Streamlining Control Flow Graph Construction with DCFlow

Mark Hills

East Carolina University, Greenville, North Carolina, USA

Abstract. A control flow graph (CFG) is used to model possible paths through a program, and is an essential part of many program analysis algorithms. While programs to construct CFGs can be written in meta-programming languages such as Rascal, writing such programs is currently quite tedious. With the goal of streamlining this process, in this paper we present DCFlow, a domain-specific language and Rascal library for defining control flow rules and building control flow graphs. Control flow rules in DCFlow are defined declaratively, based directly on the abstract syntax of the language under analysis and a number of operations representing types of control flow. Standard Rascal code is then generated based on the DCFlow definition. This code makes use of the DCFlow libraries to build CFGs for programs, which can then be visualized or used inside program analysis algorithms. To demonstrate the design of DCFlow we apply it to Pico—a very simple imperative language—and to a significant subset of PHP.

1 Introduction

A control flow graph [2] (CFG) is used to model all possible paths (the flow of control) through a program. Nodes in the graph either represent individual constructs in the program, such as individual statements or expressions (referred to collectively as *instructions* below), or are synthesized based on program information. An example of the latter is nodes created to provide a unique exit from a function in languages with `return` statements that can occur anywhere in the function body. Edges in the graph represent the actual flow of control through the program, taking account of the evaluation order and the impact of various control constructs, such as conditionals, loops, and `goto`s.

Programs to build CFGs can be written in meta-programming languages such as Rascal [15,16]. However, the process of writing such programs, especially for larger languages, can be quite tedious. For example, the code currently used to extract control flow graphs from PHP programs, developed as part of the PHP AiR project [7], is 1,583 lines of Rascal,[1] including a large amount of boilerplate code to handle similar cases and keep track of information needed to properly

[1] This is calculated using the cloc tool, and is based on counting lines of Rascal code of all modules under `lang::php::analysis::cfg`, available at `https://github.com/cwi-swat/php-analysis/tree/master/src/lang/php/analysis/cfg`.

build the graph. For actively evolving languages, such as PHP, this code also needs to be kept up to date to support new language features.

With the goal of streamlining the process of defining the control flow rules for a programming language and extracting control flow graphs from individual programs, in this paper we present a declarative, domain-specific language for specifying the control flow rules for programming languages—DCFlow, short for **D**eclarative **C**ontrol **Flow**. In DCFlow, control flow is defined at the level of the language's abstract syntax, allowing DCFlow to be used even in cases were a Rascal parser for the language under analysis is not available. Control flow rules are defined declaratively, specified in terms of the AST types and a number of operations representing different types of control flow and CFG nodes. DCFlow definitions are then used to generate standard Rascal code—a combination of custom code based on the DCFlow definition, calls to the DCFlow libraries, and calls to some user-provided code, all written in Rascal. Since DCFlow definitions are translated into standard Rascal code, it is possible to examine, debug, and extend the generated code.

DCFlow makes use of a number of Rascal features, including algebraic data types, reified types, and string templates, described in Section 2. In Section 3 we then describe DCFlow in detail, showing how specific language features in both Pico and PHP are supported. Section 4 then provides an evaluation of DCFlow, comparing it to hand-written Rascal and the DeFacto system [3]. Finally, Sections 5 and 6 present related work and a final discussion with ideas for future work, respectively. Additional information about Rascal and DCFlow can be found online.[2]

2 Enabling Rascal Features

The DCFlow languages makes use of several key Rascal language features: *algebraic data types* for creating user-defined types, including the abstract syntax types; *type literals* and *type reification* to allow meta-level access to Rascal types; *string templates* for code generation; and Rascal support for *custom, Eclipse-based IDEs*. Each of these features is described in more detail below.

2.1 The Rascal Type System

The Rascal type system provides a uniform framework including both built-in and user-defined types, with the latter including both abstract (algebraic) datatypes and grammar non-terminals (also referred to as *concrete* datatypes). The type system is based on a type lattice with **void** at the bottom and **value** (the supertype of all types) at the top. In between are the types for atomic values (**bool**, **int**, **real**, **rat**, **str**, **loc**, **datetime**), types for tree values (**node**, representing named nodes with zero or more children, and defined abstract and concrete datatypes), and composite types with typed elements. Examples of the

[2] See `http://www.rascal-mpl.org` and `http://www.cs.ecu.edu/hillsma`.

```
data TYPE = natural() | string();
alias PicoId = str;
data PROGRAM = program(list[DECL] decls, list[STATEMENT] stats);
data DECL = decl(PicoId name, TYPE tp);

data EXP = id(PicoId name)
  | natCon(int iVal)
  | strCon(str sVal)
  | add(EXP left, EXP right)
  | sub(EXP left, EXP right)
  | conc(EXP left, EXP right) ;

data STATEMENT
  = asgStat(PicoId name, EXP exp)
  | ifElseStat(EXP exp,list[STATEMENT] thenpart,list[STATEMENT] elsepart)
  | whileStat(EXP exp, list[STATEMENT] body) ;
```

Fig. 1. The Pico AST in Rascal, Defined with Algebraic Data Types.

latter are **list[int]**, **set[str]**, **tuple[str,int]**, **rel[int,bool]**, **lrel[loc,int]**, and, for a given non-terminal type `Stmt`, **map[**`Stmt`**,int]**. The **node** datatype is a supertype of both abstract and concrete datatypes, while concrete datatypes are also all subtypes of the **Tree** datatype. Sub-typing is always covariant with respect to these typed elements; with functions, as is standard, return types must be covariant, while the argument types are instead contravariant. For example, for sets, **set[str]** is a subtype of **set[value]**, while for functions, **str(value)** is a subtype of **value(str)**.

