STAGED STATIC ANALYSES AND RUN TIME PROGRAM GENERATION

BY

MICHAEL KATELMAN

B.S.E., University of Michigan, 2004

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Acknowledgements

This project was started in September 2004 when I first joined Professor Samuel Kamin and his group at the University of Illinois at Urbana-Champaign. Not only did Professor Kamin provide the initial idea, he has guided the project though many iterations over the course of a year and a half. At every step I have felt extremely edified working with him; he has been a superior advisor.

In addition, I am in considerable debt to my office mate and fellow student, Tankut Barış Aktemur. He has been involved with and substantially contributing to this project since around September of 2005, and has been a pleasure to work with throughout the entire time I have attended the University. His help, patience, and good sense are things that I have very much appreciated and benefited from. At the time of this writing, he has also spearheaded the continuation of this project, producing a more elegant framework and theoretical results.

Lastly, I would be remiss if I forgot to mention the financial support provided me by Professor Kamin for the school years 2004/2005 and 2005/2006, and in addition the summer of 2005. I am extremely grateful for the support and it is my sincere hope that it has been regarded as being well used. Lastly, I should say that any flaws in what follows are mine alone.

Table of Contents

Chapter 1

Introduction

Run time program generation (RTPG) is a programming paradigm that allows for the dynamic generation and use of code fragments. This means that part of the execution of a program is dedicated to modifying its own text section (or equivalent if the process environment is sand boxed) and adding new executable code, such as new classes or functions. The benefit of this is through the use of run time information to generate specialized and/or more efficient code. The downside is that the code generation process takes time during execution without performing any real work for the program. Part of the code generation process includes checking the well-formedness of the source code being generated: for example, that basic typing and syntax requirements are met. Code generation time can also include traditional compiler optimization passes for the purpose of emitting more efficient machine or byte code.

The problem that we address is the development of sound, generic methods to reduce the costs associated with run time code generation and thus to improve overall efficiency. Checking code well-formedness prior to generation is obligatory when using RTPG because otherwise you could generate junk code and cause erratic program behavior. Obviously that would be a situation that would be altogether undesirable and unacceptable. The optimization passes are entirely optional of course, but the smaller the cost the easier it is to justify applying that pass, which means that overall program execution should improve in general. A fundamental fact that we will exploit in this work is that large portions of the costs associated with code generation are either naturally formulated as or easily reduced to *data flow* problems. For example, variable initialization checks and some basic type checking can be formalized this

way and are used to validate code well-formedness. Reaching definitions is a classic example of a natural data flow analysis problem, and is required for a number of compiler optimizations.

This Thesis has two main chapters and some auxiliary and background material. Chapter 2 provides a basic introduction to RTPG and in particular introduces Jumbo, a RTPG compiler for Java. We use Jumbo as a test bed for the algorithms developed later on. Also included as background information is an overview of classical data flow analyses using control flow graphs (CFG). This is important because the CFG based data flow algorithms are so well ingrained in the minds of compiler researchers that they provide a necessary and convenient backdrop for explaining our own algorithms, which, as we have said, will rely on data flow formulations. Chapter 3 represents the main body of work contributed by this Thesis. It contains a general mathematical framework that can be used to compute forward data flow analysis information, and, in addition, contains a general algorithm for reducing the run time cost of applying analyses formalized in the framework. Our framework is distinguished from the CFG based one mainly by the fact that it works directly on program source and that the framework's formulation is purely functional, rather than algorithmic. Chapter 4 establishes the soundness of our framework with respect to the well known and long used CFG based one framework described in Section 2.2. Finally in Chapter 5 we end with some concluding remarks and observations.

Chapter 2

Background

2.1 Run Time Program Generation and Jumbo

In the introduction we briefly described what RTPG is and why it is studied and used. Here in this section we point to some of the RTPG systems that have been developed and describe some of the research that has been done relating to RTPG. We focus primarily on Jumbo [7, 10, 12] because it is the compiler we are most familiar with and the one we have built upon for the purposes of testing the ideas in this Thesis.

There are a number of RTPG systems, many of which use a quote/antiquote mechanism of specifying code fragments and code assembly sites. Quoted code is the basic object manipulated at run time and is the unit of code generation. In Jumbo for example, *quoted* code is anything enclosed in `$<...>$`, so that you write things like

$$\texttt{\$< while(x} > 0\texttt{) `Stmt(...) >\$}$$

The `‘Stmt` part is an *antiquote*, and Jumbo requires that it be replaced before code generation can take happen. The piece of code in this case is expected to be a statement, but in other cases may be an integer or some other type. We call the places where antiquotes exist *holes*; when another piece of code is fit into a hole it is called a *plug*. Code fragments without any holes are called *whole* and are candidates for program generation. In addition, before generation of a whole program can take place in Jumbo, there must be a check done to make sure that the code fragment represents a syntactically well-formed Java class. Other examples of RTPG systems

are MetaML [16] and ‘C [6, 8].

The mechanism through which Jumbo generates code is one of its distinguishing and novel features. Quoted code is a special object type recognized by the Jumbo compiler, called *Code*, and statically defined Code objects are processed at compile time into a set of function calls, replacing the syntactic program constructors in a nearly one-to-one way. An example from [14] describes that

```
Code safePointer (Code ptr, Code computation) {
   return $< if (Expr(ptr) == null)
      throw error();
   else Stmt(computation) >$;
}
```

translates into

```
Code safePointer (Code ptr, Code computation) {
   return makeStatements(
      makeIfThenElse(
         makeBinOp(0, ptr, nullConstant()),
   makeThrow(makeSelfInvocation("", "error", new List())),
               computation));
}
```

after preprocessing is performed by Jumbo (0 is Jumbo's code for the "==" operator). The functions that get generated from the translation *are* the code generators, and therefore invoking the compiler externally is avoided. Note, however, that the syntax is essentially still preserved. This is an important point because our algorithm is simplified by working with program source both at compile time and run time.

In a traditional compilation flow, the front end of the compiler would convert the program source into some low level intermediate form for the purposes of doing optimizations and eventually emitting code. Many optimizations would involve creating a CFG from the program's intermediate representation and running data flow analysis of varying types. In our case, the situation is somewhat different because we

apply the data flow analysis directly to the program source. Even if unable to take advantage of a good intermediate level representation, source level compilation is possible and can serve as a good platform for optimization in many cases, cf. [14, 5]. In the case of Jumbo, where code is emitted by what amounts to a traversal of the *abstract syntax tree* (AST), such efforts clearly have potential.

Previous research involving Jumbo has demonstrated methods to speed up programs [7, 12] and also generic methods for developing specialized components [3] which are more reusable and less error prone than would be possible without RTPG. Again in Jumbo, source level rewriting has been employed [14, 9] in optimizing the processed quoted code objects. Some optimizations that the rewriters are able to perform are things like constant propagation and loop unrolling. Another very interesting project involving Jumbo has translated the core functionality into a set of *rules* [15] which specify a formal semantics for the language. This semantic presentation allows for experimentation with new ideas and has led to a number of improvements in Jumbo.

2.2 Classical Data Flow Analysis

We explained in the introduction that our primary goal in this Thesis is to develop general methods for staging *data flow* analyses. The reasoning we gave for this goal is that data flow describes a class of analyses which naturally contains many of the calculations used to compile source programs. It therefore seemed to be a good place to begin attacking the problem of optimizing run time compilation.

Data flow problems have been studied extensively, with the great majority of work utilizing the *control flow graph* (CFG) representation of programs. This section reviews a well known control flow graph based framework for computing forward data flow analyses. We utilize this important framework throughout the Thesis: when we present our proposed framework in Chapter 3 we use it to help explain our design choices, and in Chapter 4 we prove the soundness of our proposed framework relative to it. Although we cannot even begin to survey the area in depth, we point the reader to the bibliographic notes in [1, Chapter 10] as reference.

In contrast to the CFG based framework laid out in this section, our data flow formalization needs to work directly on program source or some other similar representation (e.g. AST). The reason for this is that most RTPG compilers, including Jumbo, work exclusively at this higher level rather than going down to some intermediate representation. In practice this means that our analysis framework will be defined as a function over an inductive set representing well formed programs; which is rather beneficial because it allows for a direct and rigorous mathematical specification of the framework. Such a specification is essential for reasoning about the framework and in our case we will require the ability to formally prove properties of the framework useful for staging.

However, before we can set down the classical iterative algorithm we have to define the data types and functions that are used by and parameterize the framework. The first thing that needs to be described is the CFG structure itself, what it is, and how it is manipulated. Our description is informal, see [1] for a detailed treatment.

In the CFG representation of an executable program there are two main components: the graph's nodes represent *basic blocks* and the directed edges represent *control flow*. As usual, a basic block is taken to mean a single entry single exit code fragment. There is a distinguished block which dominates the graph and represents the unique starting point for the program.

Now, data flow analyses are characterized by program information being propagated along the graph's edges, along proper execution paths. This does not mean that data is always propagated in the *forward* direction of execution, some data flow analyses propagate information in the opposite direction. Common examples of forward propagating analyses include *reaching definitions* and *available expressions*, see [1]. Reaching definitions calculates the set of variable definitions that reach each node in the CFG. An example of a common *backward* propagating data flow analysis is *live variables*, which calculates for each basic block those program variables which have future uses along some execution path.

In Figure 2.2.1 we give a generic algorithm, or *framework*, for computing forward data flow analysis information; the framework is taken from [1]. The so-called *iterative* algorithm uses two auxiliary arrays (*in* and *out*) to keep track of data prop-

```
1: declare in/out arrays /* one array entry per basic block */
2: for all B, a basic block do
3:    out[B] := f_B[I]; /* initialize the out array */
4: end for
5: while out has been changed do
6:    for all B, a basic block, in breadth-first order do

7:       in[B] :=  ⋀_{B' a predecessor of B}  out[B'];

8:       /* above, meet of empty set is I */
9:       out[B] := f_B(in[B]);
10:   end for
11: end while
```

Figure 2.2.1: Iterative forward data flow analysis framework.

agation, with one entry in each array for every basic block. For each block B, the *in* array records the cumulative contribution for all paths reaching B, and the *out* array stores this value modified by the individual contribution of B. The value of $in[B]$ is calculated by applying the *confluence* operator ($\wedge$) to all of the predecessor nodes of B (line 7), and the value of $out[B]$ is this value modified by the transfer function of B, f_B (line 9).

The parameters used to make the framework generic are the data stored by the arrays and functions used to manipulate this data. From Figure 2.2.1 we can see that this includes the constant I, the transfer functions f_B, and the confluence operator $\wedge$. We already explained the transfer functions and the confluence operator. The constant I represents "no information" and thus is used for initialization. Of course, these parameters are different for each analysis.

As described in [1, Section 10.11], the framework in Figure 2.2.1 assumes a set of axioms constraining the data and functions parameterizing it. First, we assume to be a lattice, with $\wedge$ the meet operation. In addition, both *monotonicity* and *finiteness* are assumed. Monotonicity means that if we take any $d_1 \leq d_2$, ordered by

the data lattice, then $f_B(d_1) \leq f_B(d_2)$ holds for each transfer function f_B. Finiteness is a property that has to do with how cycles in the control flow graph generate and propagate information, it is described in [1, Section 10.11] where they also claim that most common analyses have this property.

In Chapter 4 we will make use of a result from [1, Section 10.11] that allows us to substitute the so-called *meet over paths* solution for the result calculated by the algorithm in Figure 2.2.1. This is calculated using the following basic ideas: take any *execution path* starting from the distinguished entry block described earlier, say

$$B_0, B_1, \ldots, B_n$$

and consider the function $f_{B_0} \circ f_{B_1} \circ \cdots \circ f_{B_n}$, called a *path function to* B_n. The *mop* solution for block B_n on control flow graph C, denoted $mop(n, C)$, is the meet of all path functions to B_n applied to the value I.

Finally, before moving on to our proposed framework and the next Chapter, we first need to slightly modify the iterative framework just presented to give us full control over the *traversal order*. In Chapter 4 when we prove the soundness of our framework we will use this modified version, which is equivalent to the original one in Figure 2.2.1 in terms of correctness. The traversal order refers to the order in which the main *update* inside the inner loop (lines 6-9) is applied to the basic blocks. The reason that the formulation from [1] uses a depth first ordering is that it has been shown to be a robust performance enhancement in that the *out* array converges more quickly. However, as long as the traversal order updates each block infinitely often, the algorithm will still converge and compute the correct result. Therefore, we can use either framework interchangeably when we are talking about correctness of the final result.Our modified version of the algorithm is given in Figure 2.2.2.

As a final side note, we again refer the reader to the bibliographic notes of [1] for a good set of references on data flow, in particular, references to work on *syntax directed* data flow analysis, such as [5]. We also note that our work presented in the following Chapter is quite different from all of the syntax directed data flow algorithms we have come across, mainly in that we work exclusively at the higher, syntactic, level.

```
1: declare in/out arrays /* one array entry per basic block */
2: for all B, a basic block do
3:    out[B] := f_B[I]; /* initialize the out array */
4: end for
5: while out has been changed do
6:    for B in a full order do
7:       /* By full order we mean an arbitrary length ordering */
8:       /* of basic blocks which includes every block at least once. */
9:       /* This ensures each block is visited infinitely often. */

10:      in[B] := ⋀_{B' a predecessor of B} out[B'];

11:      /* above, meet of empty set is I */
12:      out[B] := f_B(in[B]);
13:   end for
14: end while
```

Figure 2.2.2: Modified iterative data flow framework.

In much of the other work we have come across, CFGs are still constructed but a strong correlation to the syntactic program structure is maintained. This is useful, for example, for incremental analyses [2] and in tools that allow the user to interact with the optimizer to guide program transformations, but they still use a traditional compiler back end. The source level rewriters for Jumbo that we described in the previous section are somewhat different by that measure.

Chapter 3

A Framework for Staged Data Flow Analysis

In this Chapter we present our proposed data flow analysis framework and show how it can be used effectively for staging. *Staging* means that part of the analysis is done offline at compile time and then is completed at run time, hopefully with minimal effort.

In Section 3.1 we define the framework. Section 3.2 gives some examples. In Section 3.3 we extend the framework to handle scoping and then give a detailed type checking example in Section 3.4 making use of the additional functionality. Section 3.5 proves some properties of the framework that are useful for staging, and then in Section 3.6 we use these results to develop a generic algorithm for staging data flow analyses.

3.1 Basic Analysis Framework

The framework developed in this section does not inherently support staging, rather it serves as a foundation from which we work toward staging. When compared with the framework from Section 2.2, it has two important distinguishing features. The first is that it is purely functional, and the second is that it works directly on program source. Being functional rather than algorithmic makes it easier to reason about the framework in a rigorous, mathematical way, and using a source level representation of programs is important to us because we are working within the context of RTPG.

Our framework is made generic by keeping the data set and component functions as *parameters*, where they must be separately defined for each *analysis*. The set of data elements will always be denoted *Data*, and as we will show when we prove

$$
\begin{aligned}
\texttt{assign} &: \mathit{Node} \to \mathit{Var} \to \mathit{Exp} \to \mathit{Data} \\
\texttt{exp} &: \mathit{Exp} \to \mathit{Data} \\
; &: \mathit{Data} \to \mathit{Data} \to \mathit{Data} \\
\wedge &: \mathit{Data} \to \mathit{Data} \to \mathit{Data}
\end{aligned}
$$

Figure 3.1.1: Functions Parameterizing $\mathcal{F}[\![\cdot]\!]$

soundness in Chapter 4, *Data* can be used unchanged across the two frameworks. In the algorithm presented in Figure 2.2.1, the data elements are what are stored in the *in* and *out* arrays.

We also have essentially the same functional parameters as those used in the other framework: a set of transfer functions and a confluence operator. The major difference is that our transfer functions are explicitly built up from assignments and expressions using a function (the semi-colon (;) operator) to mimic sequential composition and therefore the generation of basic blocks. In Figure 3.1.1 we list all of the framework's functional parameters.

Now, in order to define our framework formally, we have to pin down exactly what we accept in terms of programs. Firgure 3.1.2 gives our program grammar, and thus an inductive definition of the the set of programs, which is denoted *Prgm*. The set of programs contains all of the usual structural constructs as well as breaks and labeled statements. We classify the expressions based on the result type. The reason that we have attached node labels to each construct is that many analyses will require that we produce a result that assigns a data value to each point in the AST, rather than generating one global result for the entire program. This corresponds to the per basic block arrays (*in* and *out*) in Figure 2.2.1.

Definition 3.1.1 (Data Flow Framework). Our data flow framework is defined

$$
\begin{array}{rcl}
\mathit{Var} & := & \ldots \text{(set of program variables)} \\
\mathit{Label} & := & \ldots \text{(set of program labels)} \\
\mathit{Type} & := & \tau_1 \mid \ldots \mid \tau_n \ \text{(set of data types)} \\
\tau_i - \mathit{Const} & := & \ldots \text{(set of constants of type } \tau_i) \\
\tau_i - \mathit{Exp} & := & \ldots \text{(set of expressions of type } \tau_i) \\
\mathit{Exp} & := & \tau_1 - \mathit{Exp} \mid \tau_1 - \mathit{Const} \mid \ldots \mid \tau_n - \mathit{Exp} \mid \tau_n - \mathit{Const} \\
\mathit{Node} & := & \ldots \text{(set of node labels)} \\
\mathit{Prgm} & := & \mathit{Node} : \texttt{skip} \\
 & \mid & \mathit{Node} : \mathit{Var} := \mathit{Exp} \\
 & \mid & \mathit{Node} : \texttt{break}\ \mathit{Label} \\
 & \mid & \mathit{Node} : \mathit{Label} : \mathit{Prgm} \\
 & \mid & \mathit{Node} : \mathit{Prgm}\ \texttt{;}\ \mathit{Prgm} \\
 & \mid & \mathit{Node} : \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Prgm}\ \texttt{else}\ \mathit{Prgm} \\
 & \mid & \mathit{Node} : \texttt{do}\ \mathit{Prgm}\ \texttt{while}\ \mathit{Exp}
\end{array}
$$

Figure 3.1.2: Defined sets and grammar for programs.

as a function with signature

$$
\begin{aligned}
\mathcal{F}[\![\cdot]\!] : \mathit{Prgm} \to (\mathit{Node} \rightharpoonup \mathit{Data}) \to (\mathit{Label} \rightharpoonup \mathit{Data}) \to \mathit{Data} \\
\to ((\mathit{Node} \rightharpoonup \mathit{Data}) \times (\mathit{Label} \rightharpoonup \mathit{Data}) \times \mathit{Data})
\end{aligned}
$$

For convenience and clarity we will refer to some of the aggregate types above using rather more convenient and descriptive names so that the typing becomes

$$
\mathcal{F}[\![\cdot]\!] : \mathit{Prgm} \to \mathit{DataEnv} \to \mathit{BreakEnv} \to \mathit{Result}
$$

$\mathcal{F}[\![\cdot]\!]$ is defined recursively by the equations given in Figure 3.1.3. The ifp function used in the loop case is an *iterated fixed point*. The function being iterated is given inside the first set of braces, and takes only a single parameter d_x. The initial value for this parameter is given in the second set of braces (d_1) and each iteration is

replaced with the third coordinate of the result of applying the function.

