> TermSig e  StmtL
  If      ::  e  ExpL -> e  StmtL -> e  StmtL -> TermSig e  StmtL
  While   ::  e  ExpL -> e  StmtL               -> TermSig e  StmtL

data Fix  f  l  =  In  (f  (Fix  f)  l)

type Term =  Fix  TermSig
```

All operations on unsorted terms have corresponding variants on sorted terms. We do not dwell on these. Though they may look intimidating, they are essentially the same as their unsorted variants, but with an extra type parameter throughout.

```
class  HFunctor h where
  hfmap ::  ( forall  l .  f  l  -> g   l )  -> ( forall  l .  h  f  l  -> h g  l )

hcata  ::  (HFunctor h)  => ( forall  l .  h  a  l     -> a  l )
                          -> ( forall  l .  Fix  h  l  -> a  l )
```

We shall also need the hfold function in the HFoldable typeclass, an analogue of Haskell's standard fold and Foldable. Also helpful is the type-level K (constant) combinator, needed for embedding types that do not have a sort parameter l inside a multi-sorted tree.

```
newtype K a l = K {  unK  ::  a}

class  (HFunctor h)  => HFoldable h where
  hfold  :  (Monoid m) => h (K m) l  -> m
```

3.2 A Generic Setup

We now proceed to develop a language-modular infrastructure for CFG-generators. We begin by defining the language IMP, which we defined as a showcase for our techniques. IMP features two kinds of control effects, break and goto, and contains both lambdas and function definitions, constructs which define new control-flow contours (i.e.: separate intraprocedural CFGs). It also has type declarations, for the sole purpose of demonstrating how our approach handles nodes which do not take part in computation. Its signature is below. IMP has 5 base sorts: expressions, lambdas, statements, function definitions, and types; we repurpose Haskell's list constructor to create new sorts for lists of statements and function definitions. A top-level IMP program is then given by a value of type Imp [FunDefL].

```
type Var  =  String

data ExpL;  data TypeL;  data LambdaL; data StmtL; data FunDefL

data ImpSig e  l  where
  Add        ::  e  ExpL -> e  ExpL -> ImpSig e  ExpL
  Lt         ::  e  ExpL -> e  ExpL -> ImpSig e  ExpL
  VarExp     ::  Var                -> ImpSig e  ExpL
  LambdaExp :: e  LambdaL           -> ImpSig e  ExpL
  CallExp    ::  e  ExpL -> e  ExpL -> ImpSig e  ExpL

  IntType    ::  ImpSig e  TypeL
  BoolType  ::  ImpSig e  TypeL

  Lambda :: Var  -> e  TypeL -> e  ExpL -> ImpSig e  LambdaL

  Assign     ::  Var     -> e  ExpL                 -> ImpSig e  StmtL
  If         ::  e  ExpL -> e  StmtL -> e  StmtL -> ImpSig e  StmtL
  While      ::  e  ExpL -> e  StmtL               -> ImpSig e  StmtL
  Break      ::                                        ImpSig e  StmtL
  GotoLabel  ::  String                              -> ImpSig e  StmtL
  Goto       ::  String                              -> ImpSig e  StmtL
  Block      ::  e  [StmtL] ->                       -> ImpSig e  StmtL

  EmptyStmtList ::                                  ImpSig e  [StmtL]
  ConsStmt       ::  e  StmtL -> e  [StmtL] -> ImpSig e  [StmtL]

  FunDef  ::  Var  -> e  StmtL -> ImpSig e  FunDefL

  EmptyFunDefs ::                                    ImpSig e  [FunDefL]
  ConsFunDef    ::  e  FunDefL -> e [FunDefL] -> ImpSig e  [FunDefL]

data Fix  f  l  =  In  (f  (Fix  f)  l)

type Imp  =  Fix  ImpSig
```

Somewhat unsatisfying are the constructors for the list sorts: EmptyStmtList, ConsStmt, EmptyFunDefs, and ConsFunDef. These actually are not needed in the real implementation, as CUBIX has generic support for treating lists of terms as ordinary tree nodes, but we opted not to include that feature in this presentation.

We now state the core monad operations needed in CFG construction.

```
type CfgNode = Int

class MonadCfgGen m where
  makeEnterExit :: m (CfgNode, CfgNode)
  connect       :: CfgNode -> CfgNode -> m ()
```

These operations can be implemented on any state monad whose state contains certain fields, discussed in the next section.