2.2 Algebraic Datatypes

Algebraic datatypes (ADTs) in Rascal are defined using the `data` keyword, with one or more constructors defining the alternatives available for building new values of the user-defined type. An example with several related ADTs is shown in Figure 1, which gives the definition of the abstract syntax for the Pico language. Figure 1 defines five new datatypes: `TYPE`, `PROGRAM`, `DECL`, `EXP`, and `STATEMENT`.[3] These datatypes then each include one or more constructors. `TYPE` includes two, `natural` and `string`, that are used to indicate the type of data being declared in a Pico program. These constructors are a form of constant – neither contains any fields. `EXP` defines the different types of expressions in Pico, with fields corresponding to values for identifiers or constants (e.g., `natCon` has field `iVal` which contains a Rascal `int`) or to subexpressions (e.g., `add` has fields `left` and `right` for the left and right operands of a plus expression). ADTs in Rascal are open to extension, allowing new constructors to be added by other modules, and are also inherently recursive. Rascal includes extensive support

[3] `PicoId` is a type alias—`PicoId` is another name for `str`, the Rascal string type.

for pattern matching and term traversal over both built-in and user-defined datatypes, features used extensively in DCFlow to work with program ASTs defined using types like those in Figure 1.

2.3 Type Literals and Reified Types

Reified types make it possible to manipulate types as first-class values that can be passed around, returned, queried and manipulated. Rascal's reification operator creates *self-describing* type values that contain both the reified type and all datatypes used in this type's definition. A type can be reified using the prefix reification operator (#), resulting in a value called a *type literal*. A reified type value contains a symbol to represent the type and a map of definitions for any abstract or concrete datatype dependencies. It is given the type **type[&T]**, where the type parameter `&T` is bound to the type that was reified. For example:

- **#str** produces a literal value `type(\str(),())` of type `type[str]`.
- **#rel[int,loc,str]** produces `type(\rel([\int(),\loc(), \str()]),())` of type **type[rel[int,loc,str]]**.

The **type** data constructor used to build type literals is built in to Rascal; the representations for type symbols and their definitions are defined as Rascal datatypes in a library module, `Type`. Above, the map of definitions was empty: `()`. For abstract or concrete datatypes this map will contain the complete (possibly recursive) abstract datatype or grammar. Given the `EXP` type shown in Figure 1, and focusing just on the `add` constructor:

```
data EXP = ... | add(EXP left, EXP right) | ...;
```

the reified type `#EXP` will produce the following term of type **type[EXP]** (again focusing just on `add`, and with some details elided):

```
type(adt("EXP"),
    (adt("EXP"):choice(...,cons(label("add"),adt("EXP"),
      [label("left",adt("EXP")),label("right",adt("EXP"))]),...)))
```

Type literals allow the implementation of DCFlow to work *generically* over different AST definitions for different programming languages. The implementation of DCFlow uses the type information for the AST being processed to generate correct code for CFG construction, while the IDE support uses this same type information to detect errors in the DCFlow definition.

2.4 String Templates

Rascal provides string templates for code generation, a frequently occurring operation in meta-programming. String templates are multi-line string literals with a left-margin (given with a single quote character), interpolation of arbitrary expressions, auto-indentation, and structured control flow. An example from

```
res = "public tuple[<p.astType>,LabelState] labelAST(LabelState ls, <p.astType> ast) {
      '    Lab incLabel() {
      '        ls.counter += 1;
      '        return lab(ls.counter);
      '    }
      '    labeledAst = bottom-up visit(ast) {
      '        <for (n <- gs.annotatedTypeNames) {> case <n> n => n[@lab = incLabel()]
      '        <}>
      '    };
      '    ls.cfgNodes = ( n@lab : cfgNode(n,n@lab) | /node n := labeledAst, (n@lab)?);
      '    return < labeledAst, ls >;
      '}";
```

```
public tuple[PROGRAM,LabelState] labelAST(LabelState ls, PROGRAM ast) {
    Lab incLabel() {
        ls.counter += 1;
        return lab(ls.counter);
    }
    labeledAst = bottom-up visit(ast) {
         case PROGRAM n => n[@lab = incLabel()]
         case STATEMENT n => n[@lab = incLabel()]
         case EXP n => n[@lab = incLabel()]

    };
    ls.cfgNodes = ( n@lab : cfgNode(n,n@lab) | /node n := labeledAst, (n@lab)?);
    return < labeledAst, ls >;
}
```

Fig. 2. String Templates in Rascal.

DCFlow is shown in Figure 2. The top of Figure 2 shows a string template from the `GenerateLabeler` module. Rascal code given between angle brackets, such as `<p.astType>`, is evaluated, with the results inserted into the string at that position (string interpolation); an embedded `for` loop generates a `case` (used in the Rascal `visit` construct, which is used for structure-shy traversal) for each element `n` in the set `annotatedTypeNames`, which holds the names of the abstract syntax types that should be labeled, and are thus "linkable", in the control-flow graph. The code generated by this string template, specifically for labeling Pico ASTs, is shown at the bottom of Figure 2.

2.5 Custom Eclipse IDE Support

Rascal provides built-in support for creating Eclipse-based IDEs for languages defined in Rascal. Features supported include configurable syntax highlighting, foldable code sections, user-defined code outlines displayed using a standard Eclipse outline view, user-defined annotators that can register messages that display in the IDE and the Eclipse problem view (e.g., for reporting errors), automatic checking (invoking user-provided Rascal functions) of code during editing, and the addition of menu items to trigger user-provided functions. A number of these features have been used to create an IDE for DCFlow, with error checking to ensure that common mistakes (such as misspelling a field name) are visible in the IDE even before code generation occurs.

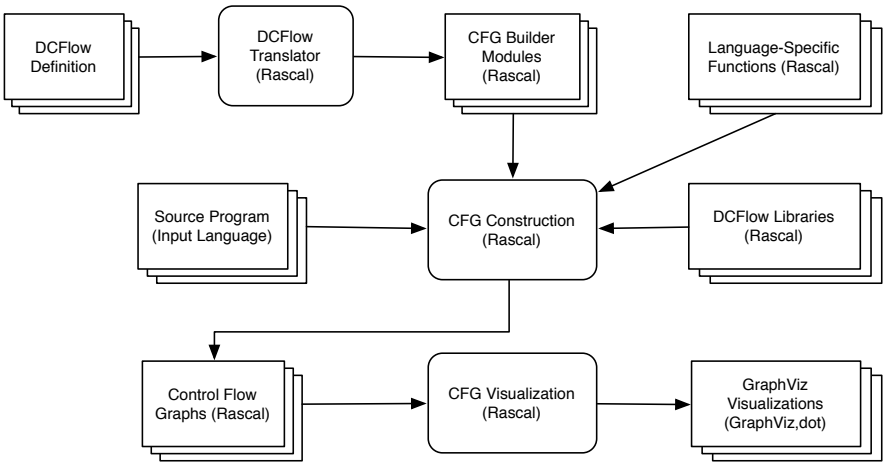

Fig. 3. DCFlow Architecture.