The data environment (*DataEnv*) associates a data value to the nodes in the program just as the *in* and *out* arrays do in the classical framework. The break environment (*BreakEnv*) is used essentially as a holding area for the data coming from paths with *dangling breaks*, that is, breaks with confluence points that have not yet been reached as values are computed recursively. The result of the analysis (*Result*) is a triple consisting of a data environment, a break environment, and a data element, in that order.

In order to support staging and to prove the soundness of our framework, we have to constrain somewhat the allowable analyses, just as we did with the framework in Section 2.2. Many of the restrictions are already axioms required by the CFG based framework and were explained previously, such as forcing the data set to be a lattice, for example. However, we add a number of other axioms which are required in proofs given in later sections.

The first axiom says that there will be a special token, `nil`, that is assumed to always be a part of the data set, but is not regular data in any way. It is essentially used as an error element. As can be seen in Figure 3.1.3, it is propagated after a break statement to mark the path as abnormal.

Axiom 3.1.1. The set of data elements is assumed to contain a special token, outside of the regular data contained in the set, called `nil`.

The next axiom states that *Data* is a lattice such that the confluence operator is the meet operation on the lattice. These are also requirements placed on the CFG based framework, as cited above. It should be noted that this axiom forces $\wedge$ to be associative, commutative, and idempotent.

Axiom 3.1.2. The set *Data* (with the `nil` token) is a lattice such that $\wedge$ is the meet operation on the lattice. Furthermore, `nil` is defined to be the top-most element, and therefore for all $d \in Data$

$$\begin{aligned} \texttt{nil} \wedge d &= d \\ d \wedge \texttt{nil} &= d \end{aligned}$$

$$\mathcal{F}[\![n : \texttt{skip}]\!]\ \varphi\ \eta\ d = (\varphi[n \mapsto d], \eta, d)$$

$$\mathcal{F}[\![n : x\ \texttt{:=}\ e]\!]\ \varphi\ \eta\ d = \text{let } d' := d; \texttt{assign}(n, x, e) \text{ in } (\varphi[n \mapsto d'], \eta, d')$$

$$\mathcal{F}[\![n : \texttt{break}\ l]\!]\ \varphi\ \eta\ d = (\varphi[n \mapsto d \wedge \eta(l)], \eta[l \mapsto d \wedge \eta(l)], \texttt{nil})$$

$$\mathcal{F}[\![n : l\ n_1 : P_1]\!]\ \varphi\ \eta\ d = \text{let } (\varphi_1, \eta_1, d_1) := \mathcal{F}[\![n_1 : P_1]\!]\ \varphi\ (\eta \setminus \{l\})\ d \text{ in } (\varphi_1[n \mapsto d_1 \wedge \eta(l)], \eta \setminus \{l\}, d_1 \wedge \eta_1(l))$$

$$\mathcal{F}[\![n : (n_1 : P_1)\ \texttt{;}\ (n_2 : P_2)]\!]\ \varphi\ \eta\ d = \text{let } (\varphi_1, \eta_1, d_1) := \mathcal{F}[\![n_1 : P_1]\!]\ \varphi\ \eta\ d \text{ in let } \varphi_2, \eta_2, d_2) := \mathcal{F}[\![n_2 : P_2]\!]\ \varphi_1\ \eta_1\ d_1 \text{ in } (\varphi_2[n \mapsto d_2], \eta_2, d_2)$$

$$\mathcal{F}[\![n : \texttt{if}\ e\ \texttt{then}\ n_1 : P_1\ \texttt{else}\ n_2 : P_2\ ]\!]\ \varphi\ \eta\ d = \text{let } (\varphi_1, \eta_1, d_1) := \mathcal{F}[\![n_1 : P_1]\!]\ \varphi\ \eta\ d; \texttt{exp}(e) \quad (\varphi_2, \eta_2, d_2) := \mathcal{F}[\![n_2 : P_2]\!]\ \varphi\ \eta\ d; \texttt{exp}(e) \text{ in } ((\varphi_1 \cup \varphi_2)[n \mapsto d_1 \wedge d_2], \eta_1 \wedge \eta_2, d_1 \wedge d_2)$$

$$\mathcal{F}[\![n : \texttt{do}\ n_1 : P_1\ \texttt{while}\ e]\!]\ \varphi\ \eta\ d = \text{let } (\varphi_1, \eta_1, d_1) := \mathcal{F}[\![n_1 : P_1]\!]\ \varphi\ \eta\ d \text{ in let } (\varphi_1', \eta_1', d_1') := ifp\{\mathcal{F}[\![n_1 : P_1]\!]\ \varphi\ \eta\ (d_x; \texttt{exp}(e) \wedge d)\}\{d_1\} \text{ in } (\varphi_1'[n \mapsto d_1'; \texttt{exp}(e)], \eta_1', d_1'; \texttt{exp}(e))$$

Figure 3.1.3: Definition of our framework, $\mathcal{F}[\![\cdot]\!]$

We also assume that there is an identity element for the semi-colon operation.

Axiom 3.1.3. There is an element $I \in Data$ which acts as the identity element for the semi-colon operation, so that for all $d \in Data$

$$\begin{aligned} I; d &= d \\ d; I &= d \end{aligned}$$

Lastly, we assume that the semi-colon operator distributes over the confluence operator. This axiom in particular is used by our main theorem leading up to our staging algorithm. Note that it is different from the distributivity condition described by [1].

Axiom 3.1.4. For all $d_1, d_2, d_3 \in Data$ the following equivalence, called *distributivity*, holds

$$d_1; (d_2 \wedge d_3) = d_1; d_2 \wedge d_1; d_3$$

There are four other operations that are used in the framework but that we have not yet defined. It was most likely clear from context what was meant, but for the sake of formality (and to cover the corner cases) we define these operations below. Most are operations on partial functions and are used in the updates of the data and break environments. We use Dom and Cod to denote the domain and codomain of a partial function, respectively. We take the domain of a partial function to include the obviously related values on which the function is undefined.

Definition 3.1.2. $(f \wedge f)$ Let f_1 and f_2 be (partial) functions with the same domain and with codomain equal to $Data$. Then we define

$$(f_1 \wedge f_2) : Dom(f_1) \rightharpoonup Data$$

such that for any $x \in Dom(f_1)$

$$(f_1 \wedge f_2)(x) = \begin{cases} f_1(x) \wedge f_2(x) & f_1(x)\downarrow \text{ and } f_2(x)\downarrow \\ \uparrow & f_1(x)\uparrow \text{ and } f_2(x)\uparrow \\ f_2(x) & f_1(x)\uparrow \text{ and } f_2(x)\downarrow \\ f_1(x) & f_1(x)\downarrow \text{ and } f_2(x)\uparrow \end{cases}$$

Definition 3.1.3. ($f[\ ]$) Let f be a partial function and take $x \in Dom(f)$ and $y \in Cod(f)$. Then we define

$$f[x \mapsto y] : Dom(f) \rightharpoonup Cod(f)$$

such that for any $z \in Dom(f)$

$$f[x \mapsto y](z) = \begin{cases} y & \text{if } z = x \\ f(z) & \text{otherwise} \end{cases}$$

Definition 3.1.4. ($f \cup f$) Let f_1 and f_2 be (partial) functions with the same domain and codomain. Then we define

$$(f_1 \cup f_2) : Dom(f_1) \rightharpoonup Cod(f_1)$$

such that for any $x \in Dom(f_1)$

$$(f_1 \cup f_2)(x) = \begin{cases} \uparrow & f_1(x)\downarrow \text{ and } f_2(x)\downarrow \text{ and } f_1(x) \neq f_2(x) \\ f_1(x) & f_1(x)\downarrow \text{ and } f_2(x)\downarrow \text{ and } f_1(x) = f_2(x) \\ \uparrow & f_1(x)\uparrow \text{ and } f_2(x)\uparrow \\ f_2(x) & f_1(x)\uparrow \text{ and } f_2(x)\downarrow \\ f_1(x) & f_1(x)\downarrow \text{ and } f_2(x)\uparrow \end{cases}$$

Definition 3.1.5. ($f \setminus \{x\}$) Let f be a partial function with $x \in Dom(f)$. Then we

$$x \in Var \qquad l \in Label \qquad n \in Node$$
$$e \in Exp \qquad d \in Data$$
$$\varphi \in DataEnv \quad \eta \in BreakEnv \quad P \in Prgm$$

Figure 3.1.4: Conventions on variable types

define

$$f \setminus \{x\} = Dom(f) \rightharpoonup Cod(f)$$

such that for any $y \in Dom(f)$

$$(f \setminus \{x\})(y) = \begin{cases} \uparrow & \text{if } y = x \\ f(y) & \text{otherwise} \end{cases}$$

The intuition behind the framework is that at each recursive step, the local control flow information is applied to the recursive results from *below* the current syntactic point. In most cases this local information is enough to take care of all of the control flow. The only exception is for break statements. In the case of breaks, a discharging of the control flow information must wait until the label being broken to is seen as recursion proceeds upward. Thus, partial data is stored in the break environment waiting until the confluence point is found. Over the next few sections, multiple examples will be presented, both for this framework and a slightly strengthened one. These examples are useful for further enhancing intuition about the framework.

As a matter of convention, we will assume throughout the Thesis that the variable names, as well as their obvious variations, given in Figure 3.1.4 have types associated to them in Figure 3.1.4.

3.2 Examples

This section presents two common data flow problems, then formulates them as analyses within our framework. The first problem validates that variable uses are preceded by definitions of that variable and the second computes reaching definitions. We think that these two examples are good empirical evidence of the utility of our

framework and also the ease with which it can be used.

3.2.1 Variable Initialization

The first example analysis that we will consider for implementation in our framework is *variable initialization*. This analysis comes in a couple of flavors, but the basic idea is to enumerate a set of variables which have uses that lie on a path with no preceding definitions An assignment statement is a *definition* of variable x if x occurs on the left side of the assignment, that is, it is the variable that is assigned to. This phrasing indicates one *global* solution for the entire program rather than associating a piece of data to every node or basic block. Therefore, we need to modify the problem statement somewhat to accommodate this. Our variable initialization analysis populates each node with the set of variables that may have been used without a previous declaration *at the time execution reaches the node*. Note that if the program fragment has a post-dominating statement or block, then the value associated with it is the global result described above.

This is an easy analysis to implement in the CFG based framework. The data values that get operated on and associated to each basic block are pairs of sets, one representing the variables that have definitely (i.e. on *every* path) been defined any time just after the basic block is executed and the second being those variables that may have been used without a previous definition up to and including the execution point. Roughly speaking, the transfer function associated with each basic block adds in any incoming, defined variables to those defined inside the block; and for those variables with uses in the block before definitions in the block we check the incoming data to see if there were definitions before executing the block. When paths merge and we apply the confluence operator, we keep all variables that may have been used without being defined, and update the set of definitely defined variables to contain only those on all incoming paths.

In our framework we basically use the same definitions all around: the same definition of data and as usual semi-colon does the work of the transfer functions,

and in both frameworks the confluence operators work the same. Therefore,

$$Data = \wp(Var) \times \wp(Var)$$

and the other definitions fall out easily using an auxiliary function that separates out the variables used in an expression.

Definition 3.2.1. (`vars-used`)

$$\texttt{vars-used} : Exp \to \wp(Var)$$

is defined such that $\texttt{vars-used}(e)$ is the set of variables used in the expression e.

Now it is easy to define each of the functions required by the framework. We start with `assign` and `exp` then move on to the semi-colon and the confluence operator. The `assign` function adds the variable on the left hand side of the definition to the definitely defined set and all of the variables in the expression on the right side of the assignment statement are added to the set of variables which are possibly used before being defined.

Definition 3.2.2. (`assign`)

$$\texttt{assign} : Node \to Var \to Exp \to Data$$

is defined such that

$$\texttt{assign}(n, x, e) = (\{x\}, \texttt{vars-used}(e))$$

The `exp` function is essentially equivalent to `vars-used` except that a place-holding empty set is used for the set of definitely defined variables.

Definition 3.2.3. (`exp`)

$$\texttt{exp} : Exp \to Data$$

is defined such that

$$\texttt{exp}(e) = (\emptyset, \texttt{vars-used}(e))$$

Now, the semi-colon operator uses the definitely defined variables in the first argument (or on the left) to prune the set of uninitialized variables in the second argument. The set of definitely defined variables grows by a simple unioning of those sets in the first and second arguments.

Definition 3.2.4. (;)

$$; : \mathit{Data} \to \mathit{Data} \to \mathit{Data}$$

is defined such that

$$(D_1, U_1) \,;\, (D_2, U_2) = (D_1 \cup D_2, U_1 \cup (U_2 \setminus D_1))$$

The confluence operator takes the intersection of the sets of definitely defined variables because for a variable to have definitely had a definition after the point of confluence it must have had a definition on every path leading in. The sets of uninitialized variables are unioned together.

Definition 3.2.5. ($\wedge$)

$$\wedge : \mathit{Data} \to \mathit{Data} \to \mathit{Data}$$

is defined such that

$$(D_1, U_1) \wedge (D_2, U_2) = (D_1 \cap D_2, U_1 \cup U_2)$$

The concrete example that we give next shows how our variable initialization analysis works. The intent is to help build intuition about how the framework and the individual analyses work together by showing the data values that are eventually attributed to each program node via the data environment. Consider the program, given as an AST, in Figure 3.2.1 and for now disregard the annotations associated with each node, those values will be discussed next.

We can apply $\mathcal{F}[\![\cdot]\!]$ directly to $n : P$ and get back a complex function, but this does not really facilitate any intuition because all of the functions that we have defined would still be hidden underneath the symbols (e.g. `assign`). The most useful example is to show how the recursion gets applied so as to finally end up with

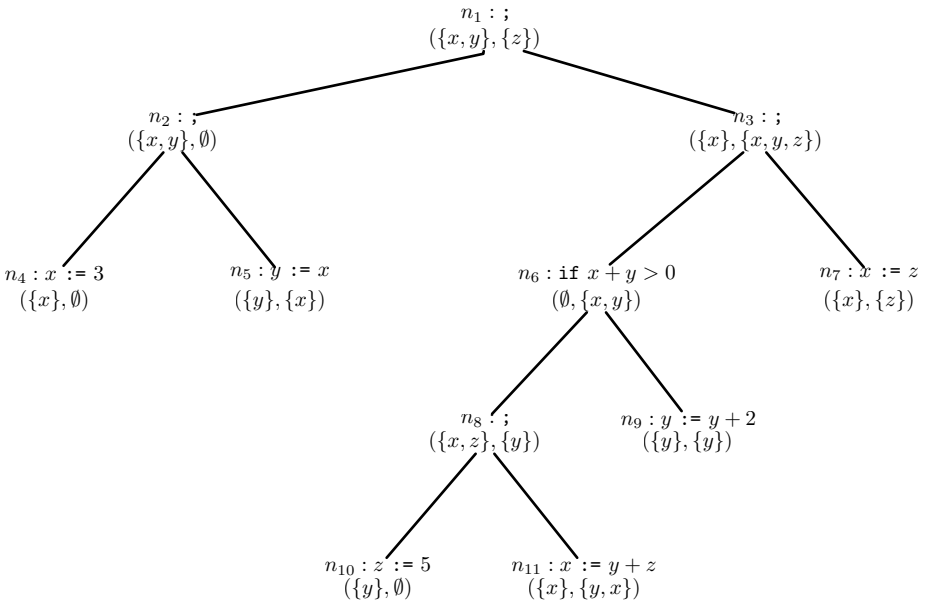


Figure 3.2.1: Variable initialization example.

the final answer. Therefore, in Figure 3.2.1 we annotate each node n_i with the value $\varphi(n_i)$, where $(\varphi, \eta, d) = \mathcal{F}[\![n_i : P_i]\!]\ \epsilon\ \epsilon\ (\emptyset, \emptyset)$; that is, the result at node i when $\mathcal{F}[\![\cdot]\!]$ is applied with some default arguments to the sub-program rooted at that node.

In Section 3.5 when we discuss staging, we will show that *distributivity* of ; over $\wedge$ can be used to generate a natural, efficient method of staging. The variable initialization analysis in this section satisfies the distributivity property, which we now prove, and thus can be staged using the methods at the end of this chapter.

Theorem 3.2.1. *For any* (D_1, U_1),(D_2, U_2),$(D_3, U_3) \in$ *Data,* ; *distributes over the* $\wedge$. *Or stated another way*

$$(D_1, U_1); ((D_2, U_2) \wedge (D_3, U_3)) = ((D_1, U_1); (D_2, U_2)) \wedge ((D_1, U_1); (D_3, U_3))$$

Proof. This equivalence is easy to derive by expanding the definitions of semi-colon

and the confluence operator given above.

$$(D_1, U_1); ((D_2, U_2) \wedge (D_3, U_3)) = (D_1, U_1); (D_2 \cap D_3, U_2 \cup U_3)$$
$$= (D_1 \cup (D_2 \cap D_3), U_1 \cup ((U_2 \cup U_3) \setminus D_1))$$

and

$$((D_1, U_1); (D_2, U_2)) \wedge ((D_1, U_1); (D_3, U_3))$$
$$= (D_1 \cup D_2, U_1 \cup (U_1 \setminus D_1)) \wedge (D_1 \cup D_3, U_1 \cup (U_3 \setminus D_1)$$
$$= (D_1 \cup (D_2 \cap D_3), U_1 \cup ((U_2 \cup U_3) \setminus D_1))$$

which shows that the equality required by the theorem holds and that the distributivity property holds for our implementation of the variable initialization analysis. □

3.2.2 Reaching Definitions

The second analysis that we give as an example is *reaching definitions*. A *variable definition* for variable x happens when x occurs on the left side of an assignment statement. Languages that allow variable aliasing may present some ambiguities with respect to variable definitions, but our language does not have aliasing and so we do not address it further. The basics of treating ambiguous references is described in [1]. Unlike variable initialization, the solution to a reaching definitions problem is inherently *local*, there is no notion of a *global* solution. Therefore, there is no need for us to morph the problem into an equivalent one that our framework can answer explicitly.

When reaching definitions is implemented in the CFG framework each basic block gets two data sets : a set of definitions *generated* by the basic block (the *gen* set), and a set of variables whose definitions would be *killed* by the statements in the basic block (the *kill* set). Each block's gen set also serves as the starting set of definitions that reach the block. The transfer functions add to their own set of reaching definitions those coming into the block and given as an argument to the

transfer function, but then output only those definitions that are for variables *not* killed by the block.

