Staging Static Analyses for Program Generation *

Sam Kamin Baris Aktemur Michael Katelman
University of Illinois at Urbana-Champaign
201 N.Goodwin Urbana, IL 61801 USA
{kamin,aktemur,katelman}@cs.uiuc.edu

Abstract

Program generators are most naturally specified using a quote/antiquote facility; the programmer writes programs with holes which are filled in, at program generation time, by other program fragments. If the programs are generated at compile-time, analysis and compilation follow generation, and no changes in the compiler are needed. However, if program generation is done at run time, compilation and analysis need to be optimized so that they will not overwhelm overall execution time. In this paper, we give a compositional framework for defining program analyses which leads directly to a method of staging these analyses. The staging allows the analysis of incomplete programs to be started at compile time; the residual work to be done at run time may be much less costly than the full analysis. We give frameworks for forward and backward analyses, present several examples of specific analyses, and give timing results showing significant speed-ups for the run-time portion of the analysis relative to the full analysis.

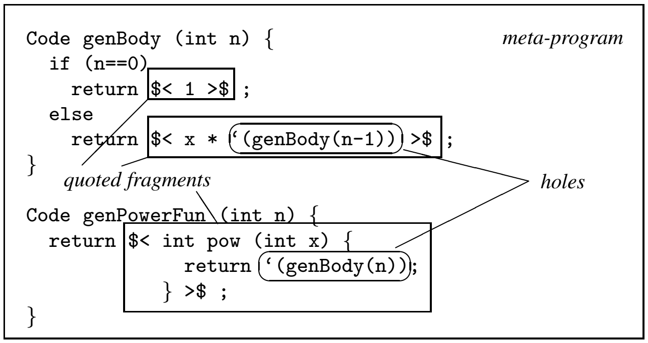
Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation

General Terms Languages, performance

Keywords run-time code generation, program generation, static analysis, staging

1. Introduction

We are concerned here with languages in which code generators are specified by embedding quoted program fragments within a larger program (the meta-program) [6, 12, 13, 4]. These quoted fragments include "holes" — portions of the program that are to be filled in with other fragments to generate a complete program (see Figure 1). Such systems provide a natural, easy to understand method of creating program generators. They raise several kinds of research questions: What properties of generated programs can be inferred from the initial set of fragments? How quickly the generated program can be generated? The latter is of most interest when program generation occurs at run time.



```
Code genBody (int n) {
  if (n==0)
    return $< 1 >$ ;
  else
    return $< x * `(genBody(n-1)) >$ ;
}
Code genPowerFun (int n) {
  return $< int pow (int x) {
         return `(genBody(n));
       } >$ ;
}
```

Figure 1. Terminology for program generators. When a fragment fills in a hole, we call it a *plug*. Note that a fragment is never used as a plug until all of its holes have been filled.

This paper addresses the question: how quickly can static analyses be performed on generated programs? To be precise: We are given a program $P[\bullet, \ldots, \bullet]$ with holes, and a collection of plugs $Q_1, \ldots, Q_n$. We want to find the result of some static analysis when applied to $P[Q_1, \ldots, Q_n]$. We could, at run time, fill in the plugs and run the analysis. However, we can save time by preprocessing P and the Q_i, and then combining them at run time to produce the same result.

We present a framework for static analyses which allows us to make a clear distinction between compile time — when we know all the fragments, but do not know which fragments will fill in which holes in which other fragments — and run time — when we create the generated program and can do the analysis. The ability to stage analyses depends upon finding an accurate representation for the dataflow functions; we present representations for several analyses. The staging can produce substantial speed-ups in the analyses.

We begin the technical presentation (Section 3) with our forward analysis framework, illustrating it with a simple

* Partial support for this work was received from NSF under grant CCR-0306221.

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.

analysis, *uninitialized variables*. We discuss how the framework allows for efficient staging of analyses, and in Section 4 present a collection of analyses. Section 5 presents the backward analysis framework. Section 6 gives performance results for various analyses and benchmark programs.

The contributions of this paper are three-fold: (1) We define frameworks for forward and backward analyses of abstract syntax trees (AST), including break statements, which explains how analyses can be staged. Staging requires that dataflow functions be represented "adequately." (2) We give representations for several dataflow problems, and for staged type checking. (3) We provide experimental evidence of speed-ups from staging.

A more detailed presentation of this work is given in [8].

2. Related Work

Our work shares with several others a concern with *representation* of dataflow functions, and some of our representations have appeared previously. In the area of *interprocedural dataflow analysis*, Sharir and Pnueli [17] introduced the idea of summarizing the analysis of an entire procedure. Rountev, Kagan and Marlowe [15] discuss concrete representations for these summary functions, to allow for "whole program" analysis of programs that use libraries; our representation for reaching definitions appears there. Reps, Horwitz, and Sagiv [14] give representations for a class of dataflow problems, including reaching definitions and linear constant propagation. (Interprocedural analysis is similar to staged analysis in that one can think of the procedure call as a "hole," and the procedure as a "plug." However, the control flow issues are very different; that work must deal with the notion of "valid" paths — where calls match returns — while we must deal with multiple-exit control structures.) To parallelize static analyses, Kramer, Gupta and Soffa [10] partition programs and analyze each partition to produce a summary of its effect on the program as a whole.

In *hybrid* analysis [16], Marlowe and Ryder partition a program based on strong components, representing dataflow functions for each component. A representation for reaching definitions that is "adequate" in our sense is given there. Marlowe and Ryder also talk about *incremental analysis* where the problem is to maintain the validity of an analysis during source program editing. But note the subtle but important distinction between *incremental* analysis and *staged* analysis: there, *any* node can change at any time; here, some parts of the program are fixed and some unknown, and the goal is to fully exploit the fixed parts.

In *approximate analysis* [18], the meta-program is analyzed to determine as much as possible about what the generated program will look like. This approach has the advantage of avoiding run-time analysis entirely, but the disadvantage that the analysis results are very approximate.

Lastly, we mention the work of Chambers et al. [3]. That work has the ambitious goal of *automatically* staging compilers: a user can indicate when some information will first become available, and the system will produce an optimizer to *efficiently* perform the optimization at that time. The broad goals of that work — optimizing run-time compilation — are the same as ours. However, we are much less ambitious about the use of automation (and, indeed, that work accommodates a limited number of optimizations); we are, instead, providing a mathematical framework that can facilitate the manual construction of staged analyses.

3. Framework for Forward Analysis

Our framework differs from the standard one [1] in that it analyzes abstract syntax trees (ASTs), not control-flow graphs (CFGs). Since program fragments appear as ASTs, this is the natural unit of analysis for our purposes. Note that we are considering only intraprocedural analysis in this paper. However, as noted above, our techniques have much in common with some interprocedural analyses; we expect the extension to be relatively straightforward.

In this section, we present our framework as the third in a sequence of frameworks of increasing complexity. For each framework, the plan is the same:

1. Present an analysis framework $\mathcal{F}$ for calculating dataflow values for AST's in a lattice $Data$.
2. Present a framework $\mathcal{R}$ for calculating representations of dataflow functions, given an "adequate" representation R.
3. Give a theorem relating representations produced by $\mathcal{R}$ to dataflow functions given by $\mathcal{F}$.
4. Give an alternative method of calculating representations, called $\mathcal{F}^R$, more efficient than $\mathcal{R}$, which uses the definition of $\mathcal{F}$ but applies it to representations rather than dataflow values.

As a running example in these sections, we use *uninitialized variables*, an analysis that calculates a list of variables that may have been used without being initialized.

The first framework contains only simple control structures; the theorems are trivial in this case, but we introduce notation and explain how staging works. The second framework handles break statements. These two frameworks calculate dataflow values only for the root of an AST; the final framework calculates values at each node within an AST.

Figure 2 shows the language we treat in this paper. Keep in mind that this is the language *inside quotations*. We do not include holes because these are not proper elements of the language. To avoid notational complexities, we allow holes only in statement position; allowing holes in expression position poses no fundamental problems.

Dataflow values are assumed to come from a lattice, called $Data$. Define *DFFun* to be the function space $Data \rightarrow Data$ (confined to functions that preserve $\top_{Data}$).