3 DCFlow

DCFlow is a declarative, domain-specific language and supporting libraries for defining the control flow rules for a programming language (referred to below as the *input language*). The architecture of DCFlow is shown in Figure 3. Once a DCFlow specification is created, the DCFlow translator converts the specification into a collection of Rascal modules. These modules handle the labeling of the AST, which assigns unique IDs to each instruction, and the creation of a control flow graph for an input program, based on the DCFlow rules specifying the control flow for the input language. The CFG construction process uses the generated modules, language-specific functions provided by the user (discussed more below), and DCFlow libraries to actually perform the control flow graph generation, giving one or more control flow graphs for an input program. These graphs can be used in analysis algorithms (see Section 4 for an example), and can also be visualized using DCFlow visualization functionality, which generates GraphViz diagrams using the `dot` language. Examples of these diagrams can be seen in Figures 6 and 7.

```
begin
    declare x : natural,
            y : natural;
    x := 3;
    if x then
        y := 10
    else
        y := 15
    fi
end
```

Fig. 4. Sample Pico Program.

The rest of this section describes the DCFlow language in detail. First we discuss control flow graphs and their representation in the Rascal DCFlow

```
map[loc, CFG]: (|pico+program://CFG/src/programs/pico/condition.pico|:cfg(
    |pico+program://CFG/src/programs/pico/condition.pico|,
    (
      lab(4):cfgNode( natCon(10), lab(4)),
      lab(5):cfgNode( asgStat( "y", natCon(10) ), lab(5)),
      ...
    ),
    {
      flowEdge( lab(5), lab(11), {}), flowEdge( lab(4), lab(5), {}),
      flowEdge( lab(7), lab(11), {}), flowEdge( lab(1), lab(2), {}),
      flowEdge( lab(2), lab(3), {}), flowEdge( lab(10), lab(1), {}),
      flowEdge( lab(3), lab(4), {conditionTrue()}),
      flowEdge( lab(3), lab(6), {conditionFalse()}),
      flowEdge( lab(6), lab(7), {})
    },
    ( ":exit":exitNode(lab(11)), ":entry":entryNode(lab(10)) )))
```

Fig. 5. DCFlow CFG Representation, in Rascal.

libraries. We then describe the DCFlow language, illustrating features of the language with a number of example control flow definitions from Pico and PHP. We end with a brief discussion of some additional features in DCFlow, as well as of what is currently not supported.

3.1 DCFlow Control Flow Graphs

Figure 4 shows an example of a simple program in Pico. After setting `x` to `3`, a conditional checks the value of `x`. The true branch, which sets `y` to `10`, is taken when `x` is not `0`, while the false branch, which sets `y` to `15`, is taken when `x` is `0`. The control flow graph for this program, extracted using a DCFlow CFG builder and given as a Rascal term, is then shown in Figure 5. A CFG is a directed graph, with nodes representing instructions or synthesized information (e.g., a synthesized exit node for a function) and directed edges showing how control flows between the nodes.

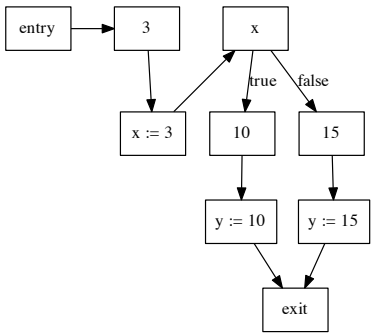

Fig. 6. CFG for Program in Figure 4.

Rascal provides built-in support for source location literals (values of type **loc**) that are Uniform Resource Identifiers[4] (URIs) optionally followed by text coordinates that allow the identification of specific text ranges in the information

[4] See `http://www.ietf.org/rfc/rfc3986.txt`.

the URI points to. Location literals are quoted with bars, such as `|http://www.rascal-mpl.org|`. Since, in many languages, a program can yield multiple control flow graphs (e.g., in PHP each function will have its own graph), DCFlow returns a map from source locations to control flow graphs. The location points to the location in the source code associated with the graph, for instance, to the function represented by the graph, and is created by a user-defined function specific to the input language. We intentionally use locations like those used in M3 [10], a model for source code artifacts. The ADT defining the control flow graph contains the location, a map from unique node labels to the actual control flow graph nodes (some of which are elided here), a set of directed flow edges (given with node labels as the from and to endpoints), and finally a map from special labels to specific nodes, in this case marking the designated entry and exit nodes for the program.

A visualization of this CFG is shown in Figures 6 and 7. The graphs clearly encode the order of evaluation: starting at the `entry` to the program, first `3` is evaluated, then the assignment to `x` is performed. After this, `x` is evaluated, with control then following either the `true` branch or the `false` branch. Along the `true` branch `10` is evaluated, followed by the assignment to `y`; along the `false` branch `15` is evaluated, again followed by the assignment to `y`. Both branches rejoin at the unique `exit` node, which represents the end of the program. Figure 6 shows a CFG with each node in its own block, while Figure 7 shows a CFG where blocks have been merged into *basic blocks* using the DCFlow `BasicBlocks` library module. A basic block is a sequence of instructions where control has to enter with the first instruction and must leave only at the end (e.g., an instruction in the middle of the block cannot transfer control to anything other than the next instruction). We still show the order of evaluation in each block, so (for instance) we still see that `15` is evaluated before the assignment `y := 15`—this is more verbose, but makes the evaluation order explicit.