```
class HasCfgGenState s where
  -- defined in next section

instance (HasCfgGenState s) => MonadCfgGen (State s)
```

We define an EnterExitPair type as in §2. We abbreviate it to EEP, as it is used quite frequently.

```
type EEP = Maybe (CfgNode, CfgNode)

combineEnterExit :: (MonadCfgGen m) => EEP -> EEP -> m EEP
```

A useful fact is that enter/exit pairs, in the presence of a monad capable of adding edges, actually form a monoid under combineEnterExit. If a, b, and c are monadic values which evaluate to enter and exit nodes of three separate CFG fragments, then a <>b <>c is a monadic value which, when evaluated, connects a, b, c in sequence, returning the enter and exit of the combined graph.

```
instance (MonadCfg m) => Monoid (m EEP)
  mempty   = return Nothing
  ma <> mb = do a <- ma
                b <- mb
                combineEnterExit a b
```

We shall soon give the general interface for CFG generators, as a catamorphism whose result is a stateful computation. One wrinkle is that values in the State monad do not have the right type for use in a catamorphism. They must be modified to take an extra sort parameter, which can be done by wrapping them with the K combinator.

```
type HState s a = K (State s a)
```

A downside is that using HState requires frequent unwrapping and rewrapping in order to use the monad operations. We are now ready to give the final interface for language-specific CFG-generators:

```
class (HFoldable f) => ConstructCfg f s where
  constructCfg' :: f (HState s EEP) l  -> HState s EEP l

constructCfg :: (ConstructCfg f s) => Fix f l -> State s ()
constructCfg t = void (unK (hcata constructCfg' t))
```

3.3 Open Products for Control Effects

Different control effects need different state. As we have shown in §2, break and continue statements demand maintenance of a stack of break and continue targets. Goto statements, meanwhile, demand a map of label names to targets. As languages may have their own exotic control effects, there can be no common CFG-generation state shared by all languages.

Our solution is to express the current CFG-generation state as an *open product*. Open products permit the definition of functions which run on **any state which has a certain field**. The generic constructCfgGoto case, for instance, runs on any state that contains a map of goto labels to targets.

A common way to implement open products is with lenses [5]. A lens from a to b is a pair of a getter a → b, which looks up a field of type b from a value of type a, and a *setter* a → b → a, which replaces said field with a modified value.

```
type Lens a b = (a -> b, a -> b -> a)
```

There are already many Haskell libraries for automatically generating lenses of this type (or one isomorphic to it) [10, 11, 23]. We sweep under the rug which one is used, and instead use the pseudocode magicMakeLens to denote the proper invocation of whichever library. We can now define the state needed for each kind of control effect: the current graph and an node-ID generator for general CFG-construction, a stack of break targets for loops, and a goto-label map for goto statements.

```
type NodeGen = Int

data CfgGenState = CfgGenState { curGraph :: Graph
                               , nodeGen  :: LabelGen }

class HasCfgGenState s where
  cfgGenState :: Lens s CfgGenState

class (HasCfgGenState s) => HasBreakStack s where
  breakStack :: Lens s [CfgNode]

class (HasCfgGenState s) => HasGotoMap s where
  gotoMap :: Lens s (Map String CfgNode)
```

We are now able to define language-agnostic CFG construction functions for all relevant statements, similar to the examples in §2.

```
constructCfgIf :: (HasCfgGenState s)
               => HState s EEP i -> HState s EEP j
               -> HState s EEP k -> HState s EEP l

contsructCfgBreak :: (HasBreakStack s) => HState s EEP l
constructCfgWhile :: (HasBreakStack s)
          => HState s EEP i -> HState s EEP j -> HState s EEP k

constructCfgGoto      :: (HasGotoMap s) => String -> HState s EEP l
constructCfgGotoLabel :: (HasGotoMap s) => String -> HState s EEP l
```

Of note is that, for forward-gotos, constructCfgGoto must allocate a CFG node for the GotoLabel statement before it is seen.

3.4 Dispatching with Easy Cases

Most AST nodes have no interesting control-behavior. In this section, we define a way to deal with the common cases without writing any custom code.

We group these non-interesting nodes into three categories, discriminating by sort. Computation sorts describe those nodes which have control flow. These AST nodes will be given their own CFG nodes. Suspended-computation sorts, abbreviated "suspend sorts," are those nodes whose bodies may have control flow, but which should be in a separate contour, not connected to the CFG nodes of the surrounding context. Lambda expressions are a typical example. Finally, all other sorts are considered to not participate in the computation. For a node n of such a sort, its children, if any exist, will be sequenced and connected to the surrounding CFG nodes, as if n did not exist.

We define type families to specify these categories. For each language signature f, ComputationSort f returns the computation sorts for that language as a type-level list, and similar for suspend sorts. It is then possible to implement dynamic checks to see if a term is of a computation sort, using standard type-level programming techniques.