$e \in$ Exp
$x \in$ Var
$\ell \in$ Label
$P \in$ Pgm ::= x = e | skip | if e then P_1 else P_2 | P_1; P_2
| while e do P | ℓ : P | break ℓ

Figure 2. The language treated in this paper

$\mathcal{F}[\![\mathsf{skip}]\!] = \mathit{id}$
$\mathcal{F}[\![x = e]\!] = \mathit{asgn}(x, e)$
$\mathcal{F}[\![P_1;\ P_2]\!] = \mathcal{F}[\![P_1]\!]; \mathcal{F}[\![P_2]\!]$
$\mathcal{F}[\![\mathsf{if\ then}\ (e)\ P_1\ \mathsf{else}\ P_2]\!] = \mathit{exp}(e); (\mathcal{F}[\![P_1]\!] \wedge \mathcal{F}[\![P_2]\!])$

Figure 3. First framework

3.1 Simple Control Structures

Our first framework (Figure 3) treats a subset of the full language, programs with only sequencing and conditionals. $\mathcal{F}$ assigns an element of *DFFun* to every program. We use semi-colon (;) for function composition in diagrammatic order. The meet ($\wedge$) operation on functions is defined pointwise, and *id* is the identity function in *DFFun*. *asgn* and *exp* are the only functions specific to a particular analysis. The types of all the names appearing in this definition are:

id : *DFFun*
asgn : Var × Exp → *DFFun*
exp : Exp → *DFFun*
; : *DFFun* × *DFFun* → *DFFun*
$\wedge$: *DFFun* × *DFFun* → *DFFun*

We earlier stated that we allow only $\top$-preserving functions in *DFFun*. The identity function has this property, and function composition and meet preserve it, so we need only to confirm it for *asgn* and *exp* for each analysis.

To get the result of the static analysis of P, apply $\mathcal{F}[\![P]\!]$ to an appropriate initial value.

As an example, we define an analysis for variable initialization. Here, $Data = \mathcal{P}(Var)^2$, with ordering

$$(D, U) \sqsubseteq (D', U') \text{ if } D \subseteq D' \text{ and } U \supseteq U'$$

The datum (D, U) entering a node means that D is the set of variables that definitely have definitions at this point, and U is the set that may have been used without definition.

$$\mathit{asgn}(x, e) = \lambda(D, U).(D \cup \{x\}, (vars(e) \setminus D) \cup U)$$
$$\mathit{exp}(e) = \lambda(D, U).(D, (vars(e) \setminus D) \cup U)$$

$vars(e)$ is the set of variables occurring in e. It is easy to see that *asgn*(x,e) and *exp*(e) preserve $\top_{\mathit{Data}}$ (the pair $(Var, \emptyset)$).

Returning to the general case, our task is to find representations of elements of *DFFun* for each analysis.

Definition Suppose R is a set with the following values and functions:

$$\begin{array}{lcl} \top_R & : & R \\ \mathit{id}_R & : & R \\ \mathit{asgn}_R & : & \mathrm{Var} \times \mathrm{Exp} \to R \end{array} \qquad \begin{array}{lcl} \mathit{exp}_R & : & \mathrm{Exp} \to R \\ ;_R & : & R \times R \to R \\ \wedge_R & : & R \times R \to R \end{array}$$

R is an *adequate representation* of a dataflow problem if there is a homomorphism **abs** : $R \to$ *DFFun*. Specifically, this requires (unsubscripted names are the values in *DFFun*)

$$\begin{aligned} \mathbf{abs}(\top_R) &= \lambda d.\top_{\mathit{Data}} \\ \mathbf{abs}(\mathit{id}_R) &= \mathit{id} \\ \mathbf{abs}(\mathit{asgn}_R(x, e)) &= \mathit{asgn}(x, e) \\ \mathbf{abs}(\mathit{exp}_R(e)) &= \mathit{exp}(e) \\ \mathbf{abs}(r\, ;_R r') &= \mathbf{abs}(r); \mathbf{abs}(r') \\ \mathbf{abs}(r \wedge_R r') &= \mathbf{abs}(r) \wedge \mathbf{abs}(r') \end{aligned}$$

($\top_R$ is not used until the next subsection.)

Definition $\mathcal{R}$: Pgm $\to R$ is the function defined as in Figure 3, but with all $\mathcal{F}$'s replaced by $\mathcal{R}$ and all operations replaced by their "sub-R" versions.

Theorem If R is an adequate representation, then for all P, $\mathbf{abs}(\mathcal{R}[\![P]\!]) = \mathcal{F}[\![P]\!]$.

Proof Trivial.

For uninitialized variables, a natural representation, which is also adequate, is almost the same as $Data$:

$$R = \mathcal{P}(Var)^2 \cup \{\top_R\}$$

For any fragment P, $\mathcal{R}[\![P]\!]$ is the pair containing the set of variables definitely defined in P and the set possibly used without definition in P. Then,

$$\mathbf{abs}(D, U) = \lambda(D', U').(D' \cup D, U' \cup (U \setminus D'))$$

($\mathbf{abs}(\top_R)$ necessarily equals $\lambda d.\top_{\mathit{Data}}$, as required by the definition of adequacy.)

The operations on this representation are[1]

$$\mathit{id}_R = (\emptyset, \emptyset)$$
$$\mathit{asgn}_R(x, e) = (\{x\}, vars(e))$$
$$\mathit{exp}_R(e) = (\emptyset, vars(e))$$
$$(D, U) ;_R (D', U') = (D \cup D', U \cup (U' \setminus D))$$
$$(D, U) \wedge_R (D', U') = (D \cap D', U \cup U')$$

To illustrate, we show a program annotated with the value of $\mathcal{R}[\![P]\!]$ for each subtree P:

```
// ({x, y}, {x, z}) (entire fragment)
y = x;              // ({y}, {x})
if (z > 10)         // ({x}, {x, y, z}) (‘if’ statement)
{                   // ({x, w}, {x, y}) (‘true’ branch)
    w = 15;         // ({w}, ∅)
    x = x + y + w; }        // ({x}, {x, y, w})
else
    x = 0;          // ({x}, ∅)
```

In Figure 2, we included while statements in our language. They can be defined using a maximal fixpoint in the usual way:

[1] Throughout the paper, to avoid clutter, we ignore $\top$ when defining functions; in every case, the definitions of *asgn*(x, e), *exp*(e), $;_R$, $\wedge_R$, and **abs** should check for $\top$ and return it.

$$\mathcal{F}[\![\mathsf{while}\ e\ \mathsf{do}\ P]\!] = \mathit{mfxp}(\lambda p.\mathit{exp}(e); (\mathcal{F}[\![P]\!]; p \wedge \mathit{id}))$$

We cannot define $\mathcal{R}[\![\mathsf{while}\ e\ \mathsf{do}\ P]\!]$ in this way, because R is not a partial order. However, in all of our analyses — and most static analyses — this fixpoint converges in a fixed number of iterations. Thus, $\mathcal{F}[\![\mathsf{while}\ e\ \mathsf{do}\ P]\!]$ will be equal to the first element of the list $\mathit{exp}(e)$, $\mathit{exp}(e); \mathcal{F}[\![P]\!]; \mathit{exp}(e)$, $\mathit{exp}(e); \mathcal{F}[\![P]\!]; \mathit{exp}(e); \mathcal{F}[\![P]\!]; \mathit{exp}(e)$, ... that is equal to the next element, and the corresponding element of R will represent it. We might only add that the iteration bound is another parameter to the analysis that can vary among analyses.

In principle, we could now move on to staging, using $\mathcal{R}$ to calculate the representation of fragments. In practice, we calculate them by using the definition of $\mathcal{F}$. This method will turn out, in the following sections, to be more efficient.

Definition $\mathcal{F}^R$: Pgm $\rightarrow R \rightarrow R$ is the function defined as in Figure 3, but with all occurrences of $\mathcal{F}$ replaced by $\mathcal{F}^R$, and operations defined as follows:

$$\begin{aligned}
&\mathit{id}^R = \mathit{id}\\
&\mathit{asgn}^R(x,e) = \lambda r.r\ ;_R\ \mathit{asgn}_R(x,e)\\
&\mathit{exp}^R(e) = \lambda r.r\ ;_R\ \mathit{exp}_R(x,e)\\
&f\ ;^R\ g = f; g\\
&f \wedge^R g = \lambda r.fr \wedge_R gr
\end{aligned}$$

Definition $r \equiv r'$ if $\mathbf{abs}(r) = \mathbf{abs}(r')$.

Theorem If R is adequate, then for all P and r, $\mathcal{F}^R[\![P]\!]r \equiv r\ ;_R\ \mathcal{R}[\![P]\!]$.

Proof The proof is a straightforward induction on P.

Corollary $\mathcal{F}^R[\![P]\!]\mathit{id}_R \equiv \mathcal{R}[\![P]\!]$.

If **abs** is injective — in which case we call R an *exact representation* — then we can replace $\equiv$ by $=$ in the above theorems. All the analyses we define in this paper are exact.

We are now ready to stage static analyses. The first stage calculates values of R, and the second, run-time, stage uses $\mathcal{F}$ to complete the analysis:

Static stage : For every fragment P, analyze every maximal hole-free subtree T by computing $\mathcal{F}^R[\![T]\!]\mathit{id}_R$.