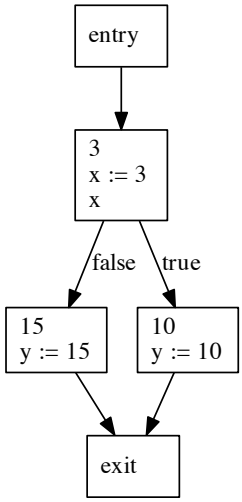

Fig. 7. CFG with Basic Blocks.

3.2 DCFlow Definitions and Sequential Control Flow

Figure 8 shows the DCFlow definition for the straight-line part of the Pico language (i.e., the entire language except for `if` and `while` statements). Since DCFlow is designed to be used with Rascal, DCFlow modules have a similar structure to Rascal modules. A DCFlow module is named using `module` (in this case, `Pico`); the name used here is then also used to name the generated files. While a number of DCFlow modules are automatically added as imports

```
module Pico

ast demo::lang::Pico::Abstract;
import lang::pico::CFGBase;
context PROGRAM::program;
astType PROGRAM;

rule PROGRAM::program = entry(exit(stats));
rule EXP::add = entry(left) --> right --> exit(self);
rule EXP::sub = entry(left) --> right --> exit(self);
rule EXP::conc = entry(left) --> right --> exit(self);
rule EXP::id = entry(exit(self));
rule EXP::strCon = entry(exit(self));
rule EXP::natCon = entry(exit(self));
rule STATEMENT::asgStat = entry(exp) --> exit(self);
```

Fig. 8. The DCFlow Definition for Straight-Line Pico.

in the generated code, additional modules can be added using the `ast` and `import` commands. Additionally, one module must be imported using `ast`, which indicates where the AST types, used extensively in DCFlow, are declared. Since a program could result in multiple control flow graphs, `context` indicates the constructors for which control flow graphs should be created. These may be nested: in PHP, control flow graphs are created for the script, representing the entire file, and for each individual function and method contained inside. Finally, `astType` actually names the "top" type of the AST, generally the type representing an individual program or compilation unit; the assumption is that there is one unique type. DCFlow loads the reified representation of this type, which also includes all types on which this depends, during code generation.

Following this initial information, `rule` is used to define the control flow rules for individual language constructs, based on the abstract syntax for the language. The general structure of a rule is:

```
rule typename::consname = flow;
```

where typename is the name of the type, as given in the `data` declaration; `consname` is the name of the constructor; and `flow` describes the control flow for the construct, given using DCFlow operations, names of special CFG nodes, and the field names of the constructor. Looking at Figure 8, the `entry` and `exit` operations indicate where flow enters the construct and where it exits the construct, while the special name `self` stands for the construct as a whole.[5] Each rule triggers the generation of three functions: `entry`, `exit`, and `internalFlow`. `entry` returns the label of the first instruction that is executed as part of the

[5] If a field in the constructor is also named self, or any other DCFlow keyword, it can be used by prefixing it with a backslash, i.e., as `\self`.

construct, while `exit` returns a set of possible final instructions for the construct. `internalFlow` builds edges to represent the flow of control inside the construct.

For instance, looking at the rule for the `id` constructor of `EXP`, the flow is given as `entry(exit(self))`. `entry` and `exit` mark where control flow enters and exits the construct—nesting one inside the other indicates that both mark the same construct. Since this is given as `self`, the generated `entry` and `exit` functions will return the label (for `exit`, a set containing just the label) for the `id` expression. Since the rule does not reference any fields, the construct has no internal control flow. Thus, the generated `internalFlow` function adds no edges.

A more complex case is that for the `add` constructor of `EXP`. Here, the control flow is given as `entry(left) --> right --> exit(self)`. The arrows signify the flow of control between the named items. Here, this means that control enters at the left operand, flows into the right operand, and then finally to self, modeling the evaluation of the left operand, followed by evaluation of the right, and then finishing with evaluation of the addition expression as a whole. Since the left operand is, itself, an expression (a fact determined by the DCFlow generator by consulting the reified representation of the `EXP` type), the generated code will determine the entry label for an occurrence of `add` by recursing on the first operand and finding its entry label, which could lead to additional recursive calls. For instance, to find the entry label for `(a+b)+c` one would find the entry label for `a+b`, which is the entry label for `a`, which (as stated above) is the same as the label for `a` itself. Since the exit label is determined by checking `self`, no recursion takes place—the entry label for `self` is always that of the item as a whole. The internal flow function generated for `add` links the exit labels of `left` to the entry label of `right`, and the exit labels of `right` to the entry label for `self`. This "wires up" the expressions representing `left`, `right`, and `left + right`, ensuring the flow in the CFG mirrors that in an executing program.

A final example is the first rule in Figure 8. The control flow for a program is based on field `stats`, which represents the list of statements making up the program. Since this is a list, DCFlow will generate code to compute the internal flow for each statement in the list and to link the `exit` and `entry` labels of the statements together in sequence. DCFlow also contains a `foreach` operations

```
module Pico

ast demo::lang::Pico::Abstract;
import lang::pico::CFGBase;
context PROGRAM::program;

rule PROGRAM::program = ^$stats;
rule EXP::add EXP::sub EXP::conc = ^left --> right --> $self;
rule EXP::id EXP::strCon EXP::natCon = ^$self;
rule STATEMENT::asgStat = ^exp --> $self;
```

Fig. 9. The DCFlow Definition for Straight-Line Pico, Condensed.

```
tuple[FlowEdges,LabelState] internalFlow(EXP item:add(EXP left,EXP right), LabelState ls) {
  FlowEdges edges = { };
  < edges, ls > = addEdges(edges, ls, left);
  < edges, ls > = addEdges(edges, ls, right);
  for(exlab <- exit(left,ls)) {
    < edges, ls > = linkItemsLabelLabel(edges, ls, exlab, entry(right,ls) );
  }
  for(exlab <- exit(right,ls)) {
    < edges, ls > = linkItemsLabelLabel(edges, ls, exlab, item@lab );
  }
  return < edges, ls >;
}
```

Fig. 10. Generated Rascal Code, Control Flow for Addition in Pico.

that can be used to iterative over lists, allowing this to be done manually, but this is a common enough occurrence that the typical behavior is the default.

