Staging Static Analyses for Program Generation *
(Extended Version)

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 pro-
grammer 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 over-
whelm 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.

1 Introduction

We are concerned here with languages in which code generators are specified by embedding quoted
program fragments within a larger program (the metaprogram) [6, 12, 13, 4]. These quoted frag-
ments 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 prop-
erties of generated programs can be inferred from the initial set of fragments? How quickly can the
generated program be generated? The latter is of most interest when program generation occurs at
run time.

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

*Partial support for this work was received from NSF under grant CCR-0306221. This technical report is the
extended version of the paper published in GPCE 2006 with the same title.
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 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.

This paper is an expanded version of [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 automaticaly
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 $F$ for calculating dataflow values for AST’s in a lattice $Data$.
2. Present a framework $R$ for calculating representations of dataflow functions, given an “ade-
   quate” representation $R$.
3. Give a theorem relating representations produced by $R$ to dataflow functions given by $F$.
4. Give an alternative method of calculating representations, called $FR$, more efficient than $R$,
   which uses the definition of $F$ 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

3
\[ e \in \text{Exp} \]
\[ x \in \text{Var} \]
\[ \ell \in \text{Label} \]
\[ P \in \text{Pgm} ::= x = e \mid \text{skip} \mid \text{if} \ e \ \text{then} \ P_1 \ \text{else} \ P_2 \mid P_1 ; P_2 \]
\[ \mid \text{while} \ e \ \text{do} \ P \mid \ell : P \mid \text{break} \ \ell \]

Figure 2: The language treated in this paper

\[ \mathcal{F}[\text{skip}] = \text{id} \]
\[ \mathcal{F}[x = e] = \text{asgn}(x, e) \]
\[ \mathcal{F}[P_1 ; P_2] = \mathcal{F}[P_1] \land \mathcal{F}[P_2] \]
\[ \mathcal{F}[\text{if then} \ (e) \ P_1 \ \text{else} \ P_2] = \text{exp}(e) ; (\mathcal{F}[P_1] \land \mathcal{F}[P_2]) \]

Figure 3: First framework.

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 \to Data (conflined to functions that preserve \top_{Data}).

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 (\( \land \)) operation on functions is defined pointwise, and \text{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:

\[ \text{id} : \text{DFFun} \]
\[ \text{asgn} : \text{Var} \times \text{Exp} \to \text{DFFun} \]
\[ \text{exp} : \text{Exp} \to \text{DFFun} \]
\[ ; : \text{DFFun} \times \text{DFFun} \to \text{DFFun} \]
\[ \land : \text{DFFun} \times \text{DFFun} \to \text{DFFun} \]

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 = \( P(Var)^2 \), with ordering

\[ (D, U) \sqsubseteq (D', U') \text{ if } D \subseteq D' \text{ and } U \supseteq U' \]
\[ R[\text{skip}] = id_R \]
\[ R[x = e] = \text{asgn}_R(x, e) \]
\[ R[P_1; P_2] = R[P_1] ;_R R[P_2] \]
\[ R[\text{if then } (e) P_1 \text{ else } P_2] = \text{exp}_R(e) ;_R (R[P_1] \land_R R[P_2]) \]

Figure 4: Representation function for the first framework.

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.

\[
\begin{align*}
\text{asgn}(x, e) &= \lambda(D, U)(D \cup \{x\}, (\text{vars}(e) \setminus D) \cup U) \\
\text{exp}(e) &= \lambda(D, U)(D, (\text{vars}(e) \setminus D) \cup U)
\end{align*}
\]

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

Returning to the general case, our task is to find representations of elements of \(\text{DFFun}\) for each analysis.