As with variable initialization we model our analysis using the corresponding implementation in the CFG based framework. In our framework, because definitions can only happen at assignment statements and each assignment statement gets a unique node name, we can encode definitions as a pair containing a variable and a node.

$$Def = Var \times Node$$

Now to mimic the gen/reaching defs set and the kill set we simply define *Data* to be a pair consisting of a set of definitions and a set of variables killed by the current node. We will continue to call the first component of data the *gen* set and the second component the *kill* set.

$$Data = \wp(Def) \times \wp(Var)$$

In the rest of this section, we use use R to denote values of type $\wp(Def)$ and we use K to denote values of type $\wp(Var)$. The definitions of `assign` and `exp` are simple; for every assignment, the definition is added to the gen set and, in addition, the variable being assigned to is added to the kill set because the definition is unambiguous.

Definition 3.2.6. (`assign`)

$$\texttt{assign} : Node \rightarrow Var \rightarrow Exp \rightarrow Data$$

is defined such that

$$\texttt{assign}(n, x, e) = (\{(x, n)\}, \{x\})$$

Expressions neither generate nor kill any definitions, and therefore `exp` always returns a pair of empty sets.

Definition 3.2.7. (`exp`)

$$\texttt{exp} : Exp \rightarrow Data$$

is defined such that

$$\texttt{exp}(e) = (\emptyset, \emptyset)$$

As we have pointed out previously, sequentially composing two programs acts similarly to moving through a basic block in a CFG which has exactly one entry and one exit point. Therefore, incoming definitions should only reach through the statement if the kill set given in the second argument does not contain the variable in the particular definition. In order to sieve the set of reaching definitions by the kill set, we need an auxiliary function.

Definition 3.2.8. (`kill`)

$$\texttt{kill} : \wp(\mathit{Def}) \to \wp(\mathit{Var}) \to \wp(\mathit{Def})$$

is defined such that $(x, n) \in \texttt{kill}(R, K)$ if and only if $(x, n) \in R$ and $x \notin K$.

The definition of the semi-colon operator now falls out easily.

Definition 3.2.9. (;)

$$; : \mathit{Data} \to \mathit{Data} \to \mathit{Data}$$

is defined such that

$$(R_1, K_1); (R_2, K_2) = (\texttt{kill}(R_1, K_2) \cup R_2, K_1 \cup K_2)$$

The confluence operator represents a number of paths being merged, and thus the set of reaching definitions should be the union of the definitions along each individual path and the set of killed variables are only those that are killed on *every* path.

Definition 3.2.10. ($\wedge$)

$$\wedge : \mathit{Data} \to \mathit{Data} \to \mathit{Data}$$

is defined such that

$$(R_1, K_1) \wedge (R_2, K_2) = (R_1 \cup R_2, K_1 \cap K_2)$$

Figure 3.2.2 shows the reaching definitions analysis applied to an example program. It is the same program we used to demonstrate the variable initialization

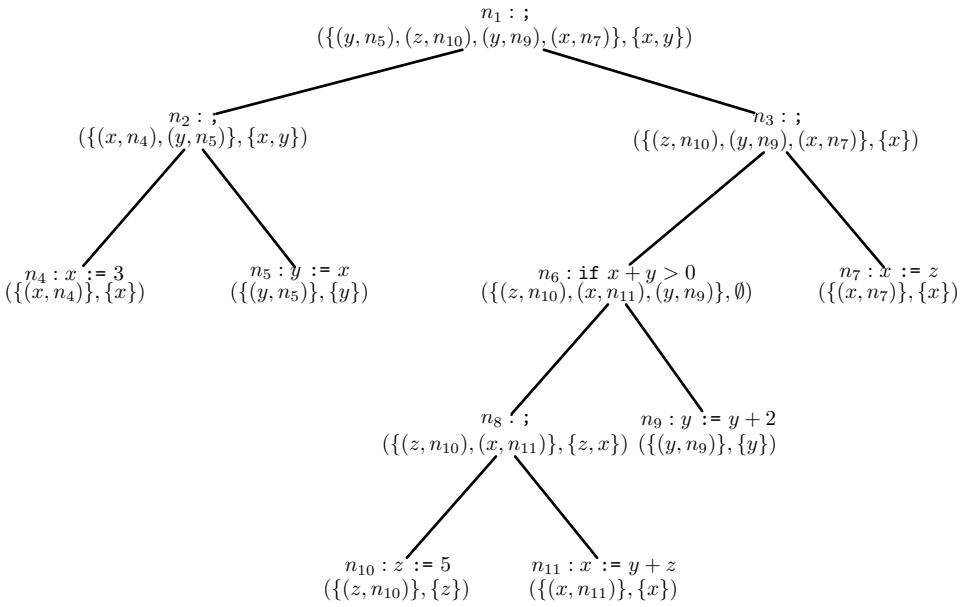


Figure 3.2.1: Reaching definitions example.

analysis. Again, we have annotated each node with the result of applying $\mathcal{F}[\![\cdot]\!]$ to the node directly and with some reasonable canonical arguments. Specifically, we annotate each node n_i with the value $\varphi(n_i)$, where $(\varphi, \eta, d) = \mathcal{F}[\![n_i : P_i]\!]\ \epsilon\ \epsilon\ (\emptyset, \emptyset)$.

For the reasons described earlier, it is desirable for our analyses to have the property that ; distributes over the $\wedge$ operator. The following theorem shows that our reaching definitions analysis is distributive. We first prove a small Lemma, and then the result will fall out easily.

Lemma 3.2.1.

$$\mathtt{kill}(R, K_1) \cup \mathtt{kill}(R, K_2) = \mathtt{kill}(R, K_1 \cap K_2)$$

Proof. By definition $(x, n) \in \mathtt{kill}(R, K_1)$ if and only if $(x, n) \in R$ and $x \notin K_1$, and similarly $(x, n) \in \mathtt{kill}(R, K_2)$ if and only if $(x, n) \in R$ and $x \notin K_2$. Therefore

$$(x, n) \in \mathtt{kill}(R, K_1) \cup \mathtt{kill}(R, K_2)$$

if and only if $(x, n) \in R$ and $x \notin K_1$ and $x \notin K_2$. This is exactly the same as $(x, n) \in \texttt{kill}(R, K_1 \cap K_2)$. □

Theorem 3.2.2. *The ; operator distributes over $\wedge$ operator, that is*

$$(R_1, K_1); ((R_2, K_2) \wedge (R_3, K_3)) = ((R_1, K_1); (R_2, K_2)) \wedge ((R_1, K_1); (R_3, K_3))$$

Proof. Expanding the left hand side of the equation we get

$$\begin{aligned}(R_1, K_1); ((R_2, K_2) \wedge (R_3, K_3)) &= (R_1, K_1); (R_2 \cup R_3, K_2 \cap K_3) \\ &= (\texttt{kill}(R_1, K_2 \cap K_3) \cup R_2 \cup R_3, K_1 \cup (K_2 \cap K_3))\end{aligned}$$

and expanding the right hand side of the equation we get

$$\begin{aligned}&((R_1, K_1); (R_2, K_2)) \wedge ((R_1, K_1); (R_3, K_3)) \\ &\quad = (\texttt{kill}(R_1, K_2) \cup R_2, K_1 \cup K_2) \wedge (\texttt{kill}(R_1, K_3) \cup R_3, K_1 \cup K_3) \\ &\quad = (\texttt{kill}(R_1, K_2) \cup R_2 \cup \texttt{kill}(R_1, K_3) \cup R_3, (K_1 \cup K_2) \cap (K_1 \cup K_3)) \\ &\quad = (\texttt{kill}(R_1, K_2 \cap K_3) \cup R_2 \cup R_3, K_1 \cup (K_2 \cap K_3))\end{aligned}$$

where the last equivalence holds by applying the Lemma. □

3.3 Extending the Framework with Scoping

In this section we present a small extension to the framework presented in Section 3.1. The extension changes the set of allowed programs to include variable declarations, and adds to the framework the ability to reason about the declarations and the *scope* in which they appear. Not only can we do more analyses when we have variable declaration and scoping information (e.g. type checking), but we can also create more sophisticated versions of other data-flow analyses; both the analysis in Section 3.2.1 and Section 3.2.2 can be meaningfully extended to account for scopes, for example.

$$
\begin{array}{rcl}
\mathit{Type} & := & \ldots \text{(set of types)} \\
\mathit{Prgm} & := & \mathit{Node} : \texttt{skip} \\
& | & \mathit{Node} : \mathit{Type}\ \mathit{Var} \\
& | & \mathit{Node} : \mathit{Var}\ :=\ \mathit{ArithmeticExp} \\
& | & \mathit{Node} : \mathit{Var}\ :=\ \mathit{BooleanExp} \\
& | & \mathit{Node} : \texttt{break}\ \mathit{Label} \\
& | & \mathit{Node} : \mathit{Label}\ \texttt{:}\ \mathit{Prgm} \\
& | & \mathit{Node} : \mathit{Prgm}\ \texttt{;}\ \mathit{Prgm} \\
& | & \mathit{Node} : \texttt{if}\ \mathit{BooleanExp}\ \texttt{then}\ \mathit{Prgm}\ \texttt{else}\ \mathit{Prgm} \\
& | & \mathit{Node} : \texttt{do}\ \mathit{Prgm}\ \texttt{while}\ \mathit{BooleanExp}
\end{array}
$$

Figure 3.3.1: Extended grammar which includes variable declarations.

Instead of having a new syntactic construct for the explicit purpose of creating a new scope, we do what most programming languages do (e.g. C and Java) and implicitly create scopes in some reasonable circumstances. For example, we constrain our framework by forcing each branch of a conditional statement to enter a new scope. This is a reasonable constraint; consider what happens if one branch has a variable declaration while the other does not and no new scope is created. Is the variable usable after the confluence point? We still keep broad generality however, by also creating a new scope for each labeled statement, which can be used arbitrarily to create a new scope. The new program grammar that we treat is given in Figure 3.3; we use, and will continue to use the variable τ to represent an arbitrary type, that is, $\tau \in \mathit{Type}$.

Figure 3.3 presents the new framework in its entirety. Because of the new syntactic constructs for variable declarations we have had to add a parameter to the framework for calculating an initial data value for declarations. The function takes a node, variable, and type as arguments, so

$$\texttt{decl} : \mathit{Node} \to \mathit{Type} \to \mathit{Var} \to \mathit{Data}$$

Now, for the reasons stated above we have decided against introducing a new, single syntactic construct to manage scope. So instead we have to modify the definition of $\mathcal{F}[\![\cdot]\!]$ for those constructs that affect scope. This introduces two new parameters into the framework which are, like the original ones (i.e. `assign`,`exp`,;, and $\wedge$), functions. The first

$$\texttt{es} : \mathit{Node} \rightarrow \mathit{Data} \rightarrow \mathit{Data}$$

modifies a piece of data to account for scoping; the symbol stands for "enter scope". The *Node* argument is intended only to supply a unique identifier for the scope, since we assume uniquely labeled nodes in our programs. The second function does the opposite, removing any remnant information from a scope when control is being passed back out of it. The `ls` function has the same type as `es`.

$$\texttt{ls} : \mathit{Node} \rightarrow \mathit{Data} \rightarrow \mathit{Data}$$

As can be seen in Figure 3.3, a new scope is created for each branch of a conditional statement, the body of each labeled statement, and also the body of every loop. For any subprogram $n : P$ which starts a new scope, the framework is modified so that instead of calculating

$$(\varphi', \eta', d') = \mathcal{F}[\![n : P]\!]\ \varphi\ \eta\ d$$

the third argument is modified by the `es` function. Therefore we calculate

$$(\varphi', \eta', d') = \mathcal{F}[\![n : P]\!]\ \varphi\ \eta\ \texttt{es}(n, d)$$

And instead of using d' directly, we make sure to remove all of the remnant information specific to the scope created for $n : P$ by applying $\texttt{ls}(n, d')$. We also have to do the same thing when we use a value stored in the break environment because these paths may come from arbitrarily nested scopes. The use of $\eta(l)$ in the calculation for labeled statements is therefore replaced with $\texttt{ls}(n, \eta(l))$.

Finally, we note that the CFG based framework is general enough already to handle scoping. The reason is that instead of building transfer functions up from some algebra as we do with `assign` and semi-colon, each basic block gets its own,

arbitrary transfer function. These functions could similarly manage scope by applying operations like `es` and `ls` whenever the block in question is associated with the start or end of a scope, respectively.

3.4 Examples

3.4.1 Simple Typing

The expanded framework can be used for the purpose of type checking of our expanded programming language. Specifically, in this section we describe an analysis for making typing judgments in the type and effect system given in Figure 3.4.1. Although it has not been formalized explicitly as part of the grammars in any of Figures 3.1.1 or 3.3 we assume that the *binary* operators used to build expressions require specific types for their arguments and also produce results of a specific type. Thinking of applications of these operators using infix notation, we will speak of the *left* and *right* arguments. The type and effect system in Figure 3.4.1 only deals with binary operators, but it would be easy to extend our framework with operators of arbitrary arity and types. Therefore, we will use $BinaryOp \subseteq Op$ in this section to represent the set of binary operators.

Definition 3.4.1. (`type`, `rtype`, and `ltype`) We just explained above that we assume that each binary operator is assumed to require a specific type for each argument and always returns values of a specific, predetermined type. The following functions compute these types.

$$\begin{aligned} \texttt{type} &: BinaryOp \to Type \\ \texttt{ltype} &: BinaryOp \to Type \\ \texttt{rtype} &: BinaryOp \to Type \end{aligned}$$

It will be convenient however, to extend `type` to apply to a wider range of values. Therefore we overload the operator so that

$$\begin{aligned} \texttt{type} &: \mathit{Const} \rightarrow \mathit{Type} \\ \texttt{type} &: \mathit{Var} \rightarrow \mathit{Var} \end{aligned}$$

and define $\texttt{type}(x) = x$ for all $x \in \mathit{Var}$ and $\texttt{type}(c)$ is an inherent property of each constant.

The typing system described in Figure 3.4.1 defines a 4-ary relation; the result of applying the relation to specific arguments is called a *typing judgment*. Note that the typing system requires that there is a Boolean data type, `bool`, since this is what is normally expected in conditional statements and for loop control. A positive typing judgment describes a set of arguments for which the relation holds, and is written

$$\Gamma \vdash x : y / \Gamma'$$

where $\Gamma, \Gamma' \in \mathit{TyEv}$ are *typing environments*. The relation means that in environment Γ, x correctly types to y and produces the *effect* Γ'. The effect is the set of variable to type mappings that are induced by x. A typing environment is defined as

$$\mathit{TyEv} = \mathit{Var} \rightharpoonup \mathit{Type}$$

x is either a program or an expression, and if x is a program then y is the special token/type `prgm`, and otherwise y is an arbitrary data type from the set *Type*. A negative typing judgment, that is when the relation does not hold, is written similarly.

$$\Gamma \nvdash x : y / \Gamma'$$

In a positive typing judgment like the one given above, y is called the *type of* x, and Γ' is the *effect of typing* x *in* Γ. It is easily seen that we are correct to say *the* type, implying uniqueness. Similarly, but perhaps less intuitively, we are also justified in referring to *the* effect of typing x in Γ. If y is well-typed with respect to Γ,

$$\mathcal{F}[\![n : \texttt{skip}]\!]\ \varphi\ \eta\ d = (\varphi[n \mapsto d], \eta, d)$$

$$\mathcal{F}[\![n : x\ \texttt{:=}\ e]\!]\ \varphi\ \eta\ d = \text{let } d' := d; \texttt{assign}(n, x, e) \text{ in } (\varphi[n \mapsto d'], \eta, d')$$

$$\mathcal{F}[\![n : \tau\ x]\!]\ \varphi\ \eta\ d = \text{let } d' := d; \texttt{decl}(n, \tau, x) \text{ in } (\varphi[n \mapsto d'], \eta, d')$$

$$\mathcal{F}[\![n : \texttt{break}\ l]\!]\ \varphi\ \eta\ d = (\varphi[n \mapsto d \wedge \eta(l)], \eta[l \mapsto d \wedge \eta(l)], \texttt{nil})$$

$$\mathcal{F}[\![n : l\ n_1 : P_1]\!]\ \varphi\ \eta\ d = \text{let } (\varphi_1, \eta_1, d_1) := \mathcal{F}[\![n_1 : P_1]\!]\ \varphi\ (\eta \setminus \{l\})\ \texttt{es}(n_1, d) \text{ in } (\varphi_1[n \mapsto \texttt{ls}(n_1, d_1) \wedge \texttt{ls}(n_1, \eta_1(l))], \eta_1 \setminus \{l\}, \texttt{ls}(n_1, d_1) \wedge \texttt{ls}(n_1, \eta_1(l)))$$

$$\mathcal{F}[\![n : (n_1 : P_1)\ ;\ (n_2 : P_2)]\!]\ \varphi\ \eta\ d = \text{let } (\varphi_1, \eta_1, d_1) := \mathcal{F}[\![n_1 : P_1]\!]\ \varphi\ \eta\ d \text{ in let } \varphi_2, \eta_2, d_2) := \mathcal{F}[\![n_2 : P_2]\!]\ \varphi_1\ \eta_1\ d_1 \text{ in } (\varphi_2[n \mapsto d_2], \eta_2, d_2)$$

$$\mathcal{F}[\![n : \texttt{if}\ e\ \texttt{then}\ n_1 : P_1\ \texttt{else}\ n_2 : P_2\ ]\!]\ \varphi\ \eta\ d = \text{let } (\varphi_1, \eta_1, d_1) := \mathcal{F}[\![n_1 : P_1]\!]\ \varphi\ \eta\ \texttt{es}(n_1, d; \texttt{exp}(e)) \quad (\varphi_2, \eta_2, d_2) := \mathcal{F}[\![n_2 : P_2]\!]\ \varphi\ \eta\ \texttt{es}(n_2, d; \texttt{exp}(e)) \text{ in } ((\varphi_1 \cup \varphi_2)[n \mapsto \texttt{ls}(n_1, d_1) \wedge \texttt{ls}(n_2, d_2)], \eta_1 \wedge \eta_2, \texttt{ls}(n_1, d_1) \wedge \texttt{ls}(n_2, d_2))$$

$$\mathcal{F}[\![n : \texttt{do}\ n_1 : P_1\ \texttt{while}\ e]\!]\ \varphi\ \eta\ d = \text{let } (\varphi_1, \eta_1, d_1) := \mathcal{F}[\![n_1 : P_1]\!]\ \varphi\ \eta\ \texttt{es}(n_1, d) \text{ in let } (\varphi_1', \eta_1', d_1') := ifp\{\mathcal{F}[\![n_1 : P_1]\!]\ \varphi\ \eta\ \texttt{es}(n_1, \texttt{ls}(n_1, d_x); \texttt{exp}(e) \wedge d)\}\{d_1\} \text{ in } (\varphi_1'[n \mapsto \texttt{ls}(n_1, d_1'); \texttt{exp}(e)], \eta_1', \texttt{ls}(n_1, d_1'); \texttt{exp}(e))$$