```
type family ComputationSorts (f :: (* -> *) -> * -> *) :: [*]
type family SuspendSorts     (f :: (* -> *) -> * -> *) :: [*]

isComputationSort :: Fix f l -> Bool
isSuspendSort     :: Fix f l -> Bool
```

As a preview, here are the definitions for ImpSig.

```
type instance ComputationSorts ImpSig = '[ExpL, StmtL, [StmtL]]
type instance SuspendSorts ImpSig = '[LambdaL, FunDefL]
```

Note that. as Imp has no expressions with interesting control-flow (e.g.: short-circuiting operators), simply removing ExpL from the list of computation sorts would change the CFG-generator definition to instead produce statement-level CFGs. We intentionally include [StmtL] in the list of computation sorts, giving ConsStmt nodes their own CFG nodes, as this will be useful when building the CFG-based inserter.

We now turn to defining the generic cases, beginning with the case for computation sorts. Thanks to our careful definitions of HState and the monoid instance for enter/exit pairs, the hfold function will run and sequence all children. The remainder of this function allocates CFG nodes for the current term, and connects them to the children.

```
constructCfgCompSort :: (HFoldable f)
                     => f (HState s EEP) l -> HState s EEP l
constructCfgCompSort t = K do
  (enter, exit) <- makeEnterExit
  let left  = return (Just (enter, enter))
  let body = hfold t
  let right = return (Just (enter, enter))
  left <> body <> right
```

Notice also what happens when there are no children: body evaluates to Nothing, and the final line simply connects enter to exit.

The final definition of constructCfgDefault dispatches on the sort of a term. For suspend sorts, it sequences the CFGs of all children, but returns an empty enter/exit pair, so that nothing will connect to the children. For non-computation nodes, it sequences the CFGs of the children and returns them

```
constructCfgDefault :: (HFoldable f)
                    => f (HState s EEP) l -> HState s EEP l
constructCfgDefault t =
  if isComputationSort t then
    constructCfgCompSort t
  else if isSuspendSort t then
    K (hfold t >> return Nothing)
  else
    K (hfold t)
```

Notice that, when t is IntType or BoolType, constructCfgDefault t has no effects and evaluates to K Nothing. Notice also how it handles terms of sort [StmtL] (it sequences them, also creating CFG nodes for ConsStmt terms) and terms of sort [FunDefL] (it runs them independently).

3.5 Victory

We now give full source code for the CFG generator for Imp, validating our claim to construct a CFG-generator for it in only 19 lines of language-specific code. In fact, there are only 18 non-empty lines in the example below. Not shown is the code to generate lenses in whichever library is chosen; in Edward Kmett's lens library [11], this would be a 1-line Template Haskell invocation, makeLenses ''ImpSigCfgState, yielding our final count of 19 lines.

```
type instance ComputationSorts ImpSig = '[ExpL, StmtL, [StmtL]]
type instance SuspendSorts ImpSig = '[LambdaL, FunDefL]

data ImpSigCfgState = { breakStack  :: [CfgNode]
                      , gotoMap     :: Map String CfgNode
                      , cfgGenState :: CfgGenState }

instance HasBreakStack ImpSigCfgState where
  breakStack = magicMakeLens
instance HasGotoMap ImpSigCfgState where
  gotoMap = magicMakeLens
instance HasCfgGenState ImpSigCfgState where
  cfgGenState = magicMakeLens

instance ConstructCfg ImpSig ImpSigCfgState where
  constructCfg' (While e s)     = constructCfgWhile e s
  constructCfg' (If e s1 s2)    = constructCfgIf e s1 s2
  constructCfg' Break           = constructCfgBreak
  constructCfg' (GotoLabel s)   = constructCfgGotoLabel s
  constructCfg' (Goto s)        = constructCfgGoto s
```