**Definition** Suppose \(R\) is a set with the following values and functions (\(\top_R\) is not used until the next subsection):

\[
\begin{align*}
\top_R : R & \quad \text{exp}_R : \text{Exp} \rightarrow R \\
id_R : R & \quad ;_R : R \times R \rightarrow R \\
\text{asgn}_R : \text{Var} \times \text{Exp} \rightarrow R & \quad \land_R : R \times R \rightarrow R
\end{align*}
\]

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

\[
\begin{align*}
\text{abs}(\top_R) &= \lambda d. \top\_\text{Data} \\
\text{abs}(id_R) &= id \\
\text{abs}(\text{asgn}_R(x, e)) &= \text{asgn}(x, e) \\
\text{abs}(\text{exp}_R(e)) &= \text{exp}(e) \\
\text{abs}(r ;_R r') &= \text{abs}(r) ; \text{abs}(r') \\
\text{abs}(r \land_R r') &= \text{abs}(r) \land \text{abs}(r')
\end{align*}
\]

Define \(R : \text{Pgm} \rightarrow R\) to be the function in Figure 4.

**Theorem** If \(R\) is an adequate representation, then for all \(P\), \(\text{abs}(R[P]) = F[P]\).

**Proof** A trivial structural induction. \(\square\)

For uninitialized variables, a natural representation, which is also adequate, is almost the same as \(\text{Data}\):

\[ R = \mathcal{P}(\text{Var})^2 \cup \{\top_R\} \]
For any fragment $P$, $R[[P]]$ is the pair containing the set of variables definitely defined in $P$ and the set possibly used without definition in $P$. The operations on this representation are:

$$\begin{align*}
id_R &= (\emptyset, \emptyset) \\
\text{asgn}_R(x, e) &= (\{x\}, \text{vars}(e)) \\
\text{exp}_R(e) &= (\emptyset, \text{vars}(e)) \\
(D, U) ; R (D', U') &= (D \cup D', U \cup (U' \setminus D)) \\
(D, U) \land_R (D', U') &= (D \cap D', U \cup U')
\end{align*}$$

The $\text{abs}$ function is defined as

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

We would like to note that $\text{abs}(\top_R)$ necessarily equals $\lambda d. \top_{\text{Data}}$, as required by the definition of adequacy.

To illustrate the analysis, we show a program annotated with the value of $R[[P]]$ for each subtree $P$:

```plaintext
// (\{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\}, \emptyset)
    x = x + y + w; // (\{x\}, \{x, y, w\})
  } else
  x = 0; // (\{x\}, \emptyset)
```

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

$$F[\text{while } e \text{ do } P] = \text{mfixp}(\lambda p. \text{exp}(e); (F[P]; F[P]; F[P]; \ldots))$$

We cannot define $R[[\text{while } e \text{ do } P]]$ 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, $F[\text{while } e \text{ do } P]$ will be equal to the first element of the list $\text{exp}(e), \text{exp}(e), F[P]; \text{exp}(e), F[P]; \text{exp}(e), \ldots$ 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 $R$ to calculate the representation of fragments. In practice, we calculate them by using the definition of $F$. This method will turn out, in the following sections, to be more efficient.

Define $F^R : \text{Pgm} \to R \to R$ to be the function in Figure 5, with the relevant operations defined as follows:

$$\begin{align*}
id^R &= \text{id} \\
\text{asgn}^R(x, e) &= \lambda r. r \text{asgn}_R(x, e) \\
\text{exp}^R(e) &= \lambda r. r \text{exp}_R(x, e) \\
f \land^R g &= \lambda r. f r \land_R g r
\end{align*}$$

1Throughout the paper, to avoid clutter, we ignore $\top$ when defining functions; in every case, the definitions of $\text{asgn}(x, e), \text{exp}(e), ;_R, \land_R$, and $\text{abs}$ should check for $\top$ and return it.
\[ F^R[\text{skip}] = \text{id}^R \]
\[ F^R[x = e] = \text{asgn}^R(x, e) \]
\[ F^R[P_1 ; P_2] = F^R[P_1] ; F^R[P_2] \]
\[ F^R[\text{if then } (e) \text{ else } P_1] = \text{exp}^R(e) ; (F^R[P_1] \wedge R F^R[P_2]) \]

Figure 5: \( F^R \) for the first framework.