Figure 3.3.1: Definition of the extended framework, $\mathcal{F}[\![\cdot]\!]$

Constants

$$\frac{c \in \mathit{Const}}{\Gamma \vdash c : \mathtt{type}(c)/\Gamma}$$

Variable Uses

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau/\Gamma}$$

Declarations

$$\frac{\Gamma(x) \uparrow}{\Gamma \vdash n : \tau\ x : \mathtt{prgm}/\Gamma[x \mapsto \tau]}$$

Compound Expressions

$$\frac{\Gamma \vdash e_1 : \mathtt{ltype}(\oplus)/\Gamma \quad \Gamma \vdash e_2 : \mathtt{rtype}(\oplus)/\Gamma \quad \oplus \in \mathit{BinaryOp}}{\Gamma \vdash e_1 \oplus e_2 : \mathtt{type}(\oplus)/\Gamma}$$

Skip

$$\frac{}{\Gamma \vdash n : \mathtt{skip} : \mathtt{prgm}/\Gamma}$$

Break

$$\frac{}{\Gamma \vdash n : \mathtt{break}\ l : \mathtt{prgm}/\Gamma}$$

Assignment

$$\frac{\Gamma \vdash x : \tau/\Gamma \quad \Gamma \vdash e : \tau/\Gamma}{\Gamma \vdash n : x\ :=\ e : \mathtt{prgm}/\Gamma}$$

Labeled Statements

$$\frac{\Gamma \vdash (n_1 : P_1) : \mathtt{prgm}/\Gamma'}{\Gamma \vdash n : l\ :\ (n_1 : P_1) : \mathtt{prgm}/\Gamma}$$

Sequential Composition

$$\frac{\Gamma \vdash (n_1 : P_1) : \mathtt{prgm}/\Gamma' \quad \Gamma' \vdash (n_2 : P_2) : \mathtt{prgm}/\Gamma''}{\Gamma \vdash n : (n_1 : P_1)\,;(n_2 : P_2) : \mathtt{prgm}/\Gamma''}$$

Conditional Statements

$$\frac{\Gamma \vdash e : \mathtt{bool}/\Gamma \quad \Gamma \vdash (n_1 : P_1) : \mathtt{prgm}/\Gamma' \quad \Gamma \vdash (n_2 : P_2) : \mathtt{prgm}/\Gamma''}{\Gamma \vdash n : \mathtt{if}\ e\ \mathtt{then}\ (n_1 : P_1)\ \mathtt{else}\ (n_2 : P_2) : \mathtt{prgm}/\Gamma}$$

Loops

$$\frac{\Gamma \vdash e : \mathtt{bool}/\Gamma \quad \Gamma \vdash (n_1 : P_1) : \mathtt{prgm}/\Gamma'}{\Gamma \vdash n : \mathtt{do}\ (n_1 : P_1)\ \mathtt{while}\ e : \mathtt{prgm}/\Gamma}$$

Figure 3.4.1: Inference rules for a simple type and effect system.

then there is only one set of effects that will witness this. With these facts in mind, it is clear that deciding whether x is well-typed in Γ can be reduced into a finite set of obligations on x and Γ, and that these can be gathered recursively by traversing the structure of x. This is the usual case for such typing systems. Therefore our implementation of type checking will work by aggregating together some number of *obligations* that will need to be *discharged* in order for the program/expression to be determined to be well-typed. Obligations are discharged as the type environment is filled in and evaluated.

As usual we first define the set of data values ($Data$) that will be manipulated:

$$Data = TySt \times Oblg$$

where $TySt$ is a *type stack* and $Oblg$ is a set of *obligations*. The type stack is defined to be a list

$$TySt = \langle \star, TyEv \rangle :: (Node \times TyEv)^*$$

where :: is used to append two lists (henceforth, type $List$), * represents iteration, and $\langle \ldots \rangle$ is used to represent tuples that are used as list elements. The token $\star$ is used specially to denote the top of the stack and throughout this section we will use ϵ in the context of lists to mean the empty list. The intent of the $TySt$ component of $Data$ is to be able to recover the correct type environment at any program point. Later in this section we will formalize this connection by defining a function that computes the type environment given a type stack as an argument. The second component in each element of $Data$ is a set of obligations that must be discharged. Individual obligations fall into one of four categories:

- (**Type 1**) These conditions are associated with a pair of program variables and assert that the two program variables must have the same type.
- (**Type 2**) These conditions are associated with a pair consisting of a program variable and a type (from $Type$); the condition asserts that the variable type equals the given type.
- (**Type 3**) These conditions are associated with a single program variable and

assert that the variable has not been given a type in the current scope.

- (**Type 4**) These conditions equate two given types, and are merely for convenience; they do not need any further evaluation but only remain to be purged from the obligation set.

Formally, we define

$$\begin{aligned} \mathit{Oblg} &= \wp(\mathit{Ot1} \cup \mathit{Ot2} \cup \mathit{Ot3} \cup \mathit{Ot4}) \cup \{\texttt{error}\} \\ \mathit{Ot1} &= \mathit{Var} \times \mathit{Var} \\ \mathit{Ot2} &= \mathit{Var} \times \mathit{Type} \\ \mathit{Ot3} &= \mathit{Var} \\ \mathit{Ot4} &= \mathit{Type} \times \mathit{Type} \end{aligned}$$

Note the special "obligation", `error`, which denotes that some typing violation has already been discovered. This special token is always propagated once it replaces a normal set of obligations.

The idea behind this definition of *Data* is actually quite straightforward. We stated above that a type environment can be recovered from the stack, but the stack structure also gives us a couple of extra useful properties. It is implicitly used to manage scoping, so that at a confluence point we can recover the common context, and it will also be useful in observing that the analysis is distributive, which will be needed when we talk about staging in the next Section. Another important implication of staging is the observation that we have to be able to delay the discharging of obligations until we have more information about the program. This is the idea behind aggregating the obligations into the second component of *Data*, instead of discharging obligations on-the-fly. When taken all together, these properties of *Data* will allow us to quickly generate *the type* and *the effects* of typing a fixed program given any arbitrary type environment, which is fundamental to efficient staging.

Now that we have all of our preliminary data-types defined, we can move on to the various functions required by our framework. In the definitions that follow we will

$$\begin{array}{ccc} \tau \in \mathit{VarType} & \oplus \in \mathit{BinaryOp} & \Gamma, \gamma \in \mathit{TyEv} \\ \Sigma \in \mathit{TySt} & \Lambda \in \mathit{List} & \Delta \in \mathit{Oblg} \end{array}$$

Figure 3.4.1: Conventions on variable types

use the variable conventions in Figure 3.4.1. In particular, note that Λ will represent an arbitrary list of tuples but is not required to be a type stack. That is, the $\star$ token is not required in the leading tuple and can, in addition, appear anywhere in the list, even multiple times.

In order to simplify the presentation and increase clarity, the definitions of the functions required by the framework will rely on a number of auxiliary functions, which are described next. The auxiliary functions fall into two primary categories: ones that will be used in the manipulation of type stacks and ones that will be used in the manipulation of obligations. We first present the auxiliary functions for type stacks, starting with one that extracts a type environment from a type stack.

Definition 3.4.2. (`stack-to-tyev`)

$$\texttt{stack-to-tyev} : \mathit{List} \rightarrow \mathit{TyEv}$$

Recall that $\cup$ was defined on partial functions in Section 3.1; then the following recursive formulation on the structure of lists defines `stack-to-tyev` completely ($\cdot \in \{\star\} \cup \mathit{Node}$)

$$\begin{array}{rcl} \texttt{stack-to-tyev}(\epsilon) & = & \epsilon \\ \texttt{stack-to-tyev}(\langle \cdot, \gamma \rangle :: \Lambda) & = & \gamma \cup \texttt{stack-to-tyev}(\Lambda) \end{array}$$

When we come to a point of confluence we will need to figure out what the current context or scope is. This is accomplished by taking the longest *common prefix* of the type stacks reaching the confluence point. Soundness of this approach with respect to the system in Figure 3.4.1 can be seen by comparing the inference rules with the possible confluence points as defined in the extended analysis framework.

Each program construct that creates a point of confluence (labeled statements, conditionals, and loops) also creates a new scope, so that the current context is in effect *saved* and can be returned to later. We do not, however, present a full proof of the soundness of our implementation.

Definition 3.4.3. (`longest-common-prefix`)

$$\texttt{longest-common-prefix} : \mathit{List} \to \mathit{List} \to \mathit{List}$$

is defined such that (where $\cdot \in \{\star\} \cup \mathit{Node}$)

$$\texttt{longest-common-prefix}(\langle \cdot, \gamma \rangle :: \Lambda_1', \langle \cdot, \gamma \rangle :: \Lambda_2') = \\ \langle \cdot, \gamma \rangle :: \texttt{longest-common-prefix}(\Lambda_1', \Lambda_2')$$

and otherwise

$$\texttt{longest-common-prefix}(\Lambda_1, \Lambda_2) = \epsilon$$

The final stack oriented auxiliary function that we define merges two type stacks together, corresponding to sequential composition. Essentially the operation folds the top of one stack into the bottom frame of the other.

Definition 3.4.4. (`merge`)

$$\texttt{merge} : \mathit{TySt} \to \mathit{TySt} \to \mathit{TySt}$$

is defined such that

$$\begin{aligned} \texttt{merge}(\langle \star, \gamma_1 \rangle, \langle \star, \gamma_2 \rangle :: \Lambda_2) &= \langle \star, \gamma_1 \cup \gamma_2 \rangle :: \Lambda_2 \\ \texttt{merge}(\Sigma_1 :: \langle n_1, \gamma_1 \rangle, \langle \star, \gamma_2 \rangle :: \Lambda_2) &= \Sigma_1 :: \langle n_1, \gamma_1 \cup \gamma_2 \rangle :: \Lambda_2 \end{aligned}$$

The first auxiliary function that operates on obligations simply combines two sets of obligations together, but makes sure to propagate `error`.

Definition 3.4.5. ($\sqcup$)

$$\sqcup : \mathit{Oblg} \to \mathit{Oblg} \to \mathit{Oblg}$$

is defined such that

$$\Delta_1 \sqcup \Delta_2 = \begin{cases} \texttt{error} & \text{if } \Delta_1 = \texttt{error} \text{ or } \Delta_2 = \texttt{error} \\ \Delta_1 \cup \Delta_2 & \text{otherwise} \end{cases}$$

Expressions are defined inductively but the structure of expressions is not traversed by the analysis function $\mathcal{F}[\![\cdot]\!]$. In the case of type checking, we will need to generate obligations from expressions and so we need a separate function to recurse over the structure of expressions and gather this data. This is exactly what the next auxiliary function does.

Definition 3.4.6. (`check-exp`)

$$\texttt{check-exp} : \mathit{Exp} \to \mathit{Type} \to \mathit{Oblg}$$

is defined inductively on the structure of expressions with *binary* operators, constants, and variables.

$$\begin{aligned} \texttt{check-exp}(c, \tau) &= \begin{cases} \texttt{error} & \text{if } \texttt{type}(c) \neq \tau \\ \emptyset & \text{otherwise} \end{cases} \\ \texttt{check-exp}(x, \tau) &= \{(x, \tau)\} \\ \texttt{check-exp}(e_1 \oplus e_2, \tau) &= \texttt{check-exp}(e_1, \texttt{ltype}(\oplus)) \sqcup \texttt{check-exp}(e_2, \texttt{rtype}(\oplus)) \\ &\sqcup \ (\texttt{type}(\oplus), \tau) \end{aligned}$$

Lastly, for convenience, we define a function that updates a set of obligations given new information in the type environment. That is, the next function is used to discharge obligations as the type environment becomes filled in more completely.

Definition 3.4.7. (`discharge`)

$$\texttt{discharge} : \mathit{TyEv} \to \mathit{Oblg} \to \mathit{Oblg}$$

is defined such that

$$\texttt{discharge}(\Gamma, \Delta) = \texttt{check-oblg}(\Gamma, \texttt{prune}(\Gamma, \Delta))$$

where

$$\texttt{prune} : \mathit{TyEv} \to \mathit{Oblg} \to \mathit{Oblg}$$

is defined so that $\texttt{prune}(\Gamma, \Delta)$ is the set of obligations obtained by substituting $\Gamma(x)$ for any variable in a Type 1 or Type 2 obligation in Δ, and also retaining any Type 3 obligations in Δ. We do not expect any Type 4 obligations in Δ because we define `check-oblg` below to get rid of all of the Type 4 obligations.

$$\texttt{check-oblg} : \mathit{TyEv} \to \mathit{Oblg} \to \mathit{Oblg}$$

discharges any Type 4 obligations created by applying `prune`. So $\texttt{check-oblg}(\Delta)$ is defined such that if either there is some x with $\Gamma(x) \downarrow$ and $x \in \texttt{check-oblg}(\Delta)$ or there exists some $(\tau, \tau') \in \Delta'$ with $\tau \neq \tau'$, then $\texttt{check-oblg}(\Delta') = \texttt{error}$. Otherwise $\texttt{check-oblg}(\Delta') = \Delta' \setminus \wp(\mathit{Ot4})$.

With the auxiliary functions it is easy to define `assign` and `exp`, as well as the the rest of the functions required by the expanded framework of Section 3.3. According to the type system given in Figure 3.4.1, an assignment statement is well-typed exactly when the type of the variable and the type of the expression agree. Therefore

Definition 3.4.8. (`assign`)

$$\texttt{assign} : \mathit{Node} \to \mathit{Var} \to \mathit{Exp} \to \mathit{Data}$$

is defined such that

$$\texttt{assign}(n, x, e) = (\langle \star, \epsilon \rangle, (\texttt{check-exp}(e, \texttt{type}(e)) \sqcup \{(x, \texttt{type}(e))\}))$$

The only time `exp` is invoked in $\mathcal{F}[\![\cdot]\!]$ is for conditional statements or loops, and implicitly we require in both cases that the expression types to `bool`. Therefore, we

simply use `check-exp` to calculate the necessary set of obligations that need to be satisfied.

Definition 3.4.9. (`exp`)

$$\texttt{exp} : \mathit{ExpData}$$

is defined such that

$$\texttt{exp}(e) = (\langle \star, \epsilon \rangle, \texttt{check-exp}(e, \texttt{bool}))$$

Now, for declarations, we need to do two things: we need to update the type stack to note that the variable was declared in this scope and has a specific type, and we need to add a Type 1 obligation to the obligation set to make sure to enforce that this new declaration is not a re-declaration.

Definition 3.4.10. (`decl`)

$$\texttt{decl} : \mathit{Node} \to \mathit{Var} \to \mathit{Type} \to \mathit{Data}$$

is defined such that

$$\texttt{decl}(n, x, \tau) = (\langle (\star, \epsilon[x \mapsto \tau]) \rangle, \{x\})$$

According to the typing system's inference rules, a new scope is created underneath labeled statements, conditionals, and loops. This can be seen in the type system's inference rules by noting the *effect* associated with each of these three constructs does not carry the changes from the construct's sub-programs. When we enter a scope we simply add a new stack frame and use the current node as the frame's label.

Definition 3.4.11. (`es`)

$$\texttt{es} : \mathit{Node} \to \mathit{Data} \to \mathit{Data}$$

is defined such that

$$\texttt{es}(n, (\Sigma, \Delta)) = (\Sigma :: \langle (n, \epsilon) \rangle, \Delta)$$

And when we leave a scope we simply pop off the appropriate frame, the one indicated by the *Node* argument to `ls`.

Definition 3.4.12. (`ls`)

$$\texttt{ls} : \mathit{Node} \to \mathit{Data} \to \mathit{Data}$$

is defined such that

$$\texttt{ls}\,(n, (\Sigma :: \langle (n, \gamma) \rangle :: \Lambda, \Delta)) = (\Sigma, \Delta)$$

We only have the semi-colon and confluence operators left to define. The semi-colon merges the two stacks, and discharges any obligations that it can from the second obligation set by using the new information from the type stack coming from the left.

Definition 3.4.13.

$$; : \mathit{Data} \to \mathit{Data} \to \mathit{Data}$$

is defined such that

$$(\Sigma_1, \Delta_1); (\Sigma_2, \Delta_2) = (\texttt{merge}(\Sigma_1, \Sigma_2), \Delta_1 \sqcup (\texttt{discharge}(\texttt{stack-to-tyev}(\Sigma_1), \Delta_2)))$$

The confluence operator has to find the common context/scope to jump back out to and then unions together the various obligations coming from the different paths.

Definition 3.4.14.

$$\wedge : \mathit{Data} \to \mathit{Data} \to \mathit{Data}$$

is defined such that

$$(\Sigma_1, \Delta_1) \wedge (\Sigma_2, \Delta_2) = (\texttt{longest-common-prefix}(\Sigma_1, \Sigma_2), \Delta_1 \sqcup \Delta_2)$$

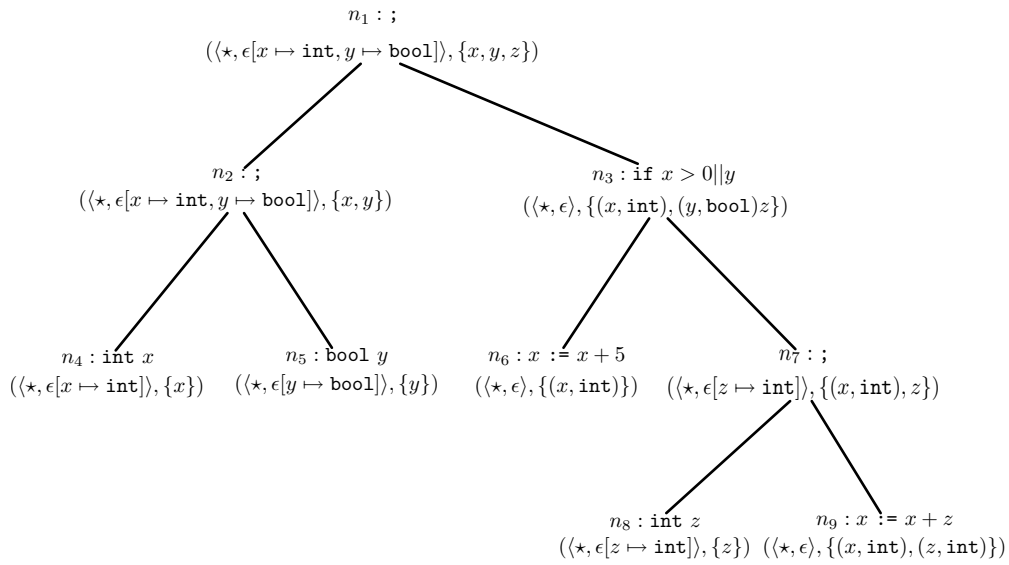


Figure 3.4.1: Type checking example.