```
constructCfg' t              = constructCfgDefault t
```

With this code, one can now run constructCfg t for any Imp program t and obtain a control-flow graph.

The brevity of this example came in part because this language has no "unusual" nodes: each node either fits one of the defaults, or is a common control-flow construct for which it is reasonable to place the behavior in a language-agnostic function. In Cubix, we implemented CFG-generators for 5 real languages, all of which did have "unusual" nodes requiring custom code. Yet they also all benefited greatly from the predefined common cases, For JavaScript, for instance, we were still able to define a complete CFG-generator passing an exhaustive test suite in only 79 lines of language-specific code, compared to 184 distinct node types.

4 CFG-Based Program Transformation

In this section, we explain our new transformation primitive, CFG-based insertion. The code in this section involves more algorithmic details and bookkeeping compared to that of the previous sections. Consequently, whereas the code in the previous sections would run verbatim except for its use of lenses, the code in this section tends slightly more towards pseudocode.

The goal of this section is to define the dominatingPrepend operation. dominatingPrepend targ s modifies the program to ensure that s always runs before targ by inserting s at the first possible predecessor along every control path.

```
dominatingPrepend :: (InsertAt f l) => Fix f i
                                    -> Fix f l
                                    -> CfgInsertion ()
```

There are many variants of this operation which we do not present here. Aside from the obvious dominatingAppend, which would instead ensure s runs *after* targ, there are variants in how to behave if multiple insertions are performed at the same point, and variants that can choose to treat each insertion point differently (to e.g.: only declare a temporary variable once).

The dominatingPrepend operation rests on both the presence of a CFG and the InsertAt interface. The InsertAt interface provides the twin operations of "is it possible to insert this node at this point" and of performing the actual insertion. The utility of this interface is that it describes both the mundane operation of inserting a statement into a list of statements **as well as** more peculiar ones which cleave open a statement to insert another.

A program point is identified by an AST node and an *evaluation point*; these shall also correspond to CFG nodes. Most AST nodes only have two evaluation points: before (Enter) and after (Exit) it executes. But a few, such as Python if-elif-elif-...-else chains, also have intermediate evaluation points (they correspondingly have more than two CFG nodes!).

```
data EvaluationPoint = Enter | Exit | Intermediate Int
```

We now define the InsertAt interface. canInsertAt @l p targ returns whether a term of sort l[3] can be inserted to run at the program point defined by p and targ. insertAt p s targ modifies targ to perform the insertion.

```
class InsertAt f l where
  canInsertAt :: EvaluationPoint -> Fix f i -> Bool
  insertAt    :: EvaluationPoint -> Fix f l -> Fix f i -> Fix f i
```

In the previous section, it sufficed for demonstration to use raw integer labels as CFG nodes, with no way to map between corresponding CFG and AST nodes. This section requires CFG nodes with a little bit more structure (as is the case in the actual implementation), with the ability to map between AST and CFG nodes, where each CFG node corresponds to an AST node / evaluation point pair. We shall also need to refer to terms of unknown sort, which we do with the existential combinator E: E (Fix f) refers to a term (Fix f l) for some unknown sort l.

```
data E f = forall i. E {unE :: f i}


type Graph f
type CfgNode f
evalPoint  :: CfgNode f -> EvaluationPoint
termFor    :: CfgNode f -> E (Fix f l)
predecessors :: CfgNode f -> [CfgNode f]


nodeFor :: Graph f -> EvaluationPoint -> Fix f l -> Maybe CfgNode
```

Note that nodeFor is not actually implementable as written, for it cannot distinguish between identical terms at different program points. This does not substantively alter our work, as it is possible to (as is actually done in Cubix) modularly add label annotations via an alternate fixpoint operation

```
data FixLab f l = In (f (FixLab f l), Label)
```

but we choose to omit such bookkeeping from this presentation. An alternative version uses paths from the root to identify unique subtrees.

We also define this helper function:

```
canInsertAtNode :: (InsertAt f l) => CfgNode f -> Bool
canInsertAtNode n = canInsertAt @l (evalPoint n) (unE (termFor n))
```

We now begin to implement CFG-insertion. We shall demonstrate on a tiny model of Python called Worm, capable of expressing the two special cases described in §1.

```
data WormSig e l where
  While :: e ExpL -> e StmtL -> WormSig e StmtL
  Continue :: WormSig e Stmtl
  IfElse  :: e ExpL -> e StmtL -> e StmtL -> WormSig e StmtL
  IfElifElse :: e ExpL -> e StmtL -- if
             -> e ExpL -> e StmtL -- elif
                       -> e StmtL -- else
                       -> WormSig e StmtL


  ConsStmt :: e StmtL -> e [StmtL] -> WormSig e [StmtL]
  EmptyStmt ::                        WormSig e [StmtL]
```

[3] The @l argument is a Haskell explicit type parameter.