**Definition** \( r \equiv r' \) if \( \text{abs}(r) = \text{abs}(r') \).

**Theorem** If \( R \) is adequate, then for all \( P \) and \( r \), \( F^R[P]r \equiv r : R R[P] \).

**Proof** The proof is by induction on the structure of \( P \).

- **skip**:
  \[
  \text{abs}(F^R[\text{skip}]r) = \text{abs}(\text{id}^R(r)) = \text{abs}(r) = \text{abs}(r) ; \text{id} = \text{abs}(r) ; \text{abs}(\text{id}^R) = \text{abs}(r) ; \text{abs}(R[\text{skip}]) = \text{abs}(r : R R[\text{skip}])
  \]

- **\( x = e \)**
  \[
  \text{abs}(F^R[x = e]r) = \text{abs}(\text{asgn}^R(x, e)(r)) = \text{abs}(r ; R \text{asgn}^R(x, e)) = \text{abs}(r : R R[x = e])
  \]

- **\( P_1; P_2 \)**
  By the induction hypothesis, we have, \( \forall r \in R \),
  \[
  \text{abs}(F^R[P_1]r) = \text{abs}(r : R R[P_1])
  \]
  \[
  \text{abs}(F^R[P_2]r) = \text{abs}(r : R R[P_2])
  \]
  Now we work on \( P_1; P_2 \):
  \[
  \text{abs}(F^R[P_1; P_2]r) = \text{abs}(F^R[P_2](F^R[P_1]r)) = \text{abs}((F^R[P_1]r) ; R R[P_2]) = \text{abs}(F^R[P_1]r) ; \text{abs}(R[P_2]) = \text{abs}(r : R R[P_1]) ; \text{abs}(R[P_2]) = \text{abs}(r : R R[P_1] ; R R[P_2]) = \text{abs}(r : R R[P_1] ; R R[P_2]) = \text{abs}(r : R R[P_1] ; R R[P_2]) = \text{abs}(r : R R[P_1] ; R R[P_2])
  \]

7
• if (e) \( P_1 \) else \( P_2 \)

By the induction hypothesis, we have, \( \forall r \in R \),

\[
\text{abs}(\mathcal{F}^R[P_1]r) = \text{abs}(r ; R \mathcal{R}[P_1])
\]

\[
\text{abs}(\mathcal{F}^R[P_2]r) = \text{abs}(r ; R \mathcal{R}[P_2])
\]

Now we work on if (e) \( P_1 \) else \( P_2 \):

\[
\text{abs}(\mathcal{F}^R[\text{if (e) } P_1 \text{ else } P_2]r) = \text{abs}((\text{exp}^R(e) ; (\mathcal{F}^R[P_1] \wedge R \mathcal{F}^R[P_2]))r)
\]

\[
= \text{abs}((\mathcal{F}^R[P_1] \wedge R \mathcal{F}^R[P_2])(r ; R \text{exp}(e)))
\]

\[
= \text{abs}((\mathcal{F}^R[P_1](r ; R \text{exp}(e)) \wedge \mathcal{F}^R[P_2](r ; R \text{exp}(e)))
\]

\[
= \text{abs}((r ; R \text{exp}(e)) ; R \mathcal{R}[P_1]) \wedge \text{abs}((r ; R \text{exp}(e)) ; R \mathcal{R}[P_2])
\]

\[
= \text{abs}(r ; R \text{exp}(e)) ; R \mathcal{R}[P_1]) \wedge \text{abs}(r ; R \text{exp}(e)) ; R \mathcal{R}[P_2])
\]

\[
= \text{abs}(r ; \text{exp}(e)) ; R \mathcal{R}[P_1]) \wedge \text{abs}(r ; \text{exp}(e)) ; R \mathcal{R}[P_2])
\]

\[
= \text{abs}(r ; \text{exp}(e)) ; R \mathcal{R}[P_1]) \wedge \text{abs}(r ; \text{exp}(e)) ; R \mathcal{R}[P_2])
\]

\[
= \text{abs}(r ; \mathcal{R}[\text{if (e) } P_1 \text{ else } P_2])
\]

\[
= \text{abs}(r ; R \mathcal{R}[P_1])
\]

\[
\square
\]