An example of the type checking analysis is presented in Figure 3.4.1. It is annotated in a manner similar to our two previous examples. Each node gets a value from the computation

$$\mathcal{F}[\![n : P]\!]\ \epsilon\ \epsilon\ (\langle\star, \epsilon\rangle, \emptyset)$$

This program is well-typed because in the summarization data associated with node n_1, which is the topmost node, there are only Type 3 obligations.

The only thing left to discuss in this section is the distributivity of ; over the $\wedge$ operator. As we have indicated previously, distributivity is a condition assumed throughout this Thesis and is integral to our staging algorithm. We prove distributivity by first focusing on the type stacks.

Theorem 3.4.1. *The type checking analysis is distributive, that is,*

$$(\Sigma_1, \Delta_1); ((\Sigma_2, \Delta_2) \wedge (\Sigma_3, \Delta_3)) = (\Sigma_1, \Delta_1); (\Sigma_2, \Delta_2) \wedge (\Sigma_1, \Delta_1); (\Sigma_3, \Delta_3)$$

Proof. Each side of the equation reduces to a *Data* value and thus to a pair (Σ, Δ).

Therefore, to prove the equivalence we have to show that for each of the two components, both sides of the equation correspond. For the Σ part we can expand the definitions to reduce the problem to showing that

$$\texttt{merge}(\Sigma_1, \texttt{longest-common-prefix}(\Sigma_2, \Sigma_3)) \\ = \texttt{longest-common-prefix}(\texttt{merge}(\Sigma_1, \Sigma_2), \texttt{merge}(\Sigma_1, \Sigma_3))$$

which follows easily from the definitions. For the Δ part we have to show (we have removed the Δ_1 from both sides)

$$\texttt{discharge}(\texttt{stack-to-tyev}(\Sigma_1), \Delta_2 \sqcup \Delta_3) \\ = \texttt{discharge}(\texttt{stack-to-tyev}(\Sigma_1), \Delta_2) \sqcup \texttt{discharge}(\texttt{stack-to-tyev}(\Sigma_1), \Delta_3)$$

Let $\Gamma = \texttt{stack-to-tyev}(\Sigma_1)$ and let $o \in \texttt{discharge}(\Gamma, \Delta_2 \sqcup \Delta_3)$. Then by definition o is either a Type 3 or Type 4 obligation in either Δ_2 or Δ_3 or it is the result of replacing the variables with types from Γ of a Type 1 or Type 2 obligation from one of the two sets. In either case o is clearly also in at least one of the two sets unioned together on the right side of the equation. The other direction and handling the `error` case are similarly easy. □

3.5 A Normal Form Theorem

The primary goal of this chapter is to develop a general algorithm for the staged computation of data flow analyses. The practical application of such an algorithm is the ability to reduce the amount of computation done at run time, which means that the analyses can more easily be applied for the purpose of optimization. Thus far we have developed a mathematical framework for calculating this information in a non-staged way, and so in this Section we begin with the work required for staging, developing the major theoretical result that our staging mechanism relies on. The next Section turns these mathematical functions and proofs into a usable algorithm.

The analysis framework defines a function that is recursive over the structure of

programs, and therefore one natural approach to staging is to precompute some of the recursive calls and then substitute in the more efficient, precomputed function as needed at run time. This is the general approach that we adopt. The difficult part with implementing this idea directly is that in most imperative programming languages there is no obvious way to create an efficient data structure for representing and evaluating functions or functional constructs. On the other hand, if we look at the three example analyses given thus far we can see that *Data* is easily represented using common data structures, and that ; and $\wedge$ are easily coded procedurally. For example, in the variable initialization example *Data* is just a pair of sets. Taking this idea and running with it, we will show how to use the set *Data*, and operators on its elements, to represent the necessary function space *symbolically*.

The theorem we eventually prove states that in place of $\mathcal{F}[\![\cdot]\!]$ we can derive a *summarization* pair (φ, η) that is in some sense a canonical representation of the function and can be used to efficiently compute the function with any arguments. The basic reason for the existence of this summarization pair is that when $\mathcal{F}[\![n : P]\!]$ is evaluated with some specific arguments, the only values that change are the ones for nodes or labels occuring in $n : P$. Of course the values computed for nodes in $n : P$ are not entirely independent of the arguments, they depend on both the second and third arguments which communicate all of the relevant data from outside $n : P$. Moreover, because our proof is constructive, the theorem actually describes a method of obtaining a summarization pair; and in that sense this witness is *normal*. We therefore refer to this theorem throughout the rest of the Thesis as the *normal form theorem*.

For the sake of clarity throughout our proofs, we define some commonly used sets of nodes and labels. These definitions usually come in pairs such that one set is a set of nodes (superscript n) and the other is some corresponding set of labels (superscript l). These superscripts should not be confused for parameters. Our first definintion simply enumerates all nodes (or labels) in a given progam.

Definition 3.5.1. Let $n : P \in \mathit{Prgm}$. We define the set of nodes *occuring* in $n : P$

$$N_P \subseteq \mathit{Node}$$

to be such that for all $n' \in \mathit{Node}$, if we let $(\varphi, \eta, d) = \mathcal{F}[\![n : P]\!]\ \epsilon\ \epsilon\ I$, then $n' \in N_P$ exactly when $\varphi(n) \downarrow$. We denote the corresponding set of *labels* occuring in $n : P$ by

$$L_P \subseteq \mathit{Label}$$

The second pair of sets that we define have to do with break statements which are not properly nested within an appropriate labeled statement. This can easily happen because we are dealing with program fragments which in most cases are not generated directly but are used as pieces to be assembled at run time. Thus, such a fragment should always be fitted into some larger context at run time and that context should contain the appropriate labeled statement. Break statements that are not properly nested within a labeled statement, and thus have no well defined target, are referred to as *dangling*.

Definition 3.5.2. Let $n : P \in \mathit{Prgm}$. We define the set of *dangling breaks* occurring in $n : P$

$$DBN_P \subseteq N_P$$

to be such that for all $n' \in \mathit{Node}$ with $n' : \texttt{break}\ l$, for some l, a subprogram of $n : P$; if we let $(\varphi, \eta, d) = \mathcal{F}[\![n : P]\!]\ \epsilon\ \epsilon\ I$, then $n' \in DBN_P$ exactly when $\varphi(n') \downarrow$ and $\eta(l) \downarrow$. The set of corresponding labels is denoted

$$DBL_P \subseteq L_P$$

Before proceeding we must impose some basic constraints on the framework's parameterizing operators in order for our proofs to go through. For the semi-colon and confluence operators we have already stated that the distributivity property will be sufficient. The definition that we now give is for a sufficient condition on the `es` and `ls` operators which are part of the expanded framework. The definition says that applying `ls` to a data element preserves all of the information up to where the corresponding `es` was applied.

Definition 3.5.3. We say that an analysis is *scope sensible* if for every $n \in \mathit{Node}$

and $d, t \in \mathit{Data}$ there is a $t' \in \mathit{Data}$ such that

$$\mathtt{ls}(n, \mathtt{es}(n, d); t) = d; t'$$

In addition, we require that $\mathtt{es}(n, d) = d; t$ for some $t \in \mathit{Data}$.

It is easy to see that the type checking analysis given in Section 3.4 is *essentially* scope sensible and that in fact as long as Δ_2 does not have a frame labeled with node n the following equation holds:

$$\mathtt{ls}(n, \mathtt{es}(n, (\Sigma_1, \Delta_1)); (\Sigma_2, \Delta_2)) = (\Sigma_1, \Delta_1); (\langle \star, \emptyset \rangle, \Delta_2)$$

The way to understand this equation is to think of the type stack being popped just past the scope started by node n, but the obligations being retained. Which should make sense intuitively. It will be fine that the type checking example is not completely scope sensible because we can prove that the above counterexample will never occur when using the framework, therefore both in theory and practice we can rely on scope sensibility with type checking.

We now move on to the jobs of proving the existence of a summarization pair. This first Lemma states that the d and $\varphi(n)$ part of $\mathcal{F}[\![n : P]\!]$ always coincide (except on dangling breaks).

Lemma 3.5.1. *For any $n : P$, φ, η and d, if we let*

$$(\varphi', \eta', d') = \mathcal{F}[\![n : P]\!]\ \varphi\ \eta\ d$$

then $\varphi'(n) = d'$ or $d' = \mathtt{nil}$.

Proof. Immediate from the definition of $\mathcal{F}[\![\cdot]\!]$. □

A second auxiliary Lemma that we will need states that the result obtained from calculating $\mathcal{F}[\![n : P]\!]\ \varphi\ \eta\ d$ has data and break environments unchanged from φ and η on nodes and labels not occuring in $n : P$.

Lemma 3.5.2. *Let $n : P \in Prgm$, and take any φ, η, and d. If we let*

$$(\varphi', \eta', d') = \mathcal{F}[\![n : P]\!]\ \varphi\ \eta\ d$$

then

- *For all $n_x \notin N_P$, $\varphi'(n_x) = \varphi(n_x)$.*
- *For all $l_x \notin L_P$, $\eta'(l_x) = \eta(l_x)$.*

Proof. The proof is by structural induction on programs.

Base Case:

- Consider when $n : P = n : \texttt{skip}$.

  $$\begin{aligned}(\varphi', \eta', d') = \\ (\varphi[n \mapsto d], \eta, d)\end{aligned}$$

  which is exactly as required by the theorem statement since $N_P = \{n\}$ and $L_P = \emptyset$.

- Consider when $n : P = n : \tau\ x$.

  $$\begin{aligned}(\varphi', \eta', d') = \\ \text{let } d' := d; \texttt{decl}(n, \tau, x) \\ \text{in } \ (\varphi[n \mapsto d'], \eta, d')\end{aligned}$$

  as required.

- Consider when $n : P = n : \texttt{break}\ l$.

  $$\begin{aligned}(\varphi', \eta', d') = \\ (\varphi[n \mapsto d \wedge \eta(l)], \eta[l \mapsto d \wedge \eta(l)], \texttt{nil})\end{aligned}$$

  as required, because $N_P = \{n\}$ and $L_P = \{l\}$.

- Consider when $n : P = n : x \ := \ e$.

$$
\begin{array}{l}
(\varphi', \eta', d') = \\
\quad \text{let } d' := d; \texttt{assign}(n, x, e) \\
\quad \text{in } \ (\varphi[n \mapsto d'], \eta, d')
\end{array}
$$

  as required.

Induction Step:

- Consider when $n : P = n : l \ : \ n_1 : P_1$.

$$
\begin{array}{l}
(\varphi', \eta', d') = \\
\quad \text{let } (\varphi_1, \eta_1, d_1) := \mathcal{F}[\![n_1 : P_1]\!] \ \varphi \ (\eta \setminus \{l\}) \ \texttt{es}(n_1, d) \\
\quad \text{in } \ (\varphi_1[n \mapsto \texttt{ls}(n_1, d_1) \wedge \texttt{ls}(n_1, \eta_1(l))], \eta_1 \setminus \{l\}, \texttt{ls}(n_1, d_1) \wedge \texttt{ls}(n_1, \eta_1(l))
\end{array}
$$

  From the induction hypothesis we get

$$
\begin{array}{rcl}
\varphi_1 \restriction_{Node \setminus N_{P_1}} & = & \varphi \restriction_{Node \setminus N_{P_1}} \\
\eta_1 \restriction_{Label \setminus L_{P_1}} & = & (\eta \setminus \{l\}) \restriction_{Label \setminus L_{P_1}}
\end{array}
$$

  as required because $N_P = N_{P_1} \cup \{n\}$ and $L_P = L_{P_1} \cup \{l\}$.

- Consider when $n : P = n : (n_1 : P_1)\,;(n_2 : P_2)$.

$$
\begin{array}{l}
(\varphi', \eta', d') = \\
\quad \text{let } (\varphi_1, \eta_1, d_1) := \mathcal{F}[\![n_1 : P_1]\!] \ \varphi \ \eta \ d \\
\quad \text{in } \ \text{let } \varphi_2, \eta_2, d_2) := \mathcal{F}[\![n_2 : P_2]\!] \ \varphi_1 \ \eta_1 \ d_1 \\
\qquad \text{in } \ (\varphi_2[n \mapsto d_2], \eta_2, d_2)
\end{array}
$$

From the induction hypothesis we get

$$\begin{aligned}
\varphi_1 \restriction_{Node \setminus N_{P_1}} &= \varphi \restriction_{Node \setminus N_{P_1}} \\
\eta_1 \restriction_{Label \setminus L_{P_1}} &= \eta \restriction_{Label \setminus L_{P_1}} \\
\varphi_2 \restriction_{Node \setminus N_{P_2}} &= \varphi_1 \restriction_{Node \setminus N_{P_2}} \\
\eta_2 \restriction_{Label \setminus L_{P_2}} &= \eta_1 \restriction_{Label \setminus L_{P_2}}
\end{aligned}$$

which imply

$$\begin{aligned}
\varphi' \restriction_{Node \setminus (N_{P_1} \cup N_{P_2} \cup \{n\})} &= \varphi \restriction_{Node \setminus (N_{P_1} \cup N_{P_2} \cup \{n\})} \\
\eta' \restriction_{Label \setminus (L_{P_1} \cup L_{P_2})} &= \eta \restriction_{Label \setminus (L_{P_1} \cup L_{P_2})}
\end{aligned}$$

as required.

- Consider when $n : P = n : \texttt{if}\ e\ \texttt{then}\ n_1 : P_1\ \texttt{else}\ n_2 : P_2$.

$$\begin{aligned}
&(\varphi', \eta', d') = \\
&\quad \text{let } (\varphi_1, \eta_1, d_1) := \mathcal{F}[\![n_1 : P_1]\!]\ \varphi\ \eta\ \texttt{es}(n_1, d; \texttt{exp}(e)) \\
&\qquad (\varphi_2, \eta_2, d_2) := \mathcal{F}[\![n_2 : P_2]\!]\ \varphi\ \eta\ \texttt{es}(n_2, d; \texttt{exp}(e)) \\
&\quad \text{in } ((\varphi_1 \cup \varphi_2)[n \mapsto \texttt{ls}(n_1, d_1) \wedge \texttt{ls}(n_2, d_2)], \eta_1 \wedge \eta_2, \texttt{ls}(n_1, d_1) \wedge \texttt{ls}(n_2, d_2))
\end{aligned}$$

From the induction hypothesis we get

$$\begin{aligned}
\varphi_1 \restriction_{Node \setminus N_{P_1}} &= \varphi \restriction_{Node \setminus N_{P_1}} \\
\eta_1 \restriction_{Label \setminus L_{P_1}} &= \eta \restriction_{Label \setminus L_{P_1}} \\
\varphi_2 \restriction_{Node \setminus N_{P_2}} &= \varphi \restriction_{Node \setminus N_{P_2}} \\
\eta_2 \restriction_{Label \setminus L_{P_2}} &= \eta \restriction_{Label \setminus L_{P_2}}
\end{aligned}$$

which implies that

$$\begin{aligned}
\varphi' \restriction_{Node \setminus (N_{P_1} \cup N_{P_2} \cup \{n\})} &= \varphi \restriction_{Node \setminus (N_{P_1} \cup N_{P_2} \cup \{n\})} \\
\eta' \restriction_{Label \setminus (L_{P_1} \cup L_{P_2})} &= \eta \restriction_{Label \setminus (L_{P_1} \cup L_{P_2})}
\end{aligned}$$

as required.

- Consider when $n : P = n : \texttt{do}\ n_1 : P_1\ \texttt{while}\ e$.

  $(\varphi', \eta', d') =$
  let $(\varphi_1, \eta_1, d_1) := \mathcal{F}[\![n_1 : P_1]\!]\ \varphi\ \eta\ \texttt{es}(n_1, d)$
  in let $(\varphi'_1, \eta'_1, d'_1) := ifp\{\mathcal{F}[\![n_1 : P_1]\!]\ \varphi\ \eta\ \texttt{es}(n_1, \texttt{ls}(n_1, d_x); \texttt{exp}(e) \wedge d)\}\{d_1\}$
  in $(\varphi'_1[n \mapsto \texttt{ls}(n_1, d'_1); \texttt{exp}(e)], \eta'_1, \texttt{ls}(n_1, d'_1); \texttt{exp}(e))$

  By the induction hypothesis we get

  $$\begin{aligned} \varphi_1 \restriction_{Node \setminus N_{P_1}} &= \varphi \restriction_{Node \setminus N_{P_1}} \\ \eta_1 \restriction_{Label \setminus L_{P_1}} &= \eta \restriction_{Label \setminus L_{P_1}} \end{aligned}$$

  and then, by a second induction and the fact that the iterated fixed point is guaranteed to reach a steady state after some maximum number of iterations based only on the structure of $n : P$, we easily get the same thing for φ'_1 and η'_1. That is,

  $$\begin{aligned} \varphi'_1 \restriction_{Node \setminus N_{P_1}} &= \varphi \restriction_{Node \setminus N_{P_1}} \\ \eta'_1 \restriction_{Label \setminus L_{P_1}} &= \eta \restriction_{Label \setminus L_{P_1}} \end{aligned}$$

  Therefore, the result holds.

□

We finally come to the normal form theorem itself. The idea of the theorem statement was loosely described above and is formalized exactly next. Some further intuition is probably useful, however, before giving the formal statement. When applied to a program, $\mathcal{F}[\![\cdot]\!]$ returns a function that takes a data environment, break environment, and data element to a triple of the same three things. The data environment is really just a *partial map* from nodes to data elements, and similarly with the break environment, but instead with labels mapping to data elements. The

normal form theorem states that there is a canonical pair of maps (φ', η') which we are calling the summarization pair. From (φ', η') we can derive, given any φ, η and d, $\mathcal{F}[\![\cdot]\!]\ \varphi\ \eta\ d$. The derivation requires at most one semi-colon and one $\wedge$ operation to the each of the data elements mapped by the summarization pair.

Theorem 3.5.1. *Let $n : P$ be a program and assume that ; distributes over $\wedge$ and also that* `es` *and* `ls` *are sensible. Then*

- *For every $n_x \in N_P$, there exists a $t \in Data$ such that for all φ, η, d; if we let*

$$(\varphi', \eta', d') = \mathcal{F}[\![n : P]\!]\ \varphi\ \eta\ d$$

  then $\varphi'(n_x) = d; t$

- *For every $l_x \in DBL_P$, there exists a $t \in Data$ such that for all φ, η, d; if we let*

$$(\varphi', \eta', d') = \mathcal{F}[\![n : P]\!]\ \varphi\ \eta\ d$$

  then $\eta'(l_x) = d; t \wedge \eta(l)$

Proof. The proof is by structural induction on programs.