This definition can be condensed using several shorthands, as shown in Figure 9. First, if the type of the AST is not provided explicitly using `astType`, DCFlow assumes it is the type of the first context list item. Second, constructs with the same control flow can be defined in the same rule, with whitespace separating the names. Third, entry and exit can be replaced with `^` and `$`, respectively—the operators are intentionally the same as those used to match the start and end of a string in regular expression syntax. While we may take advantage of more defaults in the future, we currently prefer having more explicit information in DCFlow specifications, since this creates less "magic" that the user is then required to understand, making definitions less cryptic.

To get an idea of the code generated by DCFlow, Figure 10 shows the code generated to handle the internal flow of the Pico addition expression. The input to the function is the addition expression and the label state, which, at runtime, tracks information needed to properly label the AST and build the control flow graph. An empty set to hold the generated edges is created, then `addEdges` is called twice, first on the left operand, then on the right. After this the two are linked, with all exits from the left operand (in languages with constructs such as the ternary conditional expression, there could be multiple exits) linked to the entry to the right. This same operation is then performed again, in this case linking the right operand to the add expression itself. Finally, this set of generated edges, along with the current state, are returned.

3.3 Defining Basic Decisions and Loops

The definition of the `if` and `while` statements in Pico is shown in Figure 11. Both are defined using the same building blocks shown above, with some minor additions. First, it is possible for a rule to have multiple, distinct operations, separated by commas. The first rule shown, for `if`, has three, while the second has two. Second, one or more labels can be given on an arrow by writing them inside the arrow body (after at least one dash, and also followed by at least one dash). So, the rule for `if` states that the condition (`exp`) is the entry, and that

```
rule STATEMENT::ifElseStat = ^exp,
                             exp -conditionTrue-> exit(thenpart,exp),
                             exp -conditionFalse-> exit(elsepart,exp);
rule STATEMENT::whileStat =  ^$exp -conditionTrue-> body -backedge-> exp,
                             exp -conditionFalse-> create(footer);
```

Fig. 11. Pico Decisions and Loops Modeled in DCFlow.

there are then two edges, one from `exp` to the then branch when the condition is true, and one from `exp` to the else branch when the condition is false. Third, `exit` can appear multiple times, marking multiple possible exits from the construct. Finally, `entry` and `exit` can contain a list of names instead of just a single name. In this case, the names will be tried, in order, during CFG construction, stopping when a usable label or set of labels, respectively, is found. This handles the situation where `thenpart` or `elsepart` may be empty, in which case the final instruction evaluated on that path would actually be the condition `exp`.

The `while` statement has a similar definition: the condition is tried and, if true, the body is executed. Here, we explicitly mark the edge from the body back to the condition as a loop backedge. When the condition is false, we instead need to exit the construct. We could link to the following instruction using the keyword `following`, but instead create a new footer node for the entire loop, linking to that instead. This will cause all exits from the loop (here, through `exp`, which is marked as the exit) to pass through this footer node, and will cause the footer node to be used as the `exit` when linking this to any statements following this in the program. Finally, note that, once a name has been marked as an entry or exit point, other uses of the name do not need to be so marked again.

3.4 Defining Unstructured and Structured Jumps

DCFlow distinguishes between *unstructured* and *structured* jumps. Unstructured jumps, such as `goto` statements, essentially ignore other control flow constructs, transferring control to an arbitrary instruction. In PHP, a `goto` will jump to a label defined on a statement, and cannot transfer control out of the current context (e.g., from inside a function back to the top-level script) or into a loop or switch.[6] Structured jumps, such as `break` and `continue`, work in tandem with language constructs such as `while`, `for`, and `switch` statements, with the target of the jump depending on the semantics of the associated statement. In PHP, a `continue`[7] in a `while` loop will jump back to the loop condition, while a `break`[8] will instead transfer control to the first instruction after the loop. To work with nested control constructs, both `break` and `continue` accept an optional numeric argument—if given inside a loop nested inside another loop, `break 2` would jump

[6] `http://www.php.net/manual/en/control-structures.goto.php`
[7] `http://www.php.net/manual/en/control-structures.continue.php`
[8] `http://www.php.net/manual/en/control-structures.break.php`

```
rule Stmt::goto      = jump(\label),^$self;

rule Stmt::\while    = create(footer), jumpTarget(cond,\continue),
                       jumpTarget(footer,\break),
                       ^$cond -conditionTrue-> body -backedge-> cond,
                       cond -conditionFalse-> footer;
rule Stmt::\break    = entry(breakExpr,self) --> $self,
                       jump(breakExpr,\break);
rule Stmt::\continue = entry(continueExpr,$self) --> self,
                       jump(continueExpr,\continue);
```

Fig. 12. PHP Jumps Modeled in DCFlow.

to the instruction following the outer loop. Other languages, such as Java and Rascal, provide similar functionality by instead allowing loops to be labeled, similarly to how statements are labeled for `goto` in PHP.

The DCFlow definitions of `goto`, `while`, `break`, and `continue` for PHP are shown in Figure 12. These rules introduce several new DCFlow constructs, and also assume that several Rascal functions have been defined. To support unstructured jumps, calls to user-provided function `findUnstructuredJumpTargets` are generated; this function identifies all unstructured jump targets—for PHP, statement labels—in the current context. The `jump` construct then specifies a jump in the control flow to a destination identified by the operand—here, the `label` field of the `goto` statement. This is looked up using user-provided function `getTargetsForJump` and, for unstructured jumps, must be one discovered by `findUnstructuredJumpTargets`. The code for `jump` will then create flow edges from the exit labels of the instruction to these target labels.