**Corollary** \( \mathcal{F}^R[P] \text{id}_R \equiv \mathcal{R}[P] \).

**Proof**

\[
\text{abs}(\mathcal{F}^R[P] \text{id}_R) = \text{abs}(\text{id}_R ; R \mathcal{R}[P])
\]

\[
= \text{abs}(\text{id}_R) ; R \mathcal{R}[P]
\]

\[
= \text{id} ; R \mathcal{R}[P]
\]

\[
= R \mathcal{R}[P]
\]

\[
\square
\]

If \text{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 \( \mathcal{F}^R[T] \text{id}_R \).

**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 \( Q_1, \ldots, Q_n \), we have an AST in which some nodes have already been annotated with representations (namely, the hole-free subtrees of \( P \) and the plugs \( Q_1, \ldots, Q_n \)). We can now calculate \( \mathcal{F}[P][Q_1, \ldots, Q_n] \) 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.

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 $\eta$ is a function in $\text{Env} = \text{Label} \rightarrow \text{Data}$. Now the incoming and outgoing values are pairs:

$$F[P] : \text{Env} \times \text{Data} \rightarrow \text{Env} \times \text{Data}$$

The extended analysis is shown in Figure 6. $\text{asgn}$ and $\text{exp}$ have the same types as in the previous section; semi-colon is again function composition (in the expanded space), and $\text{id}$ 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 $F[P](\eta, d)$. $d$ contains information about the control flow paths that reach $P$. $\eta$ 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 $d \land \eta(L)$. Furthermore, the “normal exit” from $P$ is $\top_{\text{Data}}$. For any statement $Q$, $F[P]Q$ preserves $\top_{\text{Data}}$ in its second argument. so any statements directly following $P$ will be ignored. For labelled statements, $F[L : P](\eta, d)$ first calculates $F[P](\eta, d)$. It is important that $\eta(L) = \top_{\text{Data}}$; this will be the case because (a) the initial value for any analysis has the environment $\lambda \ell. \top_{\text{Data}}$, (b) $L : P$ is legal, so it cannot be within another statement labelled $L$, and (c) $F[L : P](\eta, d)$ returns an environment in which $L$ is reset to $\top_{\text{Data}}$.

Representations of these functions are derived from representations of functions in $\text{DFFun}$. Assume $R$ is an adequate representation of $\text{DFFun}$. It can be extended to a representation $E_R$ of
we have the following theorem.

Figure 7 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:

\[
\text{abs}_E : E_R \rightarrow (Env \times Data \rightarrow Env \times Data)
\]

\[
\text{abs}_E(\eta_R, r) = \lambda(\eta, d). (\lambda.d. \eta) \land \text{abs}(\eta_R) d, \text{abs}(r) d
\]

we have the following theorem.

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

**Proof** The proof is by induction on the structure of \( P \).

- **skip**:

\[
\text{abs}_E(\mathcal{R}[\text{skip}]) = \text{abs}_E((\top_{Env_R}, id_R))
\]

\[= \lambda(\eta', d'). (\lambda.d'. \eta') \land \text{abs}(\top_{Env_R}) d', \text{abs}(id_R) d' \]

\[= \lambda(\eta', d'). (\lambda.d'. \eta') \land \text{abs}(\top_R) d', \text{abs}(id) d' \]

\[= \lambda(\eta', d'). (\lambda.d'. \eta') \land \top_{Data}, d' \]

\[= \lambda(\eta', d'). (\lambda.d'). \top_{Data}, d' \]