Base Case:

- Consider when $n : P = n :$ `skip`. Thus $N_P = \{n\}$ and $DBL_P = \emptyset$ and we check the first condition for n and the second condition is satisfied vacuously. By definition of $\mathcal{F}[\![n : P]\!]$ we get $\varphi'(n) = d$, so $t = I$ works.

- Consider when $n : P = n : \tau\ x$. Thus $N_P = \{n\}$ and $DBL_P = \emptyset$. $\varphi'(n) = d; \tau - \texttt{decl}(n, x)$ and we use $t = \tau - \texttt{decl}(n, x)$.

- Consider when $n : P = n : x := e$. Thus $N_P = \{n\}$ and $DBL_P = \emptyset$. $\varphi'(n) = d; \texttt{assign}(n, x, e)$ and we use $t = \texttt{assign}(n, x, e)$.

- Consider when $n : P = n :$ `break` l. Thus $N_P = \{n\}$ and $DBL_P = \{l\}$ and so this time we must check the second condition for label l. $\varphi'(n) = d = d; I$ and $\eta'(l) = d \wedge \eta(l) = d; I \wedge \eta(l)$.

Induction Step:

- Consider when $n : P = n : l \ : \ n_1 : P_1$. Let

$$(\varphi_1, \eta_1, d_1) = \mathcal{F}[\![n_1 : P_1]\!] \ \varphi \ (\eta \setminus \{l\}) \ \mathtt{es}(n, d)$$

  It's clear that $N_P = \{n\} \cup N_{P_1}$ and $DBL_P = DBL_{P_1} \setminus \{l\}$. For $n_x \in N_{P_1}$, $\varphi'(n_x) = \varphi_1(n_x)$ and we can apply the induction hypothesis and the sensibility of `es` to get the result. By Lemma 3.5.1

$$\varphi'(n) = \mathtt{ls}(n, d_1) \wedge \mathtt{ls}(n, \eta_1(l)) = \mathtt{ls}(n, \varphi_1(n_1)) \wedge \mathtt{ls}(n, \eta_1(l))$$

  Applying the induction hypothesis we get

$$\begin{aligned} \varphi'(n) &= \mathtt{ls}(n, \varphi_1(n_1)) \wedge \mathtt{ls}(n, \eta_1(l)) \\ &= \mathtt{ls}(n, \mathtt{es}(n, d); t) \wedge \mathtt{ls}(n, \mathtt{es}(n, d); t' \wedge (\eta(l) \setminus \{l\})) \\ &= \mathtt{ls}(n, \mathtt{es}(n, d); t) \wedge \mathtt{ls}(n, \mathtt{es}(n, d); t') \end{aligned}$$

  To this we just apply sensibility and distributivity to get the desired result. As to the second condition, note that $\eta' = \eta_1 \setminus \{l\}$ and so the result holds by the induction hypothesis and sensibility.

- Consider when $n : P = n : (n_1 : P_1)\,;(n_2 : P_2)$. Let

$$\begin{aligned} (\varphi_1, \eta_1, d_1) &= \mathcal{F}[\![n_1 : P_1]\!]\varphi \ \eta \ d \\ (\varphi_2, \eta_2, d_2) &= \mathcal{F}[\![n_2 : P_2]\!]\varphi_1 \ \eta_1 \ d_1 \end{aligned}$$

  We can see that $N_P = \{n\} \cup N_{P_1} \cup N_{P_2}$ and $DBL_P = DBL_{P_1} \cup B^l_{P_2}$, and from Lemma 3.5.2 we get $\varphi' \restriction_{N_{P_1}} = \varphi_1 \restriction_{N_{P_1}}$, the rest of the first condition follows by the induction hypothesis in addition to Lemma 3.5.1 for $\varphi'(n)$ specifically. Each $l_x \in DBL_P$ falls into one of three categories, either it is from exactly one of the subprograms or occurs as a dangling break in both. Each of the cases follow easily.

- Consider when $n : P = \texttt{if}\ e\ \texttt{then}\ n_1 : P_1\ \texttt{else}\ n_2 : P_2$. Let

$$\begin{aligned}(\varphi_1, \eta_1, d_1) &= \mathcal{F}[\![n_1 : P_1]\!]\varphi\ \eta\ \texttt{es}(n, d; \texttt{exp}(e)) \\ (\varphi_2, \eta_2, d_2) &= \mathcal{F}[\![n_2 : P_2]\!]\varphi\ \eta\ \texttt{es}(n, d; \texttt{exp}(e))\end{aligned}$$

  We can see that $L_P = \{n\} \cup L_{P_1} \cup L_{P_2}$ and $DBL_P = DBL_{P_1} \cup DBL_{P_2}$. Moreover, from the induction hypothesis and Lemma 3.5.1

$$\begin{aligned}\varphi'(n) = \texttt{ls}(n, d_1) \wedge \texttt{ls}(n, d_2) \\ = \texttt{ls}(n, \texttt{es}(n, d; \texttt{exp}(e)); t) \wedge \texttt{ls}(n, \texttt{es}(n, d; \texttt{exp}(e)); t')\end{aligned}$$

  and then the result follows by first applying the sensibility of `es` and `ls` and then distributivity of ; over $\wedge$. The result for $n_x \in N_{P_1}$ or $n_x \in N_{P_2}$ is easily derived from the induction hypothesis and sensibility. We get the second condition similarly, except we also apply distributivity when $l_x \in DBL_{P_1} \cap DBL_{P_2}$.

- Consider when $n : P = n : \texttt{do}\ n_1 : P_1\ \texttt{while}\ e$. Let

$$\begin{aligned}(\varphi_1, \eta_1, d_1) &= \mathcal{F}[\![n_1 : P_1]\!]\ \varphi\ \eta\ \texttt{es}(n_1, d) \\ (\varphi'_1, \eta'_1, d'_1) &= ifp\{\mathcal{F}[\![n_1 : P_1]\!]\ \varphi\ \eta\ \texttt{es}(n_1, \texttt{ls}(n_1, d_x); \texttt{exp}(e) \wedge d)\}\{d_1\}\end{aligned}$$

  It is clear that $N_P = \{n\} \cup N_{P_1}$ and $DBL_P = DBL_{P_1}$. Now in order to prove this case we have to use the fact that all of our analyses reach a fixed point after a fixed number of iterations dependent only on the structure of $n : P$. Therefore, we proceed by a second induction on this value. In the base case the fixed point converges after one iteration so that

$$(\varphi_1, \eta_1, d_1) = \mathcal{F}[\![n_1 : P_1]\!]\ \varphi\ \eta\ \texttt{es}(n_1, \texttt{ls}(n_1, d_1); \texttt{exp}(e) \wedge d)$$

  By sensibility we get $\texttt{ls}(n_1, d_1) = d; t$ for some $t \in \mathit{Data}$. Applying sensibility again and also the second induction hypothesis, we get the desired result. The induction step follows similarly. At every iteration $d_x = d'_1$, which by the

second induction hypothesis is of the form $\texttt{es}(n_1, d; t)$ for some $t \in \mathit{Data}$. We then apply sensibility and the second induction hypothesis to get the result.

□

3.6 Handling Staging Efficiently

An RTPG compiler gives the programmer the ability to write programs which at run time dynamically assemble, compile, and run new programs. Jumbo, specifically, extends Java with a built-in class called *Code*, and adds syntax and operations for manipulating these code objects. As described in Chapter 1, the basic assembly process fits together statically defined code fragments by means of typed place holders or variables, called *holes*. When a code fragment is used to fill in a hole, it is called a *plug*, and *Code* objects with no holes are called *whole*. Although Jumbo admits a more general mechanism of program assembly, the staging algorithm presented in this section works only on code fragments assembled together with plugs that are whole so that only one level of holes is considered.

When a whole program is eventually generated we have to consider some trade-offs, in particular the trade-off between the amount of time we spend on optimization and the overall benefit we can expect by doing so. If we generate a program that is run 1000 times and consumes 50% of the total program run time, then it is obvious that we may want to optimize more heavily. If on the other hand a generated program is run only once and takes a negligible amount of the entire run time, then we will just slow down the entire execution by optimizing it. We do not consider the problem of determining which whole programs are the best candidates for heavy optimization; we are simply trying to reduce the cost of performing the optimization, which has two important effects. The optimizers we are talking about are invoked at run time, so reducing the amount of time it takes for them to run affects overall program run time directly. Secondly, reducing the cost of optimization allows you to do more optimization for the same pre-reduction cost, which can potentially have a very dramatic effect on run time if you spend your extra budget optimizing the most

frequently and longest running programs.

We use our framework and the normal form theorem developed in the previous section as the basis for implementing a general algorithm for staged data flow analysis. Because we are now moving into the algorithmic domain from a purely mathematical one, we need to make clear what the corresponding algorithmic interpretations are for things like *Data*and *BreakEnv*̇There are four data types that our algorithm works with:

- `code`: This is simply a new name for the *Code* objects described above. More generically these are the program fragments manipulated by any given RTPG compiler and run time.

- `map`: An associative map data type. The constant "empty" represents the empty map and is the algorithmic counterpart to ϵ.

- `data`: This is the data type corresponding to *Data* and is dependent on the particular analysis being computed. We showed previously that *Data* usually admits a natural representation using common data structures like ones available in Java. We assume constants "I" and "`nil`" corresponding to the ones from the framework.

- `label`: This data type is needed because the algorithmic interpretation of the break environment maps things of type `label` to things of type `data`.

Partial functions that are used by the framework become instances of type `map` in the algorithmic interpretation. In addition, we have the procedures (we use *procedure* to distinguish from mathematical functions)

- ;(`data`, `data`): The semi-colon procedure.

- $\wedge$(`data`, `data`): The confluence procedure.

which in addition to `data` must of course also be coded separately for each analysis. Lastly, we define a recursive procedure that acts as the algorithmic interpretation of our framework and computes the result of applying the framework (in the form

of `map` and `data`). It should be clear that writing such a procedure, using the data types and functions described, would be very easy. Therefore we also have

- $f(\texttt{code}, \texttt{map}, \texttt{map}, \texttt{data})$: This is the algorithmic interpretation of $\mathcal{F}[\![\cdot]\!]$. Of course, whereas $\mathcal{F}[\![n : P]\!]$ is just another function, we only define the procedure f when all of the arguments are specified.

The general algorithm then, which we have implemented and integrated into Jumbo, works in two stages. At compile time we find all of the whole program fragments and generate the summarization pairs. At run time when a whole program has been assembled and we want to calculate $\mathcal{F}[\![n : P]\!]\ \varphi\ \eta\ d$ we start by applying the recursive definition of $\mathcal{F}[\![\cdot]\!]$ directly, that is, by calling the procedure $f(\cdot, \cdot, \cdot, \cdot)$. If we recursively get to a node which was summarized at compile time, we read the summarization pair, and update the data and break environments from them, exactly as required by the normal form theorem. The procedure which applies the normal form theorem using the summarization pairs from the offline phase is denoted f_s and presented in Figure 3.6.1. The two phased algorithm is given as pseudo-code in Figure 3.6.2.

Benchmarking results for our reaching definitions and type checking analyses are presented in Table 3.6. The first part of the table shows experiments with contrived examples: a small program with a large plug, a large program with a small plug, and a medium sized program with two medium sized plugs. The second of the table has results from runs with "real" examples from other projects that incorporated Jumbo, and are appropriately cited. The numbers presented are the ratio of the time it took to run $\mathcal{F}[\![\cdot]\!]$ recursively without any summarized nodes and the time to run it with summarization pairs for each plug. So a value of 2 means that the our staging mechanism would be twice as fast at run time. We ran each experiment in three different Java runtime environments, including GNU libgcj which runs native code rather than going through a JIT compiler at runtime. The analyses looked at are reaching definitions (R.D.) and type checking (T.C.).

The clear trend of the data is that the larger the holes the better, but even with small holes the speedup can be very significant. We have not analyzed carefully the

```
 1: /* f modified to use summarization pairs */
 2:
 3: procedure f_s(p : code, de : map, be : map, d : data, sp : map)
 4: if sp[p] is a valid mapping then
 5:
 6:    /* use normal form theorem result to update de */
 7:
 8:    (de', be', d') := sp[p];
 9:    for all p' mapped by de do
10:       de[p'] := d; de'[p']);
11:    end for
12:    for all l mapped by be do
13:       be[l] := be[l] ∧ (d; be'[l]);
14:    end for
15: else
16:
17:    /* do f except with recursive calls to f replaced by calls to f_s */
18:
19: end if
```

Figure 3.6.1: Framework procedure which utilizes summarization pairs.

anomalies in the type checking benchmarks or attempted a more detailed analysis of the performance results seen. Indeed, we do not claim that the numbers presented are indicative of the speedups (if any) that could be seen in the real-world, and in particular we should point out that our base analysis is not optimized. So for example we do not have any numbers to compare our staged analysis versus the best reaching definitions analysis in existence, which is probably much faster than our recursive approach. However the results do show good potential for speedup and indicate that it is a worthwhile avenue to approach optimization.

```
 1: /* offline phase */
 2:
 3: declare plugs := ...; /* static set of whole code fragments */
 4: declare sp := ...; /* map for summarization pairs */
 5: for all p ∈ plugs do
 6:    sp[p] := f(p, I);
 7: end for
 8:
 9: /* online phase */
10:
11: declare wp := ...; /* whole programs assembled at run time */
12: declare result := ...; /* map for holding result data */
13: for all p ∈ wp do
14:    result := f(p, empty, empty, I, sp); /* overloaded f defined below */
15: end for
```

Figure 3.6.2: Generic algorithm for staged data flow.

	HotSpot		libgcj		Kaffe	
	R.D.	T.C.	R.D.	T.C.	R.D.	T.C.
Contrived Examples						
Large Plug	2.10	3.65	7.43	5.15	9.73	5.63
Small Plug	2.17	3.50	6.96	4.28	10.70	5.55
Two Medium Plugs	1.67	1.66	2.59	2.90	3.83	3.18
Real World Examples						
Fibonacci-1 [11]	1.10	1.31	1.24	1.17	1.64	1.05
Fibonacci-2 [11]	1.23	0.67	1.48	1.18	2.02	1.05
Sort [13]	1.48	1.92	1.64	1.59	1.86	1.66
Huffman [11]	1.11	0.30	1.04	1.02	1.31	0.95
Marshalling-1 [3]	12.37	28.27	34.83	9.34	49.64	12.04
Marshalling-2 [3]	2.01	16.01	1.83	1.86	2.59	1.47

Table 3.1: Benchmarking results.

Chapter 4

Soundness of the Data Flow Analysis Framework

In the previous Chapter we consistently used analogies with the classical CFG based framework in order to explain the design choices we made for our own framework. In this Chapter we rigorously establish the correctness and robustness of our framework relative to the classical one. In this sense we claim that our framework is *sound*.

Section 4.1 frames. In Section 4.2 we prove some useful lemmas that lead up to the proof of our main result in Section 4.3.

4.1 Overview

The goal of this Chapter is to prove that the two frameworks, ours and the classical one, are functionally equivalent. We only treat our *original* framework without the extensions from Section 3.4. Of course, although our framework and the classical one are similar in many respects, they also have significant differences that force us to take care as we formalize the equivalence exactly. The most notable difference between the two is that they work with different program representations, and so the first part of this Section deals with the problem of converting program source into an equivalent control flow graph. After we finish with that, we explain the details of how the classical algorithm is applied, and then formalize the soundness theorem.

The first step is to define exactly what the structure of a CFG will be in our case. In the definition that we give next we use two auxiliary data sets that we now define.

$$Node^{+} = Node \cup \{\texttt{nowhere}\}$$

is the set of program nodes with an extra token, `nowhere`. This is a new token is used as a pointer to nowhere when there is no normal exit due to dangling breaks. The second auxiliary data type we need is a restricted set of programs for annotating the graph with basic block information. The set is defined by the grammar given in Figure 4.1. Note that unlike the set of programs, the set of *basic block programs* allows expressions as legitimate values.

Definition 4.1.1 (*CFG*). The set of *control flow graphs* is defined such that

$$CFG = Node \times Node^{+} \times \wp(Node^{+} \times Node^{+}) \times (Node \rightharpoonup BBPrgm)$$

The definition deserves some further explanation: the first two coordinates identify the unique entry and normal exit points for the program being represented. The third coordinate is the graph's edge relation (the control flow), and that the fourth coordinate annotates nodes with basic block information. Every CFG has an entry point, but in some cases will not have a *normal* exit. This case is marked with the `nowhere` token, and happens when all of the execution paths have dangling breaks.

In the previous Chapter we marked unreachable nodes by setting their value in the data environment to `nil`. In the next few definitions we formalize this idea so that it can be used in the definition of our conversion function given afterward. Note that these definitions are well formed even though we do not select a specific *analysis* in any of the cases.

Definition 4.1.2. Let $n : P \in Prgm$ and $(\varphi, \eta, d) = \mathcal{F}[\![n : P]\!]\ \epsilon\ \epsilon\ I$. We say that a node n' is *reachable* in $n : P$ when $\varphi(n') \downarrow$ and $\varphi(n') \neq nil$.

Definition 4.1.3. Let $n : P \in Prgm$. The set of *reachable nodes* in $n : P$

$$RN_P \subseteq N_P$$

is defined to be the set of all nodes reachable in $n : P$.

Definition 4.1.4. Let $n : P \in Prgm$. The set of *reachable dangling breaks* in $n : P$

$$RDBN_P \subseteq DBN_P$$

$$
\begin{array}{rcl}
\mathit{BBPrgm} & := & \texttt{skip} \\
& | & \mathit{Var} := \mathit{Exp} \\
& | & \mathit{Exp}
\end{array}
$$

Figure 4.1.0: Grammar for basic blocks.

is defined to be the set of dangling breaks which are reachable in $n : P$. The corresponding set of labels associated with the reachable dangling breaks is denoted

$$RDBL_P \subseteq DBL_P$$

With Definition 4.1.1 above we are now able to define a function which converts from the syntactic program representation to the graph representation. This function is quite straightforward to describe since structural programming constructs admit a very natural conversion to CFGs. The only hitch is that we also have breaks and labeled statements, which can result in non-localized control flow.

We handled a similar problem already with the definition of our framework in Section 3.1. There we used the break environment to hold the data associated with breaks; and we waited to reach the correct labeled statement before discharging the values from the environment. In our current instance we instead push label information *down* to each individual break statement where it is used locally. In what follows, we use the symbol ρ and its obvious variations to denote a partial function from *Label* to *Node*.