```
  -- other cases not shown
```

As we will need to construct new terms, as is customary with unfixed data types, we shall define *smart constructors* as syntactic sugar.

```
iWhile :: Fix WormSig ExpL -> Fix WormSig StmtL
                           -> Fix WormSig Stmtl
iWhile e s = In (While e s)

-- iContinue, iIfElse , etc not shown
```

We begin by defining `InsertAt`. This benefits from an operation to check the sort of a term: `isSort @StmtL t` tests if `t` is a statement. If it passes, an actual implementation would need to cast `t` to `Fix f StmtL` in order to use it; we gloss over this detail.

```
isSort :: Fix f i -> Bool


instance InsertAt WormSig StmtL where
  canInsertAt (Intermediate 0) _ (In ( IfElifElse _ _ _ _ _)) = True
  canInsertAt Enter _ t = isSort @StmtL t
  canInsertAt _     _ _ = False

  insertAt (Intermediate 0) stmt (In ( IfElifElse c1 S1 c2 s2 c3)) =
    iIfElse c1 s1
        (iConsStmt stmt
            (iConsStmt ( iIfElse c2 s2 s3)
                iEmptyStmt))

  insertAt Enter stmt t | isSort @StmtL t = iConsStmt stmt t
  insertAt _     _    _                   = error "Unreachable"
```

In the real implementation, most `InsertAt` cases are variants of inserting a statement into a list of statements. In addition to this case for if-elif-else, other special cases include replacing a singleton statement with a block, and splitting a Python **with** (e.g.: inserting code between the calls to `A()` and `B` in **with** `a as A(), b as B():`).

We now turn to the components of the implementation of `dominatingPrepend`. `satisfyingBoundary` is a standard graph algorithm which finds all predecessors of a node which satisfy some predicate. When used with `canInsertAt`, it locates the places at which to perform the insertion.

```
satisfyingBoundary :: Graph f -> CfgNode f -> (CfgNode f -> Bool)
                                            -> Set (CfgNode f)
satisfyingBoundary cfg start pred = go Set.empty start
  where
    go :: Set (CfgNode f) -> Cfgnode f -> Set (CfgNode f)
    go seen x = if Set.member seen x then
                  Set.empty
                else if pred x then
                  Set. singleton x
                else
                  fold (map (go (Set.insert x seen))
                            (predecessors x))
```

Because this implementation runs on immutable trees, the function `dominatingPrepend` must run in two phases: first marking the set of insertions to perform, and then performing them. We define the monad to hold the necessary state:

```
data CfgInsertionState f l =
  CfgInsertionState
    { pendingInsertions :: Map (CfgNode f) (Fix f l)
    , cfg :: Graph }

type CfgInsertion f l = State ( CfgInsertionState f l)
```

`dominatingPrepend` simply finds the insertion points and marks the intended insertion.

```
dominatingPrepend :: ( InsertAt f l) => Fix f i
                                     -> Fix f l
                                     -> CfgInsertion f l ()
dominatingPrepend target item = do
  state <- get
  let curCfg = cfg state
  let startNode = fromJust (nodeFor curCfg Enter)
  let insertionPoints =
        satisfyingBoundary curCfg startNode canInsertAtNode
  let insertions =
        Set. fold (Set.map (\node -> Map.singleton node item)
                           insertionPoints )
  puts (\s -> s {pendingInsertions = insertions })
```

For performing the insertions, we assume a primitive `rewriteAt`. `rewriteAt x t f` finds the subtree in `t` equal to `x`, and then rewrites it with `f`, returning an updated `t`. Similar to the `nodeFor` function, a real implementation would require either labeled terms or paths from the root.

```
rewriteAt :: Term f i -> Term f j
          -> (Term f i -> Term f i ) -> Term f j
```

The final step loops over the intended insertions and performs them.