Dynamic stage : Holes will be filled "bottom-up," so all fragments will have their holes filled before they themselves can be plugs. Thus, plugs are hole-free. After filling in the holes in P with plugs $Q_1, ..., Q_n$, we have an AST in which some nodes have already been annotated with representations (namely, the hole-free subtrees of P and the plugs $Q_1, ..., Q_n$). We can now calculate $\mathcal{F}[\![P[Q_1, ..., Q_n]]\!]$ with the appropriate initial value, but without traversing subtrees of the nodes that are already analyzed. It is in this exception that staging has its benefit.

3.2 Break Statements

We expand our analysis now to labelled statements and break-to-label statements. The idea is this: Since a break results in transfer of control to the end of a labelled statement, we treat it as the meet of the statements up to the break with the normal exit from the labelled statement (and with all other breaks to this label). We will see that an adequate representation in the sense of the previous section can be extended uniformly to a representation for this case.

$$\begin{aligned}
&\mathcal{F}[\![\mathsf{skip}]\!] = \mathit{id}\\
&\mathcal{F}[\![x = e]\!] = \lambda(\eta,d).(\eta, \mathit{asgn}(x,e)(d))\\
&\mathcal{F}[\![\mathsf{break}\ \ell;]\!] = \lambda(\eta,d).(\eta[\ell \mapsto d \wedge \eta(\ell)], \top_{\mathit{Data}})\\
&\mathcal{F}[\![\ell : P]\!] = \lambda(\eta,d).\ \mathsf{let}\ (\eta_1,d_1) \leftarrow \mathcal{F}[\![P]\!](\eta,d)\\
&\qquad\qquad\qquad\quad \mathsf{in}\ (\eta_1[\ell \mapsto \top_{\mathit{Data}}], d_1 \wedge \eta_1(\ell))\\
&\mathcal{F}[\![P_1;P_2]\!] = \mathcal{F}[\![P_1]\!];\mathcal{F}[\![P_2]\!]\\
&\mathcal{F}[\![\mathsf{if}\ (e)\ P_1\ \mathsf{else}\ P_2]\!] = \lambda(\eta,d).\ \mathsf{let}\ (\eta_1,d_1) \leftarrow \mathcal{F}[\![P_1]\!](\eta, \mathit{exp}(e)(d))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\eta_2,d_2) \leftarrow \mathcal{F}[\![P_2]\!](\eta, \mathit{exp}(e)(d))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ (\eta_1,d_1) \wedge (\eta_2,d_2)
\end{aligned}$$

Figure 4. Framework with break statements

Throughout this section and the next, we assume all programs are legal in the sense that they do not contain nested labelled statements with the same label.

An *environment* η is a function in Env = Label $\rightarrow$ Data. Now the incoming and outgoing values are pairs:

$$\mathcal{F}[\![P]\!] : \mathit{Env} \times \mathit{Data} \rightarrow \mathit{Env} \times \mathit{Data}$$

The extended analysis is shown in Figure 4. *asgn* and *exp* have the same types as in the previous section; semi-colon is again function composition (in the expanded space), and *id* is the identity function. We extend meet to environments element-wise and then to pairs component-wise.

A word of explanation is in order about labelled statements and breaks. Suppose a statement P is contained within a labelled statement with label L, and we are evaluating $\mathcal{F}[\![P]\!](\eta,d)$. d contains information about the control flow paths that reach P. η contains information about all the control flow paths that were terminated with a break L statement prior to reaching P; since there may be more than one, $\eta(L)$ gives a conservative approximation by taking the meet of all those paths. Thus, if P is break L, then d is incorporated into the outgoing environment by taking $d \wedge \eta(L)$. Furthermore, the "normal exit" from P is $\top_{\mathit{Data}}$. For any statement Q, $\mathcal{F}[\![Q]\!]$ preserves $\top_{\mathit{Data}}$ in its second argument. so any statements directly following P will be ignored. For labelled statements, $\mathcal{F}[\![L : P]\!](\eta,d)$ first calculates $\mathcal{F}[\![P]\!](\eta,d)$. It is important that $\eta(L) = \top_{\mathit{Data}}$; this will be the case because (a) the initial value for any analysis has the environment $\lambda\ell.\top_{\mathit{Data}}$, (b) $L : P$ is legal, so it cannot be within another statement labelled L, and (c) $\mathcal{F}[\![L : P]\!](\eta,d)$ returns an environment in which L is reset to $\top_{\mathit{Data}}$.

Representations of these functions are derived from representations of functions in *DFFun*. Assume R is an adequate representation of *DFFun*. It can be extended to a representation E_R of functions in the space $\mathit{Env} \times \mathit{Data} \rightarrow \mathit{Env} \times \mathit{Data}$. Define Env_R = Label $\rightarrow R$. Then

$$
\begin{aligned}
&\mathcal{R}[\![\textsf{skip}]\!] = (\top_{Env_R}, \mathrm{id}_R)\\
&\mathcal{R}[\![x \texttt{=} e]\!] = (\top_{Env_R}, \mathrm{asgn}_R(x, e))\\
&\mathcal{R}[\![\textsf{break } \ell;]\!] = (\top_{Env_R}[\ell \mapsto \mathrm{id}_R], \top_R)\\
&\mathcal{R}[\![\ell : P]\!] = \text{let } (\eta, r) \leftarrow \mathcal{R}[\![P]\!]\\
&\qquad\qquad \text{in } (\eta[\ell \mapsto \top_R], r \wedge_R \eta(\ell))\\
&\mathcal{R}[\![P_1; P_2]\!] = \text{let } (\eta_1, r_1) \leftarrow \mathcal{R}[\![P_1]\!], (\eta_2, r_2) \leftarrow \mathcal{R}[\![P_2]\!]\\
&\qquad\qquad\quad \text{in } (\eta_1 \wedge_R (r_1;_R \eta_2), r_1;_R r_2)\\
&\mathcal{R}[\![\textsf{if } (e) \ P_1 \ \textsf{else } P_2]\!] = \text{let } (\eta_1, r_1) \leftarrow \mathcal{R}[\![P_1]\!], (\eta_2, r_2) \leftarrow \mathcal{R}[\![P_2]\!]\\
&\qquad\qquad\qquad\qquad\qquad \text{in } \exp_R(e);_R ((\eta_1, r_1) \wedge_R (\eta_2, r_2))
\end{aligned}
$$

Figure 5. Representation for framework of Figure 4

$$E_R \ = \ Env_R \times R$$

Figure 5 gives a function to calculate representations. Although very similar to $\mathcal{F}$, $\mathcal{R}$ has one crucial difference. For statement $P_1; P_2$, where $\mathcal{F}$ simply uses function composition, $\mathcal{R}$ calculates an explicit value. Of particular interest is the way environments are affected. The environment given by $\mathcal{R}[\![P_2]\!]$ incorporates all the control flow up to any break statements in P_2. The new environment augments each value in that environment by adding r_1, which is the dataflow information for a *normal* exit from P_1. That is, an abnormal exit is either an abnormal exit from P_1 or a normal exit from P_1 followed by an abnormal exit from P_2. Furthermore, if there is a break to the same label from both P_1 and P_2, the total effect is that two separate paths meet after the statement with that label, so the functions in the two environments are joined.

Defining the abstraction function:

$$
\begin{aligned}
&\mathbf{abs}_E : E_R \rightarrow (Env \times Data \rightarrow Env \times Data)\\
&\mathbf{abs}_E(\eta_R, r) \ = \ \lambda(\eta, d).(\lambda\ell.\eta(\ell) \wedge \mathbf{abs}(\eta_R(\ell))d, \mathbf{abs}(r)d)
\end{aligned}
$$

we have the following theorem.

Theorem If R is adequate, then for all programs P, $\mathbf{abs}_E(\mathcal{R}[\![P]\!]) = \mathcal{F}[\![P]\!]$.

Again, we can (and do) calculate $\mathcal{R}$ by reinterpreting $\mathcal{F}$ using the operators of R. The function

$$\mathcal{F}^R : \text{Pgm} \rightarrow E_R \rightarrow E_R$$

is defined the same way as $\mathcal{F}$ but with the "super-R" versions of the operators. $asgn^R$ and exp^R are exactly the same as in the previous section; id^R, $;^R$, and $\wedge^R$ have the same definitions but different types; $\top^R$ is $\top_R$.

Theorem Given P, let (η, r) = $\mathcal{R}[\![P]\!]$. For all η', r', as long as $\eta'(L) = \top_R$ for any label L that occurs in P,

$$\mathcal{F}^R[\![P]\!](\eta', r') \equiv (\lambda\ell.\eta'(\ell) \wedge_R (r' ;_R \eta(\ell)), r' ;_R r).$$

Corollary $\mathcal{F}^R[\![P]\!](\top_{Env_R}, id_R) \equiv \mathcal{R}[\![P]\!]$.