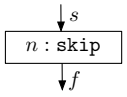
Definition 4.1.5 ($\mathcal{C}[\![\cdot]\!]$).

$$\mathcal{C}[\![\cdot]\!] : \mathit{Prgm} \to (\mathit{Label} \rightharpoonup \mathit{Node}) \to \mathit{CFG}$$

Use of $n', n'' \in \mathit{Node}$ below assumes that these nodes are *free* nodes not already used. A node is considered free if it not a member of any set N_{P_x} where P_x is a program in

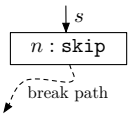
use in the exposition. Our definition includes both a formal definition and a picture.

$\mathcal{C}[\![n : \texttt{skip}]\!]\ \rho =$



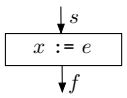
$= (n, n, \emptyset, \epsilon[n \mapsto \texttt{skip}])$

$\mathcal{C}[\![n : \texttt{break}\ l]\!]\ \rho =$



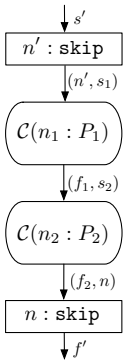
$= (n, \texttt{nil}, \{(n, \rho(l))\}, \epsilon[n \mapsto \texttt{skip}])$

$\mathcal{C}[\![n : x\ \texttt{:=}\ e]\!]\ \rho =$



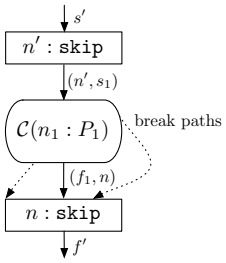
$= (n, n, \emptyset, \epsilon[n \mapsto x\ \texttt{:=}\ e])$

$\mathcal{C}[\![n : (n_1 : P_1); (n_2 : P_2)]\!]\ \rho =$



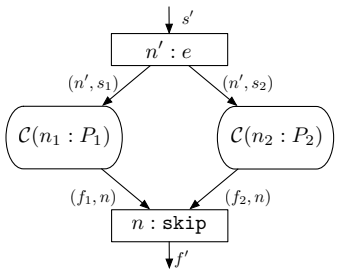
$$
\begin{aligned}
= \text{let } & (s_1, f_1, \delta_1, \lambda_1) := \mathcal{C}[\![n_1 : P_1]\!]\ \rho \\
& (s_2, f_2, \delta_2, \lambda_2) := \mathcal{C}[\![n_2 : P_2]\!]\ \rho \\
\text{in } & \text{let } s' := s_1 \\
& \quad f' := f_1 = \texttt{nowhere} \text{ or } f_2 = \texttt{nowhere}\ ?\ \texttt{nowhere} : n \\
& \quad \delta' := \delta_1 \cup \delta_2 \cup \{(n', s_1), (f_1, s_2), (f_2, n)\} \\
& \quad \lambda' := (\lambda_1 \cup \lambda_2)[n \mapsto \texttt{skip}, n' \mapsto \texttt{skip}] \\
& \text{in } (s', f', \delta', \lambda')
\end{aligned}
$$

$\mathcal{C}[\![n : l\colon\ n_1 : P_1]\!]\ \rho =$



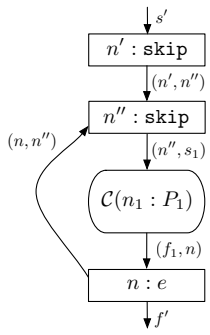
$$
\begin{aligned}
&= \text{let } (s_1, f_1, \delta_1, \lambda_1) := \mathcal{C}[\![n_1 : P_1]\!]\ \rho[l \mapsto n] \\
&\quad \text{in let } s' := n' \\
&\qquad\quad f' := f_1 = \texttt{nowhere and } l \notin RDBL_{P_1} \ ?\ \texttt{nowhere} : n \\
&\qquad\quad \delta' := \delta_1 \cup \{(n', s_1), (f_1, n)\} \\
&\qquad\quad \lambda' := \lambda_1[n \mapsto \texttt{skip}, n' \mapsto \texttt{skip}] \\
&\qquad \text{in } (s', f', \delta', \lambda')
\end{aligned}
$$

$\mathcal{C}[\![n : \texttt{if}\ e\ \texttt{then}\ n_1 : P_1\ \texttt{else}\ n_2 : P_2]\!]\ \rho =$



$$
\begin{aligned}
= \text{let } & (s_1, f_1, \delta_1, \lambda_1) := \mathcal{C}[\![n_1 : P_1]\!]\ \rho \\
& (s_2, f_2, \delta_2, \lambda_2) := \mathcal{C}[\![n_2 : P_2]\!]\ \rho \\
\text{in } & \text{let } s' := n' \\
& \quad f' := f_1 = \texttt{nowhere} \text{ and } f_2 = \texttt{nowhere}\ ?\ \texttt{nowhere} : n \\
& \quad \delta' := \delta_1 \cup \delta_1 \cup \{(n', s_1), (n', s_2), (f_1, n), (f_2, n)\} \\
& \quad \lambda' := (\lambda_1 \cup \lambda_2)[n' \mapsto e, n \mapsto \texttt{skip}] \\
& \text{in } (s', f', \delta', \lambda')
\end{aligned}
$$

$\mathcal{C}[\![n : \texttt{do}\ n_1 : P_1\ \texttt{while}\ e]\!]\ \rho =$



$$
\begin{aligned}
&= \text{let } (s_1, f_1, \delta_1, \lambda_1) := \mathcal{C}[\![n_1 : P_1]\!]\ \rho \\
&\quad \text{in let } s' := n' \\
&\qquad\quad f' := f_1 = \texttt{nowhere}\ ?\ \texttt{nowhere} : n \\
&\qquad\quad \delta' := \delta_1 \cup \{(n', n''), (n'', s_1), (f_1, n), (n, n'')\} \\
&\qquad\quad \lambda' := \lambda_1[n' \mapsto \texttt{skip}, n'' \mapsto \texttt{skip}, n \mapsto e] \\
&\qquad \text{in } (s', f', \delta', \lambda')
\end{aligned}
$$

We defined the $\mathcal{C}[\![\cdot]\!]$ function so that we have a formal way of converting program source into a control flow graph. The idea is that soundness will be a result of comparing the value of $\mathcal{F}[\![n : P]\!]$ with the r esult of applying the algorithm in Figure 2.2.2 on $\mathcal{C}[\![n : P]\!]$. We use the function $\mathcal{G}[\![\cdot]\!]$ as a wrapper function denoting application of the algorithm.

Definition 4.1.6 ($\mathcal{G}$).

$$\mathcal{G} : \mathit{CFG} \to (\mathit{Node} \rightharpoonup \mathit{Data})$$

Alternatively, using the names from Section 3.1 the $\mathcal{G}[\![\cdot]\!]$ is equivalently typed

$$\mathcal{G} : \mathit{CFG} \to \mathit{DataEnv}$$

Now, let $C = (s, f, \delta, \lambda) \in \mathit{CFG}$ and fix an analysis. We define $\mathcal{G}[\![C]\!]$ to be the result of applying the algorithm in Figure 2.2.1 with the analysis given. For each node n

in C, we define the transfer function f_n to be such that

- If $\lambda(n) = \texttt{skip}$ then $f_n = id$.
- If $\lambda(n) = x \texttt{ := } e$ then $f_n(d) = d; \texttt{assign}(n, x, e)$.
- If $\lambda(n) = e$ then $f_n(d) = d; \texttt{exp}(e)$.

For the "result" of the algorithm we take the *out* array coerced into a partial map from nodes to data elements in the obvious way.

Note that our framework returns a triple (φ, η, d) whereas the CFG based framework returns only a single data environment; which we will denote with the symbol ϑ. We need a way of comparing these two types if we are going to make a meaningful comparison of the frameworks. The primary difference between the two is that ϑ encapsulates the data for the entire program being analyzed, rather than splitting up normal and abnormal exits as we did with the separate data and break environments in the definition of our framework. Therefore, the relation we use to equate the results of the two analysis frameworks needs to reflect this; leading to the following definition.

Definition 4.1.7. For any program $n : P$ and label l, we define

$$RDBN_P \restriction_l \subseteq RDBN_P$$

to be the set of reachable dangling breaks in $n : P$ having target l.

The next definition makes the comparison discussed above. We use the notion of a *conservative approximation* of the data flow result. As discussed in Section 2.2, the *true* solution to the data flow problem is the meet over paths calculation. However, any value less than the true value in the data lattice is a conservative approximation. One approximation is *above* another if it is greater in the lattice.

Definition 4.1.8. Let $n : P \in Prgm$ and let $C = \mathcal{C}[\![n : P]\!]\ \epsilon$. For any (φ, η, d) and ϑ, we write

$$(\varphi, \eta, d) \simeq_P \vartheta$$

if and only if for all $n' \in RN_P$, $\varphi(n')$ is a conservative approximation of the data flow problem for C and above $\vartheta(n')$, and for all $l \in RDBL_P$, $\eta(l)$ is a conservative approximation above

$$\bigwedge_{n' \in RDBN_P \restriction_l} \vartheta(n')$$

Finally, we can formally state the Soundness Theorem. The statement assumes all of the axioms that we have laid out previously, *distributivity* for example.

Theorem 4.1.1 (Soundness of $\mathcal{F}[\![\cdot]\!]$). *For any program $n : P$*

$$\mathcal{F}[\![n : P]\!]\ \epsilon\ \epsilon\ I \simeq_P \mathcal{G}[\![\mathcal{C}[\![n : P]\!]\ \epsilon]\!]$$

Therefore, the data flow analysis $\mathcal{F}[\![\cdot]\!]$ is a conservative solution of the data flow problem and hence the framework is sound.

4.2 Some Lemmas about CFGs

We begin our work toward proving Theorem 4.1.1 with several definitions and lemmas that primarily cover different aspects of how program structure is preserved when applying the $\mathcal{C}[\![\cdot]\!]$ function. We reserve the symbols C and (s, f, δ, λ) and their obvious variations to denote control flow graphs.

This first definition covers some basic terminology relating to paths through the control flow graph.

Definition 4.2.1. A *control flow path* through $C = (s, f, \delta, \lambda)$ is a sequence of nodes

$$n_1, \ldots, n_j$$

such that for all $i < j$, $(n_i, n_{i+1}) \in \delta$. As we briefly noted in Section 2.2 when we discussed the *mop* calculation, if $n_1 = s$ then the control flow path is called an *execution path*. In either case we will also sometimes qualify the definition by saying a *path to* n_j.

The second definition is the control flow graph equivalent of the syntactic concept of dangling breaks, defined by Definition 3.5.2.

Definition 4.2.2. Given $C = (s, f, \delta, \lambda)$, a node n is called a *dangling break* if and only if the following two conditions can be shown to hold:

- For some label l, $\lambda(n) = \texttt{break}\ l$.
- There is at most one outgoing edge from n, and if $(n, n') \in \delta$ is that edge, then $\lambda(n') \uparrow$.

The following definition aggregates the nodes which are reachable in the control flow graph. We call these nodes the *on graph* nodes and the intention is that these will mirror the RN_P set.

Definition 4.2.3. Let $C = (s, f, \delta, \lambda) \in \mathit{CFG}$. We define the set of *on graph* nodes

$$OGN_C \subseteq \mathit{Node}$$

to be such that $n \in OGN_C$ if and only if there exists an execution path to n and n does not directly follow a dangling break. If C was generated from some program $n : P$ by $\mathcal{C}[\![\cdot]\!]$, as will usually be the case in what follows, then we exclude the free nodes from the set of on graph nodes.

We also define a similar notion for making a correspondence with the set of reachable dangling breaks $RDBN_P$.

Definition 4.2.4. Let $C = (s, f, \delta, \lambda) \in \mathit{CFG}$. We define the set of *on graph dangling breaks*, denoted

$$OGDB_C \subseteq OGN_C$$

to be the set of on graph nodes which are also dangling breaks.

The relation defined next identifies when one CFG is a subgraph of another. Note that because of how we treat the `nowhere` token, the requirement placed on the δ part cannot be *strengthened* to simple subset inclusion.

Definition 4.2.5. Let $C_x = (s_x, f_x, \delta_x, \lambda_x)$ and $C_y = (s_y, f_y, \delta_y, \lambda_y)$ be any two control flow graphs. Then the relation

$$C_x \subseteq C_y$$

holds if and only if for all $(n, n') \in (Node)^2$, if $(n, n') \in \delta_x$ then $(n, n') \in \delta_y$, and for all $n \in Node$ such that $\lambda_x(n) \downarrow$, $\lambda_y(n) = \lambda_x(n)$.

The set defined next determines how a subgraph is connected within the larger structure. For any given subgraph we pick out the nodes just on the periphery of the subgraph, meaning the ones from which control can transfer directly into the subgraph.

Definition 4.2.6. For any $C = (s, f, \delta, \lambda) \in CFG$ and $S \subseteq Node$ we define the set of *periphery nodes of* S

$$PN_{S,C} \subseteq Node$$

to be the set of nodes such that $n \in PN_{S,C}$ if and only if $n \notin S$ and there is an $n' \in S$ with $(n, n') \in \delta$.

The next definition provides the primary mechanism through which we can correlate a CFG with the program source that generated it. This is done through what we call *anchor points*, which are unique entry points on the periphery of the subgraph. We will show later that the important anchor points correspond exactly to the production rules of the grammar in Figure 3.1.2.

Definition 4.2.7. Let $C_x = (s_x, f_x, \delta_x, \lambda_x)$ and $C_y = (s_y, f_y, \delta_y, \lambda_y)$ be CFGs and take any node $n \in Node$. The *anchor point* relation

$$C_x \preceq_n C_y$$

holds if and only if $C_x \subseteq C_y$, for $S = OGN_{C_x}$, $PN_{S,C_x} = \{n\}$, and the only edge reaching from n into C_x is (n, s_x). When the subgraph is implied we refer to n as an anchor point without further qualification.

In the remainder of this Section we develop some lemmas about control flow graphs and about the $\mathcal{C}[\![\cdot]\!]$ and $\mathcal{G}[\![\cdot]\!]$ functions. In those cases where we talk about or apply the $\mathcal{C}[\![\cdot]\!]$ function in different but related contexts, we will assume that the free nodes used in the construction of the control flow graph as described in Definition 4.1.2 all correspond in the obvious way. This saves us from having to devise a way of keeping track of all the bound nodes and then renaming the free nodes to make everything consistent.

The first result is similar to Lemma 3.5.1, it says that if the f coordinate of a CFG is non-nil after applying $C[\![\cdot]\!]$ to program $n : P$, then $f = n$.

Lemma 4.2.1. *Let $n : P \in Prgm$ and take any partial mapping ρ. If we let*

$$C = (s, f, \delta, \lambda) = \mathcal{C}[\![n : P]\!]\ \rho$$

then $f = n$ or $f =$ `nowhere`.

Proof. Clear from Definition 4.1.5. □

The next Lemma says that the on graph nodes used in the construction of $\mathcal{C}[\![n : P]\!]\ \rho$ are always the reachable nodes and that dangling breaks always transfer control to the appropriate node mapped by ρ. This should not come as a surprise considering the intuition described earlier when we defined the $\mathcal{C}[\![\cdot]\!]$ function. We assume that ρ is sensible, meaning that it does not map any label to a node in N_P.

Lemma 4.2.2. *For any program $n : P$ and partial map ρ, if we let*

$$C = (s, f, \delta, \lambda) = \mathcal{C}[\![n : P]\!]\ \rho$$

then all of the following hold:

- $OGN_C = RN_P$
- $OGDB_C = RDBN_P$

- *If we let*

$$C' = (s', f', \delta', \lambda') = \mathcal{C}[\![n : P]\!]\ \epsilon$$

then $s' = s$, $f' = f$, $\lambda' = \lambda$, *and for all pairs of nodes* $(n_x, n_y) \in \delta$ *if and only if* $(n_x, n_y) \in \delta'$ *or for some label* l_x, $n_x \in RDBN_C \restriction_{l_x}$ *and* $\rho(l_x) = n_y$.

Proof. The proof is by structural induction on programs.

Base Case:

- The base case includes all of the atomic constructs listed in the grammar in Figure 3.1.2 and follows easily by applying the relevant definitions.

Induction Step:

- Consider the case where $n : P = n : l\ :\ n_1 : P_1$. Let

$$\begin{aligned} C_1 = (s_1, f_1, \delta_1, \lambda_1) &= \mathcal{C}[\![n_1 : P_1]\!]\ \rho[l \mapsto n] \\ C_1' = (s_1', f_1', \delta_1', \lambda_1') &= \mathcal{C}[\![n_1 : P_1]\!]\ \epsilon[l \mapsto n] \end{aligned}$$

For the first part we have to show that $OGN_C = RN_P$. By the induction hypothesis we get $OGN_{C_1} = RN_{P_1}$, and it's clear from Definition 4.1.5 that $OGN_{C_1} \subseteq OGN_C$. In addition, from Definition 3.1.1 we get that $RN_{P_1} \subseteq RN_P$. For node n specifically, it can be seen from Definition 4.1.5 that $n \in OGN_C$ if and only if $f_1 \in OGN_{C_1}$ or, by the induction hypothesis, if $BG^n_{C_1} \restriction_l \neq \emptyset$. Again by the induction hypothesis, and also from Lemma 4.2.1, we get that this condition is equivalent to $f_1 = n_1 \in RN_{P_1}$ or $RDBN_{P_1} \restriction_l \neq \emptyset$. By Definition 3.1.1 this is exactly when $n \in RN_P$. It is easy to verify that $OGN_C \subseteq OGN_{C_1} \cup \{n\}$ and that $RN_P \subseteq RN_{P_1} \cup \{n\}$, completing the proof.

For the second part we have to show that $OGDB_C = RDBN_P$. Let $n' \in Node$ be an arbitrary node. By the induction hypothesis and Definition 4.1.5 we get, $n' \in BG^n_P$ if and only if $n' \in BG^n_{C_1}$ and $n' \notin BG^n_{C_1} \restriction_l$. At the same

time, by Definition 3.1.1, $n' \in RDBN_P$ if and only if $n' \in RDBN_{P_1}$ and $n' \notin RDBN_{P_1} \restriction_l$. These two conditions are equivalent by the induction hypothesis.

For the final part, we start by noting that

$$\begin{aligned} \delta &= \delta_1 \cup \{(n', s_1), (f_1, n)\} \\ \delta' &= \delta'_1 \cup \{(n', s'_1), (f'_1, n)\} \end{aligned}$$

By the induction hypothesis, $s'_1 = s_1$ and $f'_1 = f_1$. The rest follows easily from Definition 4.1.5 and the induction hypothesis.