```
performInsertions :: ( InsertAt f l) => Fix f i
                                     -> Graph f
                                     -> Cfginsertion f l ()
                                     -> Fix f i
performInsertions t g m =
  let initialState = CfgInsertionState Map.empty g
  let insertions = pendingInsertions (execState m initialState )

  Map.foldrWithKey
    (\t' node item ->
        let p = evalPointNode in
        rewriteAt p (termFor node) t' (insertAt p item))
    t
    insertions
```

5 Implementation

Within this section, the term "significant lines" refers to the number of lines in a file after removing the file prologue (i.e.: import statements), comments/docstrings, and blank lines. All line counts (SLOC) refer to significant lines.

Table 1. Line and token counts of CFG-generators

Language	Cubix SLOC	Cubix tokens	Comparison	Comparison SLOC	Comparison Tokens
C	168	1837	Clang* [17]	1158	7962
Java	107	1374	Polyglot [18]	573	3888
JavaScript	79	1181	ast-flow-graph [8]	400	2924
Lua	148	1522	None found	N/A	N/A
Python	110	1261	StatiCFG [3]	232	2525
Infrastructure	475	4814			
Total	1087	11989		2463	17299

*C-relevant portions only

Language-Modular CFGs We implemented CFG generation in the language-modular style in Cubix, and built CFG-generators for C, Java, JavaScript, Lua, and Python, along with a thorough test suite with over 3000 SLOC of unit tests, together with 66 end-to-end tests, consisting of an input program and its full expected graph. Even though most CFG-generation is performed by language-agnostic code, the unit tests are language-specific, giving us high-confidence that our approach is flexible enough to adapt to the peculiarities of each language while still saving substantial labor.

Table 1 gives the size of the CFG generators built in Cubix. For each language, we searched for a comparison CFG generator, restricted to those which build a CFG from the original AST and not for an IR. We found comparison generators for all languages except for Lua, after having searched for one in Lua static analysis tools and in the source code of the Lua.org and LuaJIT implementations. We include CFG construction code, but not code for the graph data type. We also report token counts, computed by a lexer for the relevant implementation language, which tend to be more robust than line counts.

This is an imperfect comparison in every way. The comparison CFG-generators range in quality from "part of a major compiler" (Clang) to "visualization tool with 20 stars on Github" (ast-flow-graph). All of the Cubix generators are expression-level, whereas all comparison generators except Polyglot are statement-level. On the other hand, some of the comparison generators also do basic-block compression. We also biased these counts against Cubix by including infrastructure for Cubix but not the comparison generators; the relevant equivalents of §3.2 total 1659 SLOC and $12,536$ tokens across the 4 comparison generators. Finally, while they are generally better than line counts, token counts are not known to be comparable across languages.

Nonetheless, this table gives strong evidence that our approach provides substantial labor savings compared to a single-language baseline lacking generic programming techniques. Our combined generators take approximately 1100 lines compared to 2400 for the comparison generators— and this is excluding a comparison Lua generator. And the per-language marginal cost is over 3x fewer lines — and would be even better if we included each other generator's infrastructure.

CFG-Based Insertion We implemented a CFG-based inserter in Cubix in 142 SLOC. We used it to implement two of the semantics-preserving transformations described in the original Cubix paper: the test-coverage and three-address code transformations (the latter is strictly more complicated than the sanitization example in §1). These transformations attained full semantics-preservation as measured by compiler test suites totalling over 5000 test files, and handled all the special cases shown in the original Cubix paper. For space reasons, we refer the reader to the Cubix paper [13] for details.

6 Related Work

There have been a few works on techniques for constructing control-flow graphs. FlowSpec [19], a component of the Spoofax language workbench [9], contains a DSL for specifying control-flow graphs in terms of single-pushout graph rewrite rules [14] reminiscent of the GrGen graph-rewriting language [6]. Though it yields compact definitions, it has no support for generic programming. There has also been recent work on automatically deriving CFG-generators from operational semantics [12]. Though their technique has only been applied to toy languages, we found their general theory of CFGs useful for resolving debates on how to construct the graph for some constructs. Semantic Designs DMS [2] has a DSL based on attribute grammars for constructing CFGs. The results are quite verbose; their Java CFG-generator is over 5000 lines.