Again, $\equiv$ can be replaced by $=$ for all the analyses we present in this paper. The value of this theorem is more easily seen now than in the previous section. $\mathcal{R}$ contains a fundamental inefficiency in the calculation of environments in $\mathcal{R}[\![P_1; P_2]\!]$. Because this involves modifying *all* the values given in the environment of P_2, it can lead to quadratic behavior for a sequence of statements each of which contains a break statement. (The effect is far worse, in practice, in Section 3.3, where the dataflow functions for *every node* in P_2 need to be modified.) $\mathcal{F}$ does not have this inefficiency. There, the environments are threaded through the program, so a break statement causes the environment to be updated just once, and the value placed there is never changed.

Adding a break statement to our previous example, we show the values of $\mathcal{F}^R[\![P]\!](\top_{Env_R}, (\emptyset, \emptyset))$ for each node P.

```
// ({L ↦ ({x, y}, {x, z})}, ({x, w, y}, {x, z}))
y = x;              // (∅, ({y}, {x}))
if (z > 10)         // ({L ↦ ({x}, {z})}, ({x, w}, {x, y, z}))
{                   // (∅, ({x, w}, {x, y}))
    w = 15;         // (∅, ({w}, ∅))
    x = x + y + w; }    // (∅, ({x}, {x, y, w}))
else
{                   // ({L ↦ ({x}, ∅)}, ⊤)
    x = 0;          // (∅, ({x}, ∅))
    break L; }      // ({L ↦ (∅, ∅)}, ⊤)
```

The approach to staging is unchanged.

3.3 The Framework

The frameworks described so far lack one important ingredient: they do not give us information about each node in the AST, but only about the root node of the AST. Most static analyses are used to obtain information at each node: What definitions reach this particular node? What variables have constant values at this particular point in the program?

The complete analysis returns a map giving data at each node. Assuming each node is uniquely identified by an element of $Node$, we define $NodeMap \ = \ Node \nrightarrow Data$ (partial functions from $Node$ to $Data$). Now,

$$\mathcal{F}[\![P]\!] : NodeMap \times Env \times Data \rightarrow NodeMap \times Env \times Data$$

We also change the type of *asgn*:

$$asgn : Node \times \text{Var} \times \text{Exp} \rightarrow DFFun$$

for cases (such as reaching definitions) where $Node$ is contained within $Data$. In most cases, such as uninitialized variables, the first argument is ignored.

The full forward analysis is shown in Figure 6.

As in the previous section, we can start with an adequate representation and create a representation for this analysis. Specifically, define

$$F_R = (Node \nrightarrow R) \times Env_R \times R$$

The abstraction function becomes:

$$
\begin{aligned}
\mathbf{abs}_F : F_R \rightarrow (&NodeMap \times Env \times Data \rightarrow\\
&NodeMap \times Env \times Data)\\
\mathbf{abs}_F(\varphi_R, \eta_R, r) \ = \ &\lambda(\varphi', \eta', d').(\varphi' \cup (\lambda n.\mathbf{abs}(\varphi_R(n))d'),\\
&\qquad \lambda\ell.\eta'(\ell) \wedge \mathbf{abs}(\eta_R(\ell))d',\\
&\qquad \mathbf{abs}(r)d')
\end{aligned}
$$

Representations are calculated by function $\mathcal{R}$ (Figure 7).

$$
\begin{aligned}
&\mathcal{F}\llbracket n : \mathsf{skip;}\, \rrbracket = \lambda(\varphi,\eta,d).(\varphi[n \mapsto d],\eta,d) \\
&\mathcal{F}\llbracket n : x \,\mathsf{=}\, e;\, \rrbracket = \lambda(\varphi,\eta,d).\text{let } d' \leftarrow \mathit{asgn}(n,x,e)(d) \\
&\qquad\qquad\qquad\qquad\quad \text{in } (\varphi[n \mapsto d'],\eta,d') \\
&\mathcal{F}\llbracket n : \mathsf{break}\ \ell;\, \rrbracket = \lambda(\varphi,\eta,d).(\varphi[n \mapsto \top_{\mathit{Data}}], \\
&\qquad\qquad\qquad\qquad\qquad \eta[\ell \mapsto d \wedge \eta(\ell)], \top_{\mathit{Data}}) \\
&\mathcal{F}\llbracket n : (\ell : (n_1 : P)) \rrbracket = \\
&\quad \lambda(\varphi,\eta,d).\ \text{let } (\varphi_1,\eta_1,d_1) \leftarrow \mathcal{F}\llbracket n_1 : P \rrbracket(\varphi,\eta,d) \\
&\qquad\qquad\quad \text{in } (\varphi_1[n \mapsto d_1 \wedge \eta_1(\ell)], \eta_1[\ell \mapsto \top_{\mathit{Data}}], d_1 \wedge \eta_1(\ell)) \\
&\mathcal{F}\llbracket n : (n_1 : P_1; \quad n_2 : P_2) \rrbracket = \\
&\quad \lambda(\varphi,\eta,d).\ \text{let } (\varphi_1,\eta_1,d_1) \leftarrow \mathcal{F}\llbracket n_2 : P_2 \rrbracket(\mathcal{F}\llbracket n_1 : P_1 \rrbracket(\varphi,\eta,d)) \\
&\qquad\qquad\quad \text{in } (\varphi_1[n \mapsto d_1], \eta_1, d_1) \\
&\mathcal{F}\llbracket n : \mathsf{if}(e)\ n_1 : P_1\ \mathsf{else}\ n_2 : P_2 \rrbracket = \\
&\quad \lambda(\varphi,\eta,d).\ \text{let } (\varphi_1,\eta_1,d_1) \leftarrow \mathcal{F}\llbracket n_1 : P_1 \rrbracket(\varphi,\eta,\mathit{exp}(e)(d)) \\
&\qquad\qquad\qquad\quad (\varphi_2,\eta_2,d_2) \leftarrow \mathcal{F}\llbracket n_2 : P_2 \rrbracket(\varphi,\eta,\mathit{exp}(e)(d)) \\
&\qquad\qquad\quad \text{in } ((\varphi_1 \cup \varphi_2)[n \mapsto d_1 \wedge d_2], \eta_1 \wedge \eta_2, d_1 \wedge d_2)
\end{aligned}
$$

Figure 6. Forward analysis framework

$$
\begin{aligned}
&\mathcal{R}\llbracket n : \mathsf{skip} \rrbracket = (\{n \mapsto \mathit{id}_R\}, \top_{Env_R}, \mathit{id}_R) \\
&\mathcal{R}\llbracket n : x \,\mathsf{=}\, e \rrbracket = (\{n \mapsto \mathit{asgn}_R(n,x,e)\}, \top_{Env_R}, \mathit{asgn}_R(n,x,e)) \\
&\mathcal{R}\llbracket n : \mathsf{break}\ \ell;\, \rrbracket = (\{n \mapsto \top_R\}, \top_{Env_R}[\ell \mapsto \mathit{id}_R], \top_R) \\
&\mathcal{R}\llbracket n : (\ell : n_1 : P) \rrbracket \,\mathsf{=}\, \text{let } (\varphi,\eta,r) \leftarrow \mathcal{R}\llbracket P \rrbracket \\
&\qquad\qquad\qquad\qquad \text{in } (\varphi[n \mapsto r \wedge_R \eta(\ell)], \eta[\ell \mapsto \top_R], r \wedge_R \eta(\ell)) \\
&\mathcal{R}\llbracket n : (n_1 : P_1; n_2 : P_2) \rrbracket \,\mathsf{=} \\
&\quad \text{let } (\varphi_1,\eta_1,r_1) \leftarrow \mathcal{R}\llbracket P_1 \rrbracket,\ (\varphi_2,\eta_2,r_2) \leftarrow \mathcal{R}\llbracket P_2 \rrbracket \\
&\quad \text{in } \ (\lambda n'.\ \text{if } \varphi_1(n') \text{ defined then } \varphi_1(n') \\
&\qquad\qquad\quad \text{if } \varphi_2(n') \text{ defined then } r_1;_R \varphi_2(n') \\
&\qquad\qquad\quad \text{if } n' = n \text{ then } r_1;_R r_2\ , \\
&\qquad\ \ \eta_1 \wedge_R (r_1;_R \eta_2), \\
&\qquad\ \ r_1;_R r_2) \\
&\mathcal{R}\llbracket n : \mathsf{if}\ (e)\ n_1 : P_1\ \mathsf{else}\ n_2 : P_2 \rrbracket = \\
&\quad \text{let } (\varphi_1,\eta_1,r_1) \leftarrow \mathcal{R}\llbracket P_1 \rrbracket,\ (\varphi_2,\eta_2,r_2) \leftarrow \mathcal{R}\llbracket P_2 \rrbracket \\
&\quad \text{in } \ (\mathit{exp}_R(e);_R ((\varphi_1 \cup \varphi_2)[n \mapsto (r_1 \wedge_R r_2)]), \\
&\qquad\ \ \mathit{exp}_R(e);_R (\eta_1 \wedge_R \eta_2), \\
&\qquad\ \ \mathit{exp}_R(e);_R (r_1 \wedge_R r_2))
\end{aligned}
$$

Figure 7. Representation for framework of Figure 6.