The definition for `while` shows how structured jump targets are defined. An explicit footer is created for the loop first. Two jump targets are then registered with the `jumpTarget` operation—a target for `continue`, which will jump back to the condition, and a target for `break`, which will jump to the loop footer.[9] DCFlow generates calls to user-provided function `createJumpTarget` to actually perform this registration. The definition of the loop itself is then very similar to that given for Pico in Figure 11. The structured jumps to these targets then occur in the `break` and `continue` statements, which both have very similar definitions. In both cases the entry to the construct is the optional argument, with the construct itself serving as the default if this argument is empty. Flow then goes to the actual `break` or `continue` statement. The jump is again specified with the `jump` operation; the first argument gives information on the target, while the second identifies the type of jump target, which must match the type given in the `jumpTarget` command. This will result in flow edges from the exit labels of the instruction (here, just one) to the entry label of the target instruction—for while, either to the condition (for `continue`) or to the added footer (for `break`).

[9] Targets `break` and `continue` are available by default. DCFlow operations also allow defining new types of targets.

3.5 Other Features and Limitations

There are several other features of DCFlow that support less common cases, including list operations such as `first`, `next`, and `last`; an `is` operation to check to see if a field is constructed using a specific constructor; and `foreach` and `if` operations that can be used to describe more complex control flow.

There are also some control constructs DCFlow cannot currently support, the most common being exceptions. In the PHP definition, we instead define support for exceptions directly in Rascal, indicating in the DCFlow definition that the code generator should ignore the `throw`, `try/catch`, and `try/catch/finally` statements. While it would be useful to expand DCFlow to support such features, it may be quite challenging to define them generically—error handling features of languages can differ in fairly significant and sometimes subtle ways. Given this, it may be the case that using such generic features to define the control flow in DCFlow would take roughly the same amount of effort as defining the control flow directly in Rascal, in which case this would provide little benefit (as discussed in Section 4, the amount of code to handle these features for PHP is a fraction of the total code, most of which can now be generated directly from a DCFlow definition) while risking an increase in conceptual complexity.

4 Evaluation

As stated in Section 1, the purpose of DCFlow is to streamline the process of defining the control flow rules for programming languages, with the goal of generating Rascal code that can extract control flow graphs from programs in that language. In this section, we evaluate the effectiveness of DCFlow using three techniques. First, we compare DCFlow definitions to definitions given directly in Rascal. Second, we compare DCFlow definitions to definitions given using DeFacto [3], a fact extraction framework developed for Asf+Sdf [26,25] and RScript [14], a precursor to Rascal. Finally, we illustrate use of DCFlow-generated control flow graphs in a standard data flow analysis for Pico programs.

4.1 Comparison with Rascal Definitions

Since the main motivation for creating DCFlow was to simplify the process of creating control flow graphs and graph extractors in Rascal, we first compare the results of using DCFlow with custom Rascal solutions. The control flow for Pico, discussed first, has been completely defined, while the control flow for PHP, discussed second, is complete except for the definitions for a handful of features implemented directly in Rascal.

Pico: Module `demo::lang::Pico::ControlFlow`, part of the standard Rascal library, contains the definition for the control flow graph for Pico as well as all code to extract this graph from Pico ASTs. In total, this consists of 45 lines of code: 11 giving the module header, imports, and definitions of control flow

nodes and graphs, and 34 defining the rules used to extract the control flow. The DCFlow definition for Pico is 10 lines of code: 4 header lines and 6 rules. The DCFlow generator converts this into 408 lines of Rascal code—it is much larger than the custom Rascal solution because the generator is language generic, so it cannot take advantage of the simplicity of the Pico control flow rules.

PHP: As mentioned in Section 1, the PHP AiR definition of PHP control flow is 1,583 lines of Rascal. The DCFlow definition is currently 66 rules (some handling multiple constructs) and 6 header lines, generating 2,714 lines of Rascal. User-provided functions to compute jump targets add another 57 lines of code, while code used to handle features such as exceptions is another 169 lines.

4.2 Comparison with DeFacto

In DeFacto, fact extraction is performed using *fact annotations*, *annotation functions*, and *selection annotations*. Fact annotations are added to the production rules of a grammar, and state a named fact that can be computed for the given language construct. For instance, a production that defines a new identifier as having a certain type can be annotated with a `typeOf` fact stating that this identifier has the defined type. The fact is represented using a relation, with a single instance of the fact represented as a tuple in the relation. Annotation functions and selection annotations are then used to deal with lists and optional elements of productions, allowing list iteration (e.g., to get the first or last element of a list, or to get pairs of elements representing the next relation) and selection based on the presence or absence of list elements or optional subterms. DeFacto annotations can be given in separate modules which are "woven" in as needed, allowing different facts to be extracted based on the needs of the analysis. Non-local facts can then be computed using RScript, which allows relational algebra operations to be performed over these relations.

DeFacto and DCFlow share many similarities: both work by defining rules over language constructs, and both include support for handling commonly occurring constructs such as lists and optional data. There are also a number of differences between the two approaches. DCFlow is designed specifically to specify control flow rules, versus more general program facts, so it supports more specialized notation (e.g., the name decorations used in Figure 9, arrows to represent edges) and can make more default assumptions about how common constructs (e.g., a list representing the body of a block) are handled. DCFlow also works at the level of the abstract syntax, instead of concrete syntax, allowing it to be used in cases where a Rascal parser definition is not available (but also requiring an abstract syntax to be defined even if it is not otherwise needed). The underlying language is also different: RScript can be seen as a subset of Rascal, specifically focused on relational operations and fixpoint computation, but lacking the broader support for string manipulation, code generation, IDE creation, and visualization that is used in DCFlow.

Specifically focusing on Pico, the DeFacto and RScript control-flow graph extraction consists of 11 fact annotations over 3 relations and one relational

expression, giving a total of 12 statements and 13 lines of code. In DCFlow, a rule is defined for each AST constructor used to define the program, expressions, and statements, 10 in total, although these are collapsed 6 distinct rules since several have identical control flow. The entire module has a total of 10 lines of code, the 6 rules, the module name, two imports (one for the AST type, one for a language-specific function used to create a Rascal location representing Pico programs), and the definition of the context, specifying the scope of the control flow graph (here, the entire program). The module containing the language-specific function is a total of 4 lines of code: the module header, two imports, and a one-line function definition. The comparison for Pico thus shows a very similar level of effort using both DeFacto and DCFlow. DeFacto is no longer maintained, so it is hard to determine if this would hold with larger languages and/or languages with more complex control flow, such as PHP.