Despite extensive work on advanced techniques in program transformation [22], there has been a paucity on work which uses a graph to aid in transformation. We only know of two works in this category. The more well-known one is Coccinelle [16]. The capabilities of Coccinelle can be succinctly described as: simultaneous associative matching/rewriting of multiple tree patterns, connected by arbitrary control-flow paths. In a personal conversation, one of the Coccinelle creators summarized "We use the CFG in matching, but not in rewriting." Though it provides an ellipsis ("...") primitive which matches arbitrary sets of control-flow paths, each end of a path is ordinary tree rewriting. Coccinelle would still need many cases to handle the examples of §1, though "..." may help with continue statements. More relevant and more obscure is a technique of Griswold [7] used in the world's first refactoring tool. Though based on program-dependence graphs [4] rather than CFGs, it shares the characteristic of more closely tying the graph and rewriting, by providing simultaneous updates on the AST and PDG.

7 Conclusion

As the diversity of technology grows, every technique to reduce the cost of building tools is a step towards making advanced tools a part of daily programming. By teasing out the commonalities in CFGs and transformations, the two techniques of this paper offer significant savings in their narrow domains. Our full implementation is available anonymously from: https://github.com/uraul/language_modular_cfg.

References

[1] Patrick Bahr and Tom Hvitved. 2011. Compositional Data Types. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011.* 83–94.

[2] Ira D Baxter, Christopher Pidgeon, and Michael Mehlich. 2004. DMS®: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the 26th International Conference on Software Engineering.* IEEE Computer Society, 625–634.

[3] Aurelien Coet. 2020. Staticfg. https://github.com/coetaur0/staticfg/tree/9948ab8574c254f69564da46c0ca30e5ac0c35a5. (2020).

[4] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[5] J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 3 (2007), 17–es.

[6] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. 2006. GrGen: A Fast SPO-Based Graph Rewriting Tool. In *International Conference on Graph Transformation.* Springer, 383–397.

[7] William G. Griswold. 1993. Direct Update of Data Flow Representations for a Meaning-Preserving Program Restructuring Tool. In *SIGSOFT '93, Proceedings of the First ACM SIGSOFT Symposium on Foundations of Software Engineering, Los Angeles, California, USA, December 7-10, 1993.* 42–55. https://doi.org/10.1145/256428.167063

[8] Julian Jensen. 2020. ast-flow-graph. https://github.com/julianjensen/ast-flow-graph/tree/0c669d1dad54fa28004741a7e9cf82eee8d683e2. (2020).

[9] Lennart CL Kats and Eelco Visser. 2010. *The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs.* Vol. 45. ACM.

[10] Csongor Kiss, Matthew Pickering, and Nicolas Wu. 2018. Generic Deriving of Generic Traversals. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–30.

[11] Edward A. Kmett. 2020. lens: Lenses, Folds and Traversals. http://hackage.haskell.org/package/lens-4.19.2. (April 2020).

[12] James Koppel, Kearl Jackson, and Armando Solar-Lezama. [n. d.]. Synthesizing Control-Flow Graph Generators from Operational Semantics. http://www.jameskoppel.com/files/papers/mandate_111019.pdf. ([n. d.]).

[13] James Koppel, Varot Premtoon, and Armando Solar-Lezama. 2018. One Tool, Many Languages: Language-Parametric Transformation with Incremental Parametric Syntax. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–28.

[14] Michael Löwe. 1993. Algebraic Approach to Single-Pushout Graph Transformation. *Theoretical Computer Science* 109, 1-2 (1993), 181–224.

[15] Nathaniel Nystrom, Michael R Clarkson, and Andrew C Myers. 2003. Polyglot: An Extensible Compiler Framework for Java. In *International Conference on Compiler Construction.* Springer, 138–152.

[16] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and Automating Collateral Evolutions in Linux Device Drivers. *Acm sigops operating systems review* 42, 4 (2008), 247–260.

[17] LLVM project authors. 2020. Clang. https://github.com/llvm/llvm-project/tree/eed333149d178b69fdaf39b9419b7ca032520182. (2020).

[18] Polyglot project authors. 2020. Polyglot. https://github.com/polyglot-compiler/polyglot/tree/6235855368ce3b0ab27cb29cd117ca5d0fba54e7. (2020).

[19] Jeff Smits and Eelco Visser. 2017. FlowSpec: Declarative Dataflow Analysis Specification. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering.* 221–231.

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

[21] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: A DSL for the Definition of Incremental Program Analyses. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on.* IEEE, 320–331.

[22] Eelco Visser. 2005. A Survey of Strategies in Rule-Based Program Transformation Systems. *Journal of Symbolic Computation* 40, 1 (2005), 831–873.

[23] Sebastiaan Visser, Erik Hesselink, Chris Eidhof, and Sjoerd Visscher. 2020. fclabels: First class accessor labels implemented as lenses. http://hackage.haskell.org/package/fclabels-2.0.5. (May 2020).