Theorem If R is adequate, then for all programs P, $\mathbf{abs}_F(\mathcal{R}\llbracket P \rrbracket) = \mathcal{F}\llbracket P \rrbracket$.

We can define $\mathcal{F}^R$ as in previous sections, and obtain

Theorem Let $(\varphi,\eta,r) = \mathcal{R}\llbracket P \rrbracket$. Then for all φ', η', r',

$$
\begin{aligned}
&\mathcal{F}^R\llbracket P \rrbracket(\varphi',\eta',r') \equiv \\
&\quad (\varphi' \cup \lambda.r'\,;_R \varphi(n), \lambda l.\eta(l) \wedge_R (r'\,;_R \eta(l)), r'\,;_R r)
\end{aligned}
$$

Our previous example with numbered nodes is in Figure 8. We show the value of function $\mathcal{R}\llbracket P \rrbracket$ only at the top node. The environment and data values are just as in Section 3.2: $\{\mathtt{L} \mapsto (\{\mathtt{x},\mathtt{y}\},\{\mathtt{x},\mathtt{z}\})\}$ and $(\{\mathtt{x},\mathtt{w},\mathtt{y}\},\{\mathtt{x},\mathtt{z}\})$, respectively. The node map is:

$$
\begin{aligned}
\{\ & n_1 \mapsto (\{\mathtt{x},\mathtt{w},\mathtt{y}\},\{\mathtt{x},\mathtt{z}\}), && n_2 \mapsto (\{\mathtt{y}\},\{\mathtt{x}\}), \\
& n_3 \mapsto (\{\mathtt{x},\mathtt{w},\mathtt{y}\},\{\mathtt{x},\mathtt{z}\}), && n_4 \mapsto (\{\mathtt{x},\mathtt{w},\mathtt{y}\},\{\mathtt{x},\mathtt{z}\}), \\
& n_5 \mapsto \top_R, && n_6 \mapsto (\{\mathtt{w},\mathtt{y}\},\{\mathtt{x},\mathtt{z}\}), \\
& n_7 \mapsto (\{\mathtt{x},\mathtt{w},\mathtt{y}\},\{\mathtt{x},\mathtt{z}\}), && n_8 \mapsto (\{\mathtt{x},\mathtt{y}\},\{\mathtt{x},\mathtt{z}\}),\ n_9 \mapsto \top_R\ \}
\end{aligned}
$$

```
n1:  // entire fragment
n2:  y = x;
n3:  if (z > 10)
n4:  {
n6:       w = 15;
n7:       x = x + y + w; }
     else
n5:  {
n8:       x = 0;
n9:       break L; }
```

Figure 8. The example program with numbered nodes.

Note that the values associated with the nodes are different from those in the previous analyses. This node map incorporates what is known about each node *at the top node* (as in [17]). For example, when we get through node n_6, we will have defined `w` and `y`, and will have used `x` and `z` possibly without definition. Thus, suppose we put this fragment into a hole at a position where `x` has been defined. We can look at, for example, node n_6 and immediately find that only `z` may have been used without definition. Note also that the fragment as a whole definitely defines `w`, even though it is only defined in one branch of the conditional; since the false branch ends in a break, control can only reach the end of this statement by taking the true branch.

Thus, we can analyze selected nodes without analyzing the entire tree, which can have a salutary effect on the runtime performance of the analysis.

Again, staging is not fundamentally different in this more complicated framework. One new wrinkle is that a single plug cannot be used in to fill two holes because its node names would then not be unique in the larger AST; thus, nodes in plugs need to be uniformly renamed before insertion in a larger tree, a process that is easily done.

4. Adequate Representations

We now present several analyses. Like variable initialization, all the representations we present here are exact.

4.1 Reaching Definitions I

The reaching definitions at a point in a program include any assignment statement which may have been the most recent assignment to a variable prior to this point.

$$\mathit{Data} = \mathcal{P}(\text{Node}) \cup \{\top\}$$

Sets in Data are ordered by reverse inclusion, with $\emptyset$ being the element just below $\top$. The operations are

$$
\begin{aligned}
&\mathit{asgn}(n,x,e) = \lambda D.(D \setminus D_x) \cup \{n\} \\
&\mathit{exp}(e) = \lambda D.D
\end{aligned}
$$

where D_x means the definitions of x. The representation is:

$$R = (\mathcal{P}(\text{Var}) \times \mathcal{P}(\text{Node})) \cup \{\top_R\}$$

If $\mathcal{R}[\![P]\!] = (V, N)$, V are all the variables defined in P and N are the assignment statements that define those variables and may reach the end of P.

$$\begin{array}{l}
\mathit{id}_R = (\emptyset, \emptyset) \\
\mathit{asgn}_R(n, x, e) = (\{x\}, \{n\}) \\
\mathit{exp}_R(e) = (\emptyset, \emptyset) \\
(K_1, G_1);_R (K_2, G_2) = (K_1 \cup K_2, G_2 \cup (G_1 \setminus K_2)) \\
(K_1, G_1) \wedge_R (K_2, G_2) = (K_1 \cap K_2, G_1 \cup G_2)
\end{array}$$

$$\mathbf{abs}(K, G) = \lambda D.\, G \cup (D \setminus K)$$

where $G \backslash K = \{n \in G \,|\, n \text{ is the definition of some } x \in K\}$.

4.2 Available Expressions

Available expressions are those expressions that have been previously computed, such that no intervening assignment has made their value obsolete. A given statement makes some expressions available, kills some expressions (by assigning to the variables they contain), and lets others pass through unmolested.

$$\mathit{Data} = \mathcal{P}(\text{Exp}) \cup \{\top\}$$

Sets in Data are ordered by set inclusion.

$$\begin{array}{l}
\mathit{asgn}(n, x, e) = \lambda E.(E \cup \{e' \,|\, e' \in \mathit{sub}(e)\}) \setminus E_x \\
\mathit{exp}(e) = \lambda E.E \cup \{e' \,|\, e' \in \mathit{sub}(e)\}
\end{array}$$

where E_x is the set of expressions that contain x and $\mathit{sub}(e)$ is the set of all subexpressions of e.

The following seems an obvious representation.

$$R = (\mathcal{P}(\text{Var}) \times \mathcal{P}(\text{Exp})) \cup \{\top_R\}$$

The value (V, E) represents that E is the set of expressions made available by a statement, and V is the set of variables defined by that statement (so that the statement kills any expressions containing those variables).

$$\begin{array}{l}
\mathit{id}_R = (\emptyset, \emptyset) \\
\mathit{asgn}_R(n, x, e) = (\{x\}, \{e' \,|\, e' \in \mathit{sub}(e), x \notin \mathit{vars}(e')\} \\
\mathit{exp}_R(e) = (\emptyset, \{e' \,|\, e' \in \mathit{sub}(e)\}) \\
(K_1, G_1);_R (K_2, G_2) = (K_1 \cup K_2, G_2 \cup (G_1 \setminus K_2)) \\
(K_1, G_1) \wedge_R (K_2, G_2) = (K_1 \cup K_2, G_1 \cap G_2)
\end{array}$$

$$\mathbf{abs}(K, G) = \lambda E.G \cup (E \setminus K)$$

where $G \backslash K = \{e \in G \,|\, \text{none of the variables in } e \text{ occur in } K\}$.

However, this is *not* an adequate representation for the analysis. Consider the statement: if (cond) $\{a$ = $\ldots$; $\ldots$ = $a + b\}$ else $\{\}$. Suppose that $a + b$ is available before this statement. It will also be available afterwards. However, since there is an assignment to a in one branch, the statement kills any expression containing a. Furthermore, $a + b$ is not generated in the other branch. Thus, the only R value that we could assign to this if-statement is $(\{a\}, \emptyset)$. But this will kill the incoming definition of $a + b$.

To obtain an adequate representation, we need to record that some expressions are guaranteed to survive a statement, even if they contain variables that are in its kill set, while others will be killed, as usual. We do this by putting annotations on expressions in the available set:

Definition For set S, $S_{Annot} = \{s_{must} \,|\, s \in S\} \cup \{s_{sur} \,|\, s \in S\}$. Also define the operation "." on annotations: $\mathit{must}.\mathit{must} = \mathit{must}$ and otherwise $a.a' = \mathit{sur}$.