\[= \mathcal{F}[\text{skip}]\]
\[ x = e \]

\[
\text{abs}_{E}(R[x = e]) = \text{abs}_{E}(\langle T_{Env}, \text{asgn}_{R}(x, e) \rangle) = \lambda(\eta', d').(\lambda_{\ell}.\eta'(\ell) \land \text{abs}(T_{Env}(\ell))d', \text{abs}(\text{asgn}_{R}(x, e))d')
\]

\[
= \lambda(\eta', d').(\lambda_{\ell}.\eta'(\ell) \land \text{abs}(T_{R})d', \text{asgn}(x, e)(d'))
\]

\[
= \lambda(\eta', d').(\lambda_{\ell}.\eta'(\ell) \land (\lambda_{d}.T_{Data})d', \text{asgn}(x, e)(d'))
\]

\[
= \lambda(\eta', d').(\lambda_{\ell}.\eta'(\ell) \land T_{Data}, \text{asgn}(x, e)(d'))
\]

\[
= \lambda(\eta', d').(\eta', \text{asgn}(x, e)(d'))
\]

\[
= \mathcal{F}[x = e]
\]

- \[ \text{break } \ell \]

\[
\text{abs}_{E}(R[\text{break } \ell]) = \text{abs}_{E}(\langle T_{Env}[\ell \mapsto \text{id}_{R}], T_{R} \rangle) = \lambda(\eta', d').(\lambda_{\ell'}.\eta'(\ell') \land \text{abs}(T_{Env}[\ell \mapsto \text{id}_{R}](\ell'))d', \text{abs}(T_{R})d')
\]

\[
= \lambda(\eta', d').(\lambda_{\ell'}.\eta'(\ell') \land \text{abs}(\text{id}_{R})d' \text{ if } \ell = \ell', \text{abs}(\text{id}_{R})d' \text{ if } \ell \neq \ell', \text{abs}(T_{R})d')
\]

\[
= \lambda(\eta', d').(\lambda_{\ell'} \{ \eta'(\ell') \land d' \text{ if } \ell = \ell', \eta'(\ell') \land d' \text{ if } \ell \neq \ell' \}, \text{abs}(T_{Data})) = \lambda(\eta', d').(\eta'[\ell \mapsto \eta'(\ell') \land d'], \text{abs}(T_{Data})) = \mathcal{F}[\text{break } \ell]
\]

- \[ \ell : P \]

Let \((\eta, r) = R[P]\). By the induction hypothesis we have

\[
\mathcal{F}[P] = \text{abs}_{E}(R[P]) = \text{abs}_{E}(\langle T_{R}, r \land \eta(\ell) \rangle) = \lambda(\eta', d').(\lambda_{\ell'}.\eta'(\ell') \land \text{abs}(\eta(\ell'))d', \text{abs}(r \land \eta(\ell))d')
\]

\[
= \lambda(\eta', d').(\lambda_{\ell'}.\eta'(\ell') \land \text{abs}(\eta(\ell'))d' \text{ if } \ell = \ell', \text{abs}(\eta(\ell'))d' \text{ if } \ell \neq \ell', \text{abs}(r \land \eta(\ell))d')
\]

\[
= \lambda(\eta', d').(\lambda_{\ell'} \{ \eta'(\ell') \land \text{abs}(\eta(\ell'))d' \text{ if } \ell = \ell', \eta'(\ell') \land \text{abs}(\eta(\ell'))d' \text{ if } \ell \neq \ell' \}, \text{abs}(r \land \eta(\ell))d') = \lambda(\eta', d').(\lambda_{\ell'} \{ T_{Data} \land \text{abs}(\eta(\ell'))d' \text{ if } \ell = \ell', \text{abs}(r \land \eta(\ell))d' \land \text{abs}(\eta(\ell'))d' \text{ if } \ell \neq \ell' \}, \text{abs}(r \land \eta(\ell))d')
\]

\[
= \lambda(\eta', d').(\lambda_{\ell'} \{ \eta'(\ell') \land \text{abs}(\eta(\ell'))d' \text{ if } \ell = \ell', \text{abs}(r \land \eta(\ell))d' \land \text{abs}(\eta(\ell'))d' \text{ if } \ell \neq \ell' \}) (1)
\]