4.3 Reaching Definitions with DCFlow CFGs

As part of our evaluation, we have defined a standard reaching definitions analysis for Pico using the CFG created by DCFlow. An alternate version [15], working directly over relations of control flow facts, is in the Rascal standard library in module `demo::ReachingDefs`. Figure 13 shows an example Pico program, with the instruction labels shown at the end of several lines (e.g., the first assignment to `x` is labeled 2). The implementation of the reaching definitions algorithm is then shown in Figure 14. Function `computeDefs` computes a relation over the entire program, from Pico identifiers to the labels where these identifiers are defined (here, using assignment statements). `gen` computes the set of all labels corresponding to definitions introduced by the instruction—assignments introduce the label of the assignment statement, while all other instructions introduce the empty set (indicated with `default`, meaning this function handles all other cases). `kill` also treats assignment as a special case—a new assignment into a name will remove all defs of that name except for the current one. Function `computeReach` then uses these to compute the `in` and `out` sets for each instruction, returned as relations from instruction labels to definition labels. `in` will contain all definitions that may reach the start of the labeled instruction, while `out` contains all definitions that reach the end. Starting with empty relations, and the definitions for the program given by `computeDefs`, a fixpoint computation (indicated using `solve`) iteratively computes the `in` and `out` sets for each label. The `in` set is the result of the `out` sets for all predecessors (computed with `pred`,

```
begin
declare x : natural,
        y : natural;
x := 1; // 2
y := 2; // 4
if x then
  y := y + 1 // 9
else
  while y do
    x := x + 1; // 14
    y := y - x // 18
  od
fi;
y := x // 22
end
```

Fig. 13. Reaching Definitions Example, in Pico.

```
rel[PicoId,Lab] computeDefs(CFG c) =
  { < name, l > | cfgNode(asgStat(PicoId name, _),l) <- c.nodes<1> };

set[Lab] gen(cfgNode(asgStat(PicoId name, _),l)) = { l };
default set[Lab] gen(CFGNode n) = { };

set[Lab] kill(cfgNode(asgStat(PicoId name, _),l),rel[PicoId,Lab] defs) = defs[name]-l;
default set[Lab] kill(CFGNode n, rel[PicoId,Lab] defs) =  { };

tuple[rel[Lab,Lab] reachIn, rel[Lab,Lab] reachOut] computeReach(CFG c) {
  rel[Lab,Lab] reachIn = { };
  rel[Lab,Lab] reachOut = { };
  defs = computeDefs(c);
  solve(reachIn,reachOut) {
    reachIn  = { < l, r > | l <- c.nodes, r <- reachOut[pred(c,l)] };
    reachOut = { < l, r > | l <- c.nodes,
                            r <- (gen(c.nodes[l]) + (reachIn[l] - kill(c.nodes[l],defs))) };
  }
  return < reachIn, reachOut >;
}
```

Fig. 14. Reaching Definitions Algorithm, in Rascal.

a DCFlow library function), while the `out` set is the result of the `in` set, minus anything killed by the current instruction, plus anything generated—basically, any definitions that come in to the instruction that are not killed by it, plus any definitions the instruction generates itself. When the fixpoint completes the relations are returned.

Looking at several points of interest in Figure 13, running the algorithm shows that no definitions reach instruction 2, since at the start of the instruction no definitions have occurred yet; the definition at 2 reaches 4; and the definition of `x` at 2 reaches 14, but not 18, since 14 redefines `x` and is always run before 18. The definition of `y` at instruction 18 can also reach itself, since an assignment to `y` made in one iteration of the loop will reach the next iteration. Any of the definitions before 22 can reach 22, since control may have flowed through either the true or false branch of the conditional.

5 Related Work

In this section we look at two areas of related work. First, we look at general fact extraction techniques, such as DeFacto. Second, we look specifically at recent research on specifying control flow declaratively, and on using domain-specific languages to specify program properties which can be used in analysis.

Fact Extraction: Basic fact extraction can be performed using tools such as Lex [17] and languages such as AWK [1], Perl, or Python, using regular expressions to match patterns in the code and then record the associated facts. These approaches are language specific—different patterns would be needed for each language—and cannot naturally handle the nested constructs common in programming languages. Murphy and Notkin [20,21] have extended this approach

to include additional contextual information, allowing regular expressions to be given in a hierarchy where some expressions only match after others have already matched (e.g., an expression matching a function call may match only after an expression matching a function definition has already matched). Extracted facts can also be organized in relations, allowing additional facts to be computed after scanning is complete.

Approaches based on grammars can more naturally handle the nested constructs common to programming languages, but also generally require the source code to be syntactically correct. The most basic example of a grammar-based extractor would be one that used the semantic actions in Yacc [11] or other parsing systems to record and compute facts. More complex tools include the Rigi system [19], which provides fixed fact extractors for several languages, representing extracted facts as tuples in a format named RSF (Rigi Standard Format), and systems that use attribute grammars [12,22,6,23,29], which use synthesized attributes to specify facts and inherited attributes to propagate these through the parse tree.

Other approaches have focused on using queries to build relations, with relational operations then used to combine facts and perform the analysis. Rigi, mentioned above, uses tuples given in the RSF format and a language, the Rigi Command Library (RCL), to manipulate these tuples. GROK [9] and CrocoPat [4,5] (using a notation called RML) instead use relational algebra, with GROK supporting binary relations and CrocoPat supporting n-ary relations. The DeFacto system [3], discussed in Section 4, uses RScript [14], which also supports n-ary relations and relational algebra, as a query language for extracted facts, as does Vankov's work on formulating program slicing using relational techniques [27]. Rascal [15,16] has n-ary relations as a native datatype, while relational operations, such as transitive closure, are built in to the language.