Then, this analysis is defined as follows:

$$R = \mathcal{P}(\text{Var}) \times \mathcal{P}(\text{Exp}_{Annot}) \cup \{\top_R\}$$

$$\begin{array}{l}
\mathit{id}_R = (\emptyset, \emptyset) \\
\mathit{asgn}_R(n, x, e) = (\{x\}, \{e'_{must} \,|\, e' \in \mathit{sub}(e), x \notin \mathit{vars}(e')\} \\
\mathit{exp}_R(e) = (\emptyset, \{e'_{must} \,|\, e' \in \mathit{sub}(e)\}) \\
(K_1, G_1);_R (K_2, G_2) = (K_1 \cup K_2, \\
\quad \{e_{must} \,|\, e_{must} \in G_2\} \cup \\
\quad \{e_m \,|\, e_{sur} \in G_2, e_m \in G_1\} \cup \\
\quad \{e_{sur} \,|\, e_{sur} \in G_2, e_m \notin G_1, \mathit{vars}(e) \cap K_1 = \emptyset\} \cup \\
\quad \{e_m \,|\, e_m \in G_1, e_n \notin G_2, \mathit{vars}(e) \cap K_2 = \emptyset\}) \\
(K_1, G_1) \wedge_R (K_2, G_2) = (K_1 \cup K_2, \\
\quad \{e_{m.n} \,|\, e_m \in G_1, e_n \in G_2\} \cup \\
\quad \{e_{sur} \,|\, e_m \in G_1, e_n \notin G_2, \mathit{vars}(e) \cap K_2 = \emptyset\} \cup \\
\quad \{e_{sur} \,|\, e_m \in G_2, e_n \notin G_1, \mathit{vars}(e) \cap K_1 = \emptyset\})
\end{array}$$

$$\begin{array}{l}
\mathbf{abs}(K, G) = \lambda E.\{e \,|\, e_{must} \in G\} \cup \\
\quad \{e \,|\, e_{sur} \in G, e \in E\} \cup \\
\quad \{e \,|\, e \in E, e_a \notin G, \mathit{vars}(e) \cap K = \emptyset\}
\end{array}$$

The most interesting case is in the definition of semicolon, when $e_{sur} \in G_2$ and $e \in G_1$ (with either annotation). In that case, e is included in the available set, *even if it is killed by* K_2. Looking again at the if statement we discussed above, the true branch gives $(\{a\}, \{(a + b)_{must}\})$, and the false branch gives $(\emptyset, \emptyset)$. The meet of these values is $(\{a\}, \{(a + b)_{sur}\})$. This value summarizes the effect of the if statement correctly: if $(a + b)_{must}$ is in the incoming available set, then it will be in the resulting available set.

Annotations are used again in the alternative representation for reaching definitions and for constant propagation.

4.3 Reaching Definitions II

Using annotations, we give an alternative representation for reaching definitions.

$$R = (\text{Var} \rightarrow \mathcal{P}(\mathit{Node})_{Annot}) \cup \{\top_R\}$$

$$\begin{array}{l}
\mathit{id}_R = \lambda v.\emptyset_{sur} \\
\mathit{asgn}(n, x, e) = (\lambda v.\emptyset_{sur})[x \mapsto \{n\}_{must}] \\
\mathit{exp}(e) = \lambda v.\emptyset_{sur}
\end{array}$$

$$\begin{array}{l}
S_1;_R S_2 = \lambda x.\text{let } p_m \leftarrow S_1(x), q_n \leftarrow S_2(x) \\
\qquad\qquad\quad \text{in if } n = \mathit{must} \text{ then } q_n \text{ else } (p \cup q)_m \\
S_1 \wedge_R S_2 = \lambda x.\text{let } p_m \leftarrow S_1(x), q_n \leftarrow S_2(x) \\
\qquad\qquad\quad \text{in } (p \cup q)_{m.n}
\end{array}$$

We assume that $S(x)$ defaults to $\emptyset_{sur}$

$$\begin{array}{l}
\mathbf{abs}(S) = \lambda D.\{n \in D \,|\, n : x = e \text{ and } S(x) = p_{sur}\} \cup \\
\qquad\qquad\quad \{n \in p \,|\, n : x = e \text{ and } S(x) = p_m\}
\end{array}$$

4.4 Constant Propagation

The framework can be instantiated for constant propagation with the following definitions.

$$Data = (\mathrm{Var} \to \mathbb{Z}_{\perp}^{\top}) \cup \{\top_R\}$$

of $Data$ is ordered under the usual pointwise ordering.

$$\begin{array}{l}
\mathit{asgn}(n, x, e) = \\
\qquad \lambda M.\text{if isConstant}(e, M) \text{ then } M[x \mapsto \text{consVal}(e, M)] \\
\qquad\qquad \text{else } M[x \mapsto \perp] \\
\mathit{exp}(e) = \lambda M.M
\end{array}$$

For the representation, R is a function giving values for variables. However, these values are actually sets of variables, integer literals, and binary expressions, meaning "the set will be reduced to a constant c, if every element it contains eventually reduces to the constant c". Using this set, we effectively delay the meet operation, and gradually complete it as information becomes available.

$$\begin{array}{l}
R = Var \to CS_{Annot} \\
CS = P(\mathit{Exp} \cup \{\perp\})
\end{array}$$

We assume that, for all $C \in CS$, if $\perp \in C$ then $C \equiv \{\perp\}$; if there exist two integers $i_1, i_2 \in C$ such that $i_1 \neq i_2$ then $C \equiv \{\perp\}$. In the following definitions, M_1 and $M_2 \in Data, C$ and $C' \in CS, m$ and $n \in Annot$.

$$\begin{array}{l}
\mathit{id}_R = \lambda v.\emptyset_{sur} \\
\mathit{asgn}_R(n, x, e) = (\lambda v.\emptyset_{sur})[x \mapsto \{e\}_{must}] \\
\mathit{exp}_R(e) = \lambda v.\emptyset_{sur} \\
M_1 \wedge_R M_2 = \lambda x.\text{let } C_m \leftarrow M_1(x), C'_n \leftarrow M_2(x) \\
\qquad\qquad\qquad\qquad \text{in } (C \cup C')_{m.n} \\
M_1;_R M_2 = \lambda x.\mathit{semicolon}(M_1, M_1(x), M_2(x))
\end{array}$$

$$\begin{array}{l}
\mathit{semicolon}(M, C_m, C'_{must}) = \mathit{update}(M, C')_{must} \\
\mathit{semicolon}(M, C_m, C'_{sur}) = (\mathit{update}(M, C') \cup C)_m
\end{array}$$

The function $\mathit{update}(M, C)$ checks the constant map M for each variable found in the elements of the set C, and if there exists a mapping in M for that variable, uses it to update C. For example, let $M(y) = \{w, z\}$, and $C = \{y + 1\}$. Then $\mathit{update}(M, C)$ returns $\{w + 1, z + 1\}$.

The **abs** function, where $i \in \mathbb{Z}$, is

$$\begin{array}{l}
\mathbf{abs}(M) = \lambda S.\lambda x.\text{ let } C_M \leftarrow \mathit{semicolon}(S, S(x), M(x)) \\
\qquad\qquad\qquad\qquad \text{in if } C = \{i\} \text{ then } i \text{ else } \perp
\end{array}$$

4.5 Type Checking

Type checking is the most complicated of our analyses (see [9] for a full presentation). It requires that the framework be extended to accommodate declarations and scopes:

$$\begin{array}{l}
\mathcal{F}[\![n : \text{int } x]\!] = \lambda(\varphi, \eta, d).\text{let } d' \leftarrow \mathit{intDecl}(n, x)(d) \\
\qquad\qquad\qquad\qquad\qquad \text{in } (\varphi[n \mapsto d'], \eta, d') \\
\mathcal{F}[\![n : \{n_1 : P\}]\!] = \\
\quad \lambda(\varphi, \eta, d).\text{let } \eta' \leftarrow \mathit{map}(\mathit{beginScope}, \eta), \\
\qquad\qquad\qquad (\varphi_1, \eta_1, d_1) \leftarrow \mathcal{F}[\![n_1 : P]\!](\varphi, \eta', \mathit{beginScope}(n, d)) \\
\qquad\qquad \text{in let } d' \leftarrow \mathit{endScope}(d_1) \\
\qquad\qquad\quad \text{in } (\varphi_1[n \mapsto d'], \mathit{map}(\mathit{endScope}, \eta_1), d')
\end{array}$$

The $Data$ values consist of a stack of type environments, to accommodate different levels of scopes. In the lattice, a shorter stack appears below a longer one. If the stack frames are the same, ordering is done pairwise among the type environments kept in the frames.

$$\begin{array}{l}
Data = TySt \cup \{\textbf{error}\} \\
TySt = ((Node \cup \{\star\}) \times TyEv)^* \\
TyEv = Var \nrightarrow Type \\
Type = \{\textbf{int}, \textbf{bool}\}
\end{array}$$

$$\begin{array}{l}
\mathit{asgn}(n, x, e) = \lambda\Gamma.\text{ if } \mathit{type}(x, \Gamma) = \mathit{type}(e, \Gamma) \text{ then } \Gamma \text{ else } \textbf{error} \\
\mathit{intDecl}(n, x) = \lambda\Gamma.\text{ if } \Gamma(x) \text{ is defined then } \textbf{error} \\
\qquad\qquad\qquad\qquad\qquad \text{else } \mathit{add}(\Gamma, x, \textbf{int}) \\
\mathit{exp}(e) = \lambda\Gamma.\text{ if } \mathit{type}(e, \Gamma) = \textbf{bool} \text{ then } \Gamma \text{ else } \textbf{error} \\
\mathit{beginScope}(n, \Gamma) = [\Gamma, (n, \epsilon)] \\
\mathit{endScope}([\Gamma, (n, \gamma)]) = \Gamma
\end{array}$$

The star in $TySt$ denotes the initial frame of the stack.