- Consider the case where $n : P = n : (n_1 : P_1)\,;(n_2 : P_2)$. Let

$$\begin{aligned} C_1 = (s_1, f_1, \delta_1, \lambda_1) &= \mathcal{C}[\![n_1 : P_1]\!]\ \rho[l \mapsto n] \\ C_2 = (s_2, f_2, \delta_2, \lambda_2) &= \mathcal{C}[\![n_2 : P_2]\!]\ \rho[l \mapsto n] \end{aligned}$$

  For the first part we have to show that $OGN_P = RN_P$. By the induction hypothesis we get $OGN_{C_1} = RN_{P_1}$, and it's clear from Definition 4.1.5 that $OGN_{C_1} \subseteq OGN_C$. In addition, Definition 3.1.1 and Lemma 3.5.2 show that $RN_{P_1} \subseteq RN_P$. Now, again applying Definition 4.1.5 we get that $OGN_{C_2} \subseteq OGN_C$ if and only if $f_1 \in OGN_{C_1}$ which by Lemma 4.2.1 is true if and only if $f_1 = n_1$. By the induction hypothesis this happens exactly when $n_1 \in RN_{P_1}$, which by Definition 3.1.1, Lemma 3.5.1, and Theorem 3.5.1 is equivalent to $RN_{P_2} \subseteq RN_P$. Moreover, the induction hypothesis shows that $OGN_{C_2} = RN_{P_2}$. The fact that $n \in OGN_C$ if and only if $n \in RN_P$ can be shown in a similar manner. Finally, it is straightforward to verify that $OGN_C \subseteq OGN_{C_1} \cup OGN_{C_2} \cup \{n\}$ and that $RN_P \subseteq RN_{P_1} \cup RN_{P_2} \cup \{n\}$.

  The second and third parts can be shown in a manner similar to the argument given for the labeled statement case.

- The other two cases, where $n : P$ is a conditional statement or a do-while loop, follow from entirely similar arguments to ones given above for the labeled statement and sequential composition constructs.

□

In Section 2.2 we explained that updates to the *out* array can be applied in any order as long as each block is updated at least once during every iteration of the outer loop in Figure 2.2.2. The following lemma applies this property to determine how single changes to the *out* array propagate into a subgraph. Specifically, when $out[n]$ changes and we want to know the full effect of the change on subgraph $C_1 \subseteq C$, we simply keep updating the on graph nodes of C_1 until they reach a *steady state*.

To avoid confusion we will sometimes subscript the *in* and *out* arrays to denote which calculation $G[\![\cdot]\!]$ we are referring to when there could be more than one. So out_C refers to the out array used in the calculation of $\mathcal{G}[\![C]\!]$.

Lemma 4.2.3. *Let $C_1 = (s_1, f_1, \delta_1, \lambda_1)$ and $C = (s, f, \delta, \lambda)$ be CFGs and suppose that for some node n, $C_1 \preceq_n C$. In addition, let $\vartheta_1 = \mathcal{G}[\![C_1]\!]$ and consider running the algorithm in Figure 2.2.2 to calculate $\mathcal{G}[\![C]\!]$, supposing that $out_C[n]$ is updated and now takes the value t. It is then correct (a conservative estimate) to set, for all $n' \in OGN_{C_1}$, $out_C[n'] = out_C[n]; \vartheta_1(n') = t; \vartheta_1(n')$ as an alternative to calculating the steady state value, as described above.*

Proof. Consider the CFG $C_1' = (n, f_1, \delta_1 \cup \{(n, s_1)\}, \lambda)$ and the calculation of $\vartheta_1' = \mathcal{G}[\![C_1']\!]$ with f_n defined such that $f_n(d) = d; t$. It is clear from the algorithm in Figure 2.2.2 that for every $n' \in OGN_{C_1}$ the steady state value of $out_C[n']$ equals $\vartheta_1'(n')$.

In Section 2.2 we explained that the *meet over paths* solution is considered the fully correct one for the data flow problem. We can then apply this result to calculate ϑ_1' in terms of ϑ_1 and hence a correct alternative to the the steady state out_C values in terms of ϑ_1. For any node $n' \in OGN_{C_1}$, the set of paths reaching n' in C_1' is exactly the set of paths reaching n' in C_1 with block n added as a prefix. Therefore, by distributivity, $mop(n', C_1') = f_n(I); mop(n', C_1)$. From monotonicity we get that $I; t; \vartheta_1(n') = t; \vartheta_1(n')$ is a convervative approximation of the data flow solution. To get that it is above $\vartheta(n')$ requires that we apply monotinicity repeatedly while updating the graph in breadth first order. □

Corollary 4.2.1. *Let $C_1 = (s_1, f_1, \delta_1, \lambda_1)$ and $C = (s, f, \delta, \lambda)$ be CFGs and suppose that $C_1 \preceq_n C$ for some node n. If we let $\vartheta_1 = \mathcal{G}[\![C_1]\!]$ and $\vartheta = \mathcal{G}[\![C]\!]$, then for all $n' \in OGN_{C_1}$, $\vartheta(n); \vartheta_1(n')$ is a conservative estimate of the data flow solution.*

Proof. Obvious from Lemma 4.2.3. □

4.3 Proof of Soundness

We are now set to prove the Soundness Theorem for our data flow analysis framework, showing that our framework calculates the correct values relative to the classical CFG-based algorithm. We reprint Theorem 4.1.1 here for convenience.

Theorem 4.3.1. *For any program $n : P$*

$$\mathcal{F}[\![n : P]\!]\ \epsilon\ \epsilon\ I \simeq_P \mathcal{G}[\![\mathcal{C}[\![n : P]\!]\ \epsilon]\!]$$

Proof. The proof is by induction on the structure of programs.

Base Case:

- The result is easily derived for the atomic constructs.

Induction Step:

- Consider when $n : P = n : l\ :\ n_1 : P_1$. We begin by applying Definition 3.1.1 to get

$$\begin{aligned}(\varphi, \eta, d) = \mathcal{F}[\![n : l\ :\ n_1 : P_1]\!]\ \epsilon\ \epsilon\ I = \\ \text{let } (\varphi_1, \eta_1, d_1) := \mathcal{F}[\![n_1 : P_1]\!]\ \epsilon\ (\epsilon \setminus \{l\})\ I \\ \text{in } \ (\varphi_1[n \mapsto d_1 \wedge \epsilon(l)], \epsilon \setminus \{l\}, d_1 \wedge \eta_1(l))\end{aligned}$$

and Definition 4.1.2 to get

$$
\begin{array}{l}
C = \mathcal{C}[\![n : l \ : \ n_1 : P_1]\!] \ \epsilon = \\
\qquad \text{let } (s_1, f_1, \delta_1, \lambda_1) := \mathcal{C}[\![n_1 : P_1]\!] \ \epsilon[l \mapsto n] \\
\qquad \text{in } \ \text{let } s' := n' \\
\qquad\qquad f' := f_1 = \texttt{nowhere and } l \notin RDBL_{P_1} \ ? \ \texttt{nowhere} : n \\
\qquad\qquad \delta' := \delta_1 \cup \{(n', s_1), (f_1, n)\} \\
\qquad\qquad \lambda' := \lambda_1[n \mapsto \texttt{skip}, n' \mapsto \texttt{skip}] \\
\qquad\quad \text{in } (s', f', \delta', \lambda')
\end{array}
$$

where n' is a free node. In addition, we let

$$\vartheta = \mathcal{G}[\![C]\!]$$

and

$$
\begin{array}{rcl}
C_1 & = & (s_1, f_1, \delta_1, \lambda_1) \\
\vartheta_1 & = & \mathcal{G}[\![C_1]\!]
\end{array}
$$

We need to show that $(\varphi, \eta, d) \simeq_P \vartheta$. Note that by Lemma 4.2.2

$$OGN_C = RN_P \subseteq OGN_{C_1} \cup \{n\} = RN_{P_1} \cup \{n\}$$

and

$$OGN_{C_1} = RN_{P_1}$$

Let $n_x \in OGN_{C_1}$ and, furthermore, note that $C_1 \preceq_{n'} C$. Therefore, by Corollary 4.2.1, $\vartheta(n'); \vartheta_1(n_x)$ is a conservative approximation above $\vartheta(n_x)$, and since it's clear that $\vartheta(n') = I$, $\vartheta_1(n_x)$ is a conservative approximation above $\vartheta(n_x)$. Now, by Definition 3.1.1, $\varphi(n_x) = \varphi_1(n_x)$, therefore $\varphi(n_x)$ is also a conservative approximation above $\vartheta(n_x)$ by the induction hypothesis and Lemma 4.2.2. In addition, for node n we have $\varphi(n) = d_1 \wedge \eta_1(l)$, which by Lemma 3.5.1 means that $\varphi(n) = \varphi_1(n_1) \wedge \eta_1(l)$. Again from the induction hypothesis and Lemma

4.2.2, $\eta_1(l)$ is a conservative approximation above

$$\bigwedge_{n_x \in RDBN_{P_1}\restriction_l} \vartheta_1(n_x)$$

Taking into consideration all this, Lemma 4.2.2 again, and the algorithm in Figure 2.2.2 we get that

$$\vartheta_1(n_1) \wedge \bigwedge_{n_x \in RDBN_{P_1}\restriction_l} \vartheta_1(n_x)$$

is a conservative approximation above $\vartheta(n)$. Therefore $\varphi(n)$ is a conservative approximation above $\vartheta(n)$.

We also have to show that for all $l_x \in RDBL_P$, $\eta(l_x)$ is a conservative approximation above

$$\bigwedge_{n_x \in RDBN_P\restriction_{l_x}} \vartheta(n_x)$$

From Lemma 4.2.2 we get $RDBN_P = RDBN_{P_1} \setminus RDBN_{P_1}\restriction_l$, and therefore by above results

$$\bigwedge_{n_x \in RDBN_{P_1}\restriction_{l_x}} \vartheta_1(n_x)$$

is a conservative approximation above

$$\bigwedge_{n_x \in RDBN_P\restriction_{l_x}} \vartheta(n_x)$$

From the induction hypothesis we have $\eta_1(l_x)$ as a conservative approximation above

$$\bigwedge_{n_x \in RDBN_{P_1}\restriction_{l_x}} \vartheta_1(n_x)$$

and since $\eta_1(l_x) = \eta(l_x)$, we get $\eta(l_x)$ is a conservative approximation above

$$\bigwedge_{n_x \in RDBN_P \restriction_{l_x}} \vartheta(n_x)$$

- Consider when $n : P = n : (n_1 : P_1)\ ;\ (n_2 : P_2)$. We begin by applying Definition 3.1.1 to get

$$\begin{aligned}(\varphi, \eta, d) = \mathcal{F}[\![n : (n_1 : P_1)\ ;\ (n_2 : P_2)]\!]\ \epsilon\ \epsilon\ I = \\ \text{let } (\varphi_1, \eta_1, d_1) := \mathcal{F}[\![n_1 : P_1]\!]\ \epsilon\ \epsilon\ I; \texttt{exp}(e) \\ (\varphi_2, \eta_2, d_2) := \mathcal{F}[\![n_2 : P_2]\!]\ \epsilon\ \epsilon\ I; \texttt{exp}(e) \\ \text{in } ((\varphi_1 \cup \varphi_2)[n \mapsto d_1 \wedge d_2], \eta_1 \wedge \eta_2, d_1 \wedge d_2)\end{aligned}$$

and Definition 4.1.2 to get

$$\begin{aligned}C = \mathcal{C}[\![n : (n_1 : P_1)\ ;\ n_2 : P_2]\!]\ \epsilon = \\ \text{let } (s_1, f_1, \delta_1, \lambda_1) := \mathcal{C}[\![n_1 : P_1]\!]\ \epsilon \\ (s_2, f_2, \delta_2, \lambda_2) := \mathcal{C}[\![n_2 : P_2]\!]\ \epsilon \\ \text{in let } s' := n' \\ f' := f_1 = \texttt{nowhere} \text{ and } f_2 = \texttt{nowhere}\ ?\ \texttt{nowhere} : n \\ \delta' := \delta_1 \cup \delta_1 \cup \{(n', s_1), (n', s_2), (f_1, n), (f_2, n)\} \\ \lambda' := (\lambda_1 \cup \lambda_2)[n' \mapsto e, n \mapsto \texttt{skip}] \\ \text{in } (s', f', \delta', \lambda')\end{aligned}$$

where n' is a free node. In addition, we let

$$\vartheta = \mathcal{G}[\![C]\!]$$

and

$$\begin{aligned} C_1 &= (s_1, f_1, \delta_1, \lambda_1) \\ \vartheta_1 &= \mathcal{G}[\![C_1]\!] \\ C_2 &= (s_2, f_2, \delta_2, \lambda_2) \\ \vartheta_2 &= \mathcal{G}[\![C_2]\!] \end{aligned}$$

and

$$(\varphi_2', \eta_2', d_2') = F[\![n_2 : P_2]\!] \ \epsilon \ \epsilon \ I$$

and we need to show that $(\varphi, \eta, d) \simeq_P \vartheta$. Note that by Lemma 4.2.2

$$OGN_C = RN_P \subseteq OGN_{C_1} \cup OGN_{C_2} \cup \{n\} = RN_{P_1} \cup RN_{P_2} \cup \{n\}$$

and

$$\begin{aligned} OGN_{C_1} &= RN_{P_1} \\ OGN_{C_2} &= RN_{P_2} \end{aligned}$$

Let $n_x \in OGN_{C_1}$, and, furthermore, note that $C_1 \preceq_{n'} C$. Therefore by Corollary 4.2.1, $\vartheta(n'); \vartheta_1(n_x)$ is a conservative approximation above $\vartheta(n_x)$, and since it's clear that $\vartheta(n') = I$, $\vartheta(n_x)$ is a conservative approximation above $\vartheta(n_x)$. Now, by Definition 3.1.1 and Lemma 3.5.2, $\varphi(n_x) = \varphi_1(n_x)$, and therefore $\varphi(n_x)$ is a conservative approximation above $\vartheta(n_x)$ by the induction hypothesis. Now, let $n_x \in OGN_{C_2}$ and note that $C_2 \preceq_{n_1} C$ or $OGN_{C_2} \cap OGN_C = \emptyset$. In the first case, by Corollary 4.2.1, $\vartheta(n_1); \vartheta_2(n_x)$ is a conservative approximation above $\vartheta(n_x)$. Then by above results we get that $\varphi_1(n_1); \vartheta_2(n_x)$ is a conservative approximation above $\vartheta(n_x)$. By Theorem 3.5.1 and Lemma 3.5.1 we get $\varphi(n_x) = \varphi_1(n_1); \varphi_2'(n_x)$ and then by the induction hypothesis we get $\varphi(n_x)$ is a conservative approximation above $\vartheta(n_x)$. For n, we get $\vartheta(n) = \vartheta(n_2)$. Similarly $\varphi(n) = \varphi_1(n_1); \varphi_2'(n_2)$ as shown above, therefore the result holds.

We also have to show that for all $l_x \in RDBN_P$, $\eta(l_x)$ is a conservative approximation above

$$\bigwedge_{n_x \in RDBN_P \restriction_{l_x}} \vartheta(n_x)$$

This follows from a case by case analysis using similar arguments to those given above.

- The other two constructs follow from similar arguments. In the case of do-while loops, we simply apply induction over the necessary length of the fixed point iteration and an argument similar to what we have for sequential composition above.

□

Chapter 5

Conclusions

Run time program generation is an elegant technology useful for managing complexity while developing large software programs. For example, RTPG has been fruitfully applied toward run time code optimization by taking advantage of dynamically available data, and also in writing generic, adaptable code. Naturally, the development of RTPG technology has introduced a number of new research problems. The work presented in this Thesis addresses the particular one of optimizing run time compilation through the staging of data flow calculations.

Our notable contributions started with the description of a formal framework for calculating data flow analyses. We then showed though some examples that our framework is robust, supporting many of the classical analyses as well as more sophisticated ones, like type checking. From this foundation we built up a theory around staging: forcing as much work as possible into an "offline" stage, on static data. We then showed how the theory can be made executable as a procedural algorithm implementable as part of a RTPG system. We gave initial results of an implementation we did in Jumbo, which were promising. Finally, we proved the soundness of our framework, showing that it computes the correct result.

In the future we should investigate carefully how to make the framework industrially feasible in terms of performance, and explore further generalizations of the framework more broadly applicable to the compilation process. In particular, the starting point for such work would be a careful profiling of the run time compilation process to identify the most promising cases for advancement and impact.

References

[1] Alfred V. Aho and Ravi Sethi and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Barbara G. Ryder and Marvin C. Paull. Incremental data-flow analysis algorithms. *ACM Trans. Program. Lang. Syst.*, 10(1):1–50, 1988.

[3] Baris Aktemur and Joel Jones and Samuel Kamin and Lars Clausen. Optimizing Marshalling by Run-time Program Generation. In *Generative Programming and Component Engineering: 4th International Conference, GPCE 2005*, volume 3676 of *Lecture Notes in Computer Science*. Springer, 2005.

[4] Baris Aktemur and Samuel Kamin. Mumbo: A Rule Based Implementation of a Run-time Program Generation Language. *ENTCS*, 147(1):31–55, 2006. Proceedings of the 6th International Workshop on Rule-Based Programming (RULE 2005).

[5] Barry K. Rosen. High-level Data Flow Analysis. *Commun. ACM*, 20(10):712–724, 1977.

[6] Dawson R. Engler and Wilson C. Hsieh and M. Frans Kaashoek. 'C: A Language for High-level, Efficient, and Machine-independent Dynamic Code Generation. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 131–144. ACM Press, 1996.

[7] Lars Reder Clausen. *Optimizations in Distributed Run-time Compilation*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.

[8] Massimiliano Poletto and Wilson C. Hsieh and Dawson R. Engler and M. Frans Kaashoek. C and tcc: a language and compiler for dynamic code generation. *ACM Trans. Program. Lang. Syst.*, 21(2):324–369, 1999.

[9] Philip Morton. Source Level Rewriting of the Jumbo Runtime Code Generation Framework. Master's thesis, University of Illinois at Urbana-Champaign, 2005.

[10] Sam Kamin. Routine Run-Time Code Generation. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 208–220. ACM Press, 2003.

[11] Sam Kamin. Invited Application Paper: Program Generation Considered Easy. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial Evaluation and Semantics-based program Manipulation*, pages 68–79. ACM Press, 2004.

[12] Sam Kamin and Lars Clausen and Ava Jarvis. Jumbo: Run-time Code Generation for Java and Its Applications. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 48–56. IEEE Computer Society, 2003.

[13] Sam Kamin and Miranda Callahan and Lars Clausen. Lightweight and Generative Components II: Binary-Level Components. In *In SAIG 2000*, pages 28–50, 2000.

[14] Samuel Kamin and Baris Aktemur and Philip Morton. Source Level Optimization of Run-time Program Generators. In *Generative Programming and Component Engineering: 4th International Conference, GPCE 2005*, volume 3676 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2005.

[15] Tankut Baris Aktemur. A Rule-Based Model of a Runtime Program Generation System. Master's thesis, University of Illinois at Urbana-Champaign, 2005.

[16] Walid Taha and Tim Sheard. MetaML and Multi-stage Programming with Explicit Annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.