DSLs and Declarative Control Flow: Other than DeFacto, the most closely related work to DCFlow uses JastAdd [6] to declaratively define control flow rules and dataflow analysis algorithms based on abstract syntax trees [24]. Reference attributes are used to represent the control flow edges in the AST; collection attributes allow the specification of inverse relations (such as the predecessor relation, given an existing successor relation between control flow nodes); and higher-order attributes allow the synthesis of new AST nodes, such as standard entry and exit nodes for methods. In contrast, DCFlow focuses just on the declarative specification of control flow rules, and uses Rascal functionality, instead of attribute grammars, to create the control flow graph. For instance, computing pred can either be performed by inverting the flow relationship, given as a graph, or by pattern matching over the control flow edges.

DCFlow is also similar, conceptually, to other work on using focused domain-specific languages to support program analysis tasks. This includes the DHAL language [18] and its variants, for data flow analysis, and an approach for performing incremental name and type analysis [28], implemented as part of the Spoofax language workbench [13], which includes a task language with a number

of instructions related to name and type analysis (e.g., to lookup or cast a type) and a number of combinators to combine the results of subtasks.

6 Discussion and Future Work

In this paper we presented DCFlow, a domain-specific language for declaratively specifying the control flow rules for a programming language based on its abstract syntax. DCFlow can specify the control flow for a large number of typical language constructs, generating the Rascal source code needed to construct control flow graphs for programs using these features. As shown in Section 4, these specifications are much shorter than a custom Rascal solution, especially for larger languages. For features that are not currently supported, such as exceptions, Rascal code can be written directly, extending the code generated by DCFlow and taking advantage of DCFlow library modules.

In the future, we plan to continue development of DCFlow, extending it to handle features that are not currently supported in cases where a general form, reusable across multiple languages, can be defined without adding too much additional complexity. We also want to improve the visualization support provided by the DCFlow library, allowing control flow graphs to be visualized directly in Rascal as well as using GraphViz. Finally, we would like to explore enabling DCFlow to be used as part of the Rascal resources framework [8], allowing code generation and import of the CFG builder to be triggered by importing DCFlow specifications into Rascal modules.

References

1. A. Aho, B. Kernighan, and P. Weinberger. Awk - A Pattern Scanning and Processing Language. *Software–Practice and Experience*, 9(4):267–280, 79.
2. F. E. Allen. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.
3. H. J. S. Basten and P. Klint. DeFacto: Language-Parametric Fact Extraction from Source Code. In *Proceedings of SLE'08*, volume 5452 of *LNCS*, pages 265–284. Springer, 2008.
4. D. Beyer, A. Noack, and C. Lewerentz. Simple and efficient relational querying of software structures. In *Proceedings of the 10th Working Conference on Reverse Engineering*, pages 216–225, 2003.
5. D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering*, 31(2):137+, 2005.
6. T. Ekman and G. Hedin. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69(1–3):14–26, 2007.
7. M. Hills and P. Klint. PHP AiR: Analyzing PHP Systems with Rascal. In *Proceedings of CSMR-WCRE 2014*, pages 454–457. IEEE, 2014.
8. M. Hills, P. Klint, and J. J. Vinju. Meta-language Support for Type-Safe Access to External Resources. In *Proceedings of SLE'12*, volume 7745 of *LNCS*, pages 372–391. Springer, 2012.
9. R. Holt. Binary Relational Algebra Applied to Software Architecture. CSRI 345, University of Toronto, March 1996.

10. A. Izmaylova, P. Klint, A. Shahi, and J. J. Vinju. M3: An Open Model For Measuring Code Artifacts. Technical Report arXiv-1312.1188, CWI, December 2013.
11. S. C. Johnson. Yacc: Yet Another Compiler-Compiler. Technical Report CS TR 32, Bell Labs, 1975.
12. M. Jourdan, D. Parigot, C. Julié, O. Durin, and C. L. Bellec. Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System. In *Proceedings of PLDI'90*, pages 209–222, 1990.
13. L. C. L. Kats and E. Visser. The Spoofax Language Workbench. In *OOPSLA 2010 Companion*, pages 237–238. ACM, 2010.
14. P. Klint. Using Rscript for Software Analysis. In *Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008)*, 2008.
15. P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In *Post-Proceedings of GTTSE'09*, volume 6491 of *LNCS*, pages 222–289. Springer, 2011.
16. P. Klint, T. van der Storm, and J. J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of SCAM'09*, pages 168–177. IEEE, 2009.
17. M. Lesk. Lex - a lexical analyzer generator. Technical Report CS TR 39, Bell Labs, 1975.
18. L. Moonen. Data Flow Analysis for Reverse Engineering. Master's thesis, University of Amsterdam, 1996.
19. H. Müller and K. Klashinsky. Rigi – a system for programming-in-the-large. In *Proceedings of ICSE'88*, pages 80–86, April 1988.
20. G. Murphy and D. Notkin. Lightweight source model extraction. In *Proceedings of FSE'95*, pages 116–127, New York, NY, USA, 1995. ACM Press.
21. G. C. Murphy and D. Notkin. Lightweight Lexical Source Model Extraction. *ACM TOSEM*, 5(3):262–292, 1996.
22. J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.
23. A. M. Sloane. Lightweight Language Processing in Kiama. In *Post-Proceedings of GTTSE'09*, volume 6491 of *LNCS*, pages 408–425. Springer, 2011.
24. E. Söderberg, T. Ekman, G. Hedin, and E. Magnusson. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Science of Computer Programming*, 78(10):1809–1827, 2013.
25. M. van den Brand, M. Bruntink, G. Economopoulos, H. de Jong, P. Klint, T. Kooiker, T. van der Storm, and J. Vinju. Using The Meta-environment for Maintenance and Renovation. In *Proceedings of CSMR'07*, pages 331–332. IEEE, 2007.
26. M. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In *Proceedings of CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.
27. I. Vankov. Relational approach to program slicing. Master's thesis, University of Amsterdam, 2005.
28. G. Wachsmuth, G. D. P. Konat, V. A. Vergu, D. M. Groenewegen, and E. Visser. A Language Independent Task Engine for Incremental Name and Type Analysis. In *Proceedings of SLE'2013*, volume 8225 of *LNCS*, pages 260–280. Springer, 2013.
29. E. V. Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1-2):39–54, 2010.