Summarizing a node requires that we remember certain "proof obligations" which we may not be able to discharge until we have the entire program together. These obligations are of three kinds: ensuring that two variables have the same type; ensuring that a given variable has a given type; and ensuring that a variable is not being redeclared. An R value, in addition to a stack of type environments, consists of a set $Oblg$ which can carry the three kinds of obligations.

$$\begin{array}{l}
R = TySt \times \mathit{Oblg} \\
\mathit{Oblg} = \mathcal{P}(Var^2 \cup (Var \times \mathit{Type}) \cup Var) \cup \{\textbf{error}\}
\end{array}$$

The appearance of an expression or assignment statement generates a set of obligations:

$$\begin{array}{l}
\mathit{asgn}_R(n, x, e) = (\texttt{[]}, \mathit{mkOblg}(x, e)) \\
\mathit{exp}_R(e) = (\texttt{[]}, \mathit{mkOblg}(e, \textbf{bool})) \\
\mathit{intDecl}_R(n, x) = ([(\star, \epsilon[x \mapsto \textbf{int}])], \{x\})
\end{array}$$

where *mkOblg* is an overloaded function defined by:

$$\begin{array}{l}
\mathit{mkOblg}(x, y) = (x, y) \\
\mathit{mkOblg}(x, e_1 \oplus e_2) = \mathit{mkOblg}(e_1, \mathit{ltype}(\oplus)) \\
\qquad \sqcup\, \mathit{mkOblg}(e_2, \mathit{rtype}(\oplus)) \sqcup (x, \mathit{type}(\oplus)) \\
\mathit{mkOblg}(x, T) = (x, T) \\
\mathit{mkOblg}(e_1 \oplus e_2, T) = \mathit{mkOblg}(e_1, \mathit{ltype}(\oplus)) \\
\qquad \sqcup\, \mathit{mkOblg}(e_2, \mathit{rtype}(\oplus)) \\
\qquad \text{if } \mathit{type}(\oplus) = T \ (\textbf{error} \text{ otherwise})
\end{array}$$

where $\oplus$ denotes any binary operation. $\sqcup$ is union if both sides are not the special **error** value, but when one of the arguments is **error**, then the error value is propagated. *ltype*, *rtype*, *type* denote the *expected* type of the left argument, right argument, and return value, of the operator.

We define the meet and semicolon operations as

$$\begin{array}{l}
(\Gamma, \Delta);_R (\Gamma', \Delta') = \text{let } \Gamma'' := \mathit{sequence}(\Gamma, \Gamma') \\
\qquad\qquad\qquad\qquad\quad \Delta'' := \mathit{sequence}(\Delta, \Delta', \Gamma) \\
\qquad\qquad\qquad\quad \text{in } (\Gamma'', \Delta'')
\end{array}$$

where *sequence* $: TySt \times TySt \to TySt$ is

$$\mathit{sequence}(\Gamma_1, \Gamma_2) = \mathit{concatenate}(\Gamma_1, \Gamma_2)$$

$$
\begin{aligned}
&\mathcal{B}\llbracket \mathsf{skip};\rrbracket = \mathit{id}\\
&\mathcal{B}\llbracket x \texttt{ = } e;\rrbracket = \lambda(\eta,d).(\eta, \mathit{asgn}(x,e)(d))\\
&\mathcal{B}\llbracket \mathsf{break}\ \ell;\rrbracket = \lambda(\eta,d).(\eta,\eta(\ell))\\
&\mathcal{B}\llbracket \ell : P\rrbracket = \lambda(\eta,d).\ \mathsf{let}\ (\eta',d') \leftarrow \mathcal{B}\llbracket P\rrbracket(\eta[\ell\mapsto d],d)\\
&\qquad\qquad\qquad\qquad\quad \mathsf{in}\ (\eta'[\ell\mapsto \top_R], d')\\
&\mathcal{B}\llbracket P_1;P_2\rrbracket = \mathcal{B}\llbracket P_2\rrbracket;\mathcal{B}\llbracket P_1\rrbracket\\
&\mathcal{B}\llbracket \mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2\rrbracket = \lambda\eta d.\ \mathsf{let}\ (\eta_1,d_1)\leftarrow \mathcal{B}\llbracket P_1\rrbracket(\eta,d)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\eta_2,d_2)\leftarrow \mathcal{B}\llbracket P_2\rrbracket(\eta,d)\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \mathsf{in}\ (\eta, \mathit{exp}(e)(d_1\wedge d_2))
\end{aligned}
$$

Figure 9. Backward analysis framework

$$
\begin{aligned}
&\mathcal{R}\llbracket \mathsf{skip}\rrbracket = (\top_{Env_R}, \mathit{id}_R)\\
&\mathcal{R}\llbracket x \texttt{ = } e\rrbracket = (\top_{Env_R}, \mathit{asgn}_R(x,e))\\
&\mathcal{R}\llbracket \mathsf{break}\ \ell;\rrbracket = (\top_{Env_R}[\ell\mapsto \mathit{id}_R], \top_R)\\
&\mathcal{R}\llbracket \ell : P\rrbracket = \mathsf{let}\ (\eta,r)\leftarrow \mathcal{R}\llbracket P\rrbracket\\
&\qquad\qquad\quad\ \mathsf{in}\ (\eta[\ell\mapsto\top_R], r\wedge_R \eta(\ell))\\
&\mathcal{R}\llbracket P_1;P_2\rrbracket = \mathsf{let}\ (\eta_1,r_1)\leftarrow\mathcal{R}\llbracket P_1\rrbracket, (\eta_2,r_2)\leftarrow\mathcal{R}\llbracket P_2\rrbracket\\
&\qquad\qquad\qquad\ \mathsf{in}\ (\eta_1\wedge_R(\eta_2;_R r_1), r_2;_R r_1)\\
&\mathcal{R}\llbracket \mathsf{if}\ (e)\ P_1\ \mathsf{else}\ P_2\rrbracket = (\mathcal{R}\llbracket P_1\rrbracket\wedge_R\mathcal{R}\llbracket P_2\rrbracket);_R \mathit{exp}_R(e)
\end{aligned}
$$

Figure 10. Representation for framework of Figure 9.

and *sequence* : $Oblg \times Oblg \times TySt \to Oblg$ is

$$\mathit{sequence}(\Delta_1,\Delta_2,\Gamma) = \{\delta\ :\ \delta\in\Delta_1 \text{ or}, \delta\in\Delta_2 \text{ and } \Gamma\not\vdash\delta\}$$

For meet we have

$$(\Gamma,\Delta)\wedge_R(\Gamma',\Delta') = (\mathit{longestCommonPrefix}(\Gamma,\Gamma'), \Delta\cup\Delta')$$

Finally, in the **abs** function, if the obligations imposd by the plug are not satisfied by the incoming type stack, we return error, otherwise we just sequence the incoming type stack with the plug's.

$$
\begin{aligned}
\mathbf{abs}(\Gamma_R,\Delta_R) = \lambda\Gamma.\ &\text{let } (\Gamma',\Delta')\leftarrow(\Gamma,\emptyset);_R(\Gamma_R,\Delta_R)\\
&\text{in if } \Delta'=\emptyset \text{ then } \Gamma' \text{ else } \mathbf{error}
\end{aligned}
$$

5. Backward Analysis Framework

We can define a similar framework for backwards analysis, although break statements significantly complicate matters. Due to space constraints, we only provide the intermediate framework here. It is presented in Figure 9, and the representation in Figure 10. The abstraction function is

$$\mathbf{abs}_E(\eta_R, r) = \lambda(\eta,d).(\eta, \mathbf{abs}(r)(d)\wedge \bigwedge_{l\in\text{Label}} \mathbf{abs}(\eta_R(l))(\eta(l)))$$

Theorem For all P, if the *DFFun* functions are distributive (i.e. $f(d\wedge d') = f(d)\wedge f(d')$), $\mathbf{abs}_E(\mathcal{R}\llbracket P\rrbracket) = \mathcal{B}\llbracket P\rrbracket$.

For the full framework which builds a node map at the top node, the intermediate framework can again be extended naturally as in forward analysis (Figure 6). However, defining $\mathcal{R}$ is not that straightforward. We need to keep an environment for every node in the node-map. So the type of the representation function is

$$\mathcal{R} : \text{Pgm} \to (Node \to (Env_R \times R)) \times Env_R \times R$$

Analogous to how $\mathcal{R}\llbracket P_1;P_2\rrbracket$ in the forward representation function of Figure 7 updates the node-map for each node in P_1 and P_2, $\mathcal{R}\llbracket L : P\rrbracket$ and $\mathcal{R}\llbracket P_1;P_2\rrbracket$ in the full backward representation function update each mapping in their node-maps as well.

5.1 Live Variables

$$Data = (\mathcal{P}(\text{Var}))\cup\{\top\}$$

ordered by reverse set inclusion.

$$
\begin{aligned}
&\mathit{asgn}(n,x,e) = \lambda L.(L\setminus\{x\})\cup vars(e)\\
&\mathit{exp}(e) = \lambda L.L\cup vars(e)
\end{aligned}
$$

$$
\begin{aligned}
&R = \mathcal{P}(\text{Var})^2\\
&\mathit{asgn}_R(n,x,e) = (\{x\}, vars(e))\\
&\mathit{exp}_R(e) = (\emptyset, vars(e))
\end{aligned}
$$

Definitions of id_R, $;_R$, $\wedge_R$ and **abs** are the same as in reaching definitions I (Section 4.1).

5.2 Very Busy Expressions

The definitions, except the following, are the same as in available expressions.

$$
\begin{aligned}
&\mathit{asgn}(n,x,e) = \lambda E.(E\setminus E_x)\cup sub(e)\\
&\mathit{asgn}_R(n,x,e) = (\{x\}, \{e'_{must}\mid e'\in sub(e)\})
\end{aligned}
$$

6. Performance

We are interested in the *run-time* costs of two methods of doing static analysis. One method is to fill in the holes and analyze the complete program at run time (the *base analysis*); the other is to use our *staged analysis*.

The benchmarks we present are of two kinds: *artificial* benchmarks illustrate how performance is affected by specific features in a program; *realistic* benchmarks are program generators drawn from previous publications.

For some analyses, one needs only the dataflow information for the root node; examples are uninitialized variables and type-checking. For most, we need the information at many, though not necessarily all, nodes. (Note that the base case must visit every node at run-time, even if it is only interested in a subset.)

We implemented the framework in Java. In Table 1, we present the performance of three analyses, on a variety of benchmark programs, as ratios between the base and the staged analyses; higher numbers represent greater speedup. We run the experiments in three different Java runtime environments: Sun's HotSpot, GNU's libgcj, and Kaffe. For reaching definitions (RD) and constant propagation (CP), we perform the analysis at every assignment statement (roughly half the nodes in the programs). For type checking (TC), we analyze only the top node.

	HotSpot			libgcj			Kaffe		
Sample Program	RD	CP	TC	RD	CP	TC	RD	CP	TC
Big-plug	2.10	1.19	3.65	7.43	3.78	5.15	9.73	5.23	5.63
Small-plug-A	2.17	1.12	3.50	6.96	3.91	4.28	10.7	4.62	5.55
Small-plug-B	2.40	1.14	2.97	4.78	3.41	4.39	7.03	4.65	5.40
Two-plug	1.67	1.17	1.66	2.59	2.19	2.90	3.83	2.83	3.18
Fib1 ([7])	1.10	1.07	1.31	1.24	0.93	1.17	1.64	1.26	1.05
Fib2 ([7])	1.23	1.16	0.67	1.48	0.99	1.18	2.02	1.47	1.05
Sort ([5])	1.48	1.21	1.92	1.64	1.08	1.59	1.86	1.29	1.66
Huffman ([7])	1.11	1.29	0.30	1.04	0.93	1.02	1.31	1.30	0.95
Marshalling 1 ([2])	12.37	3.93	28.27	34.83	15.42	9.34	49.64	18.92	12.04
Marshalling 2 ([2])	2.01	1.75	16.01	1.83	1.33	1.86	2.59	2.27	1.47

Table 1. Benchmarking results. The numbers show the ratio of the *base case* to the *staged* case.

We briefly describe the benchmarks used in Table 1. **Big-plug** is a small program with one hole, filled in by a large plug. **Small-plug-A** is a large program with a hole near the beginning, filled in by a small plug. **Small-plug-B** is a large program with a hole near the end, filled in by a small plug. **Two-plug** is a medium-sized program with two holes, filled in by medium-sized plugs. **Fib1** and **Fib2** is two versions of a Fibonacci function divided into small pieces for exposition [7]. **Sort** is a generator that produces a sort function by inlining the comparison operation [5]. **Huffman** is a generator that turns a Huffman tree into a sequence of conditional statements [7]. **Marshalling 1** is part of a program that produces customized serializers in Java; characteristics much like Big-plug [2]. **Marshalling 2** is a different part of the same program; has many holes and many small plugs.

As often happens, the invented benchmark examples show the best performance improvements. Our approach does result in slow-downs in some cases; the worst cases are Fib2 and Huffman, both of which consist of many holes and small plugs. Overall, the results are quite promising.

7. Conclusions

We have presented a framework for static analysis of ASTs that allows these analyses to be staged (when the representations are adequate). The method has application to run-time program generation: by optimizing the static analysis of programs, it can speed up overall run-time code generation time.

We are aware that the kinds of analyses we have presented are not normally done on source code. One area for future work is to explore analyses that occur naturally at source level; the type-checking analysis is one example. Another is to adapt our approach to CFGs. However, CFGs with multiple exits are difficult to use as plugs. An alternative is to use an intermediate language that is itself structured.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.

[2] B. Aktemur, J. Jones, S. Kamin, L. Clausen. Optimizing Marshalling by Run-time Program Generation. *GPCE '05*, Tallinn, Estonia, 2005.

[3] C. Chambers. Staged compilation. *PEPM '02*, Portland, OR, USA, 2002.

[4] K. Czarnecki, J. O'Donnell, J. Striegnitz, W. Taha. DSL Implementation in MetaOCaml, Template Haskell, and C++. *DSPG '04*, Dagstuhl, Germany, 2004

[5] S. Kamin, M. Callahan, L. Clausen. Lightweight and Generative Components I: Source-Level Components. *GCSE '99*, Erfurt, Germany, 1999.

[6] S. Kamin, L. Clausen, A. Jarvis. Jumbo: run-time code generation for java and its applications. *CGO '03*, Washington, DC, USA, 2003.

[7] S. Kamin. Program generation considered easy. *PEPM '04*, Verona, Italy, 2004.

[8] S. Kamin, B. Aktemur, M. Katelman. Staging Static Analyses for Program Generation (Full Version). University of Illinois Technical Report, 2006.

[9] M. Katelman Staged Static Analyses and Run-time Program Generation M.S. Thesis, Computer Science Dept., Univ. of Illinois, 2006.

[10] R. Kramer, R. Gupta, M. Soffa. The Combining DAG: A Technique for Parallel Data Flow Analysis. *IEEE Trans. Parallel Distrib. Syst.* 5(8), 1994

[11] T. Marlowe, B. Ryder. An efficient hybrid algorithm for incremental data flow analysis. *POPL '90*, San Francisco, CA, USA, 1990

[12] Y. Oiwa, H. Masuhara, A. Yonezawa. Type safe dynamic code generation in java. *JSST Workshop on Programming and Programming Languages (PPL2001)*, 2001.

[13] M. Poletto, W. Hsieh, D. Engler, M. Kaashoek. C and tcc: a language and compiler for dynamic code generation. *ACM TOPLAS*, 21(2):324–369, 1999.

[14] T. Reps, S. Horwitz, M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. *POPL '95*, San Fransisco, CA, USA, 1995

[15] A. Rountev, S. Kagan, T. Marlowe. Interprocedural Dataflow Analysis in the Presence of Large Libraries. *CC '06*, Vienna, Austria, 2006

[16] B. Ryder, M. Paull. Incremental data-flow analysis algorithms. *ACM TOPLAS*, 10(1):1–50, 1988.

[17] M. Sharir, A. Pnueli. Two approaches to interprocedural dataflow analysis. In *Program Flow Analysis: Theory and Applications*, 189–234, 1981.

[18] F. Smith, D. Grossman, G. Morrisett, L. Hornof, T. Jim. Compiling for runtime code generation. Technical report, Department of Computer Science, Cornell University, 2000.