11
And $\mathcal{F}[ℓ : P]$:

$$\mathcal{F}[ℓ : P] = \lambda(\eta', d'). \text{let } (η_1, d_1) \leftarrow \mathcal{F}[P][η')(ℓ', d')$$

in $(η_1[ℓ \mapsto \top_data], d_1 \land η_1(ℓ))$

$$= \lambda(\eta', d'). \text{let } (η_1, d_1) \leftarrow (\lambda(\eta', d').(\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η(ℓ'))d'), \text{abs}(r(ℓ'))d')(η', d')$$

in $(η_1[ℓ \mapsto \top_data], d_1 \land η_1(ℓ))$

$$= \lambda(\eta', d'). \text{let } (η_1, d_1) \leftarrow (\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η(ℓ'))d'), \text{abs}(r(ℓ'))d')$$

in $(η_1[ℓ \mapsto \top_data], d_1 \land η_1(ℓ))$

$$= \lambda(\eta', d').((\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η(ℓ'))d')[ℓ \mapsto \top_data], \text{abs}(r(ℓ'))d' ∧ (\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η(ℓ'))d')][ℓ \mapsto \top_data], \text{abs}(r(ℓ'))d')$$

$$= \lambda(\eta', d').((\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η(ℓ'))d')[ℓ \mapsto \top_data], \text{abs}(r(ℓ'))d' ∧ η(ℓ) ∧ \text{abs}(η(ℓ))d')$$

$$= \lambda(\eta', d').((\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η(ℓ'))d')[ℓ \mapsto \top_data], \text{abs}(r(ℓ'))d' ∧ η(ℓ) ∧ \text{abs}(η(ℓ))d')$$

$$= \lambda(\eta', d').(\lambdaℓ'. \left\{ \begin{array}{ll} \top_data & \text{if } ℓ = ℓ' \\ \eta'(ℓ') ∧ \text{abs}(η(ℓ'))d' & \text{if } ℓ \neq ℓ' \end{array} \right., \text{abs}(r(ℓ'))d' ∧ \top_data ∧ \text{abs}(η(ℓ))d')$$

$$= (1)$$

- $P_1; P_2$
  Let $(η_1, r_1) = R[P_1]$ and $(η_2, r_2) = R[P_2]$. By the induction hypothesis we have

$$\mathcal{F}[P_1] = \text{abs}_E(R[P_1])$$

$$= \lambda(\eta', d').(\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η(ℓ'))d'), \text{abs}(r_1)d')$$

$$\mathcal{F}[P_2] = \text{abs}_E(R[P_2])$$

$$= \lambda(\eta', d').(\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η(ℓ'))d'), \text{abs}(r_2)d')$$

Now we work on $\text{abs}_E(R[P_1; P_2])$.

$$\text{abs}_E(R[P_1; P_2]) = \text{abs}_E((η_1 ∧ R (r_1; R η_2), r_1; R r_2))$$

$$= \lambda(\eta', d').(\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η_1 ∧ R (r_1; R η_2))(ℓ'))d'), \text{abs}(r_1; R r_2)d')$$

$$= \lambda(\eta', d').(\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η_1(ℓ'))d') ∧ \text{abs}(η_2(ℓ'))d)(\text{abs}(r_1)d'), \text{abs}(r_2)(\text{abs}(r_1)d'))$$

$$= (2)$$

And $\mathcal{F}[P_1; P_2]$

$$\mathcal{F}[P_1; P_2] = \lambda(\eta', d').(\mathcal{F}[P_1; P_2])(η', d')$$

$$= \lambda(\eta', d').\mathcal{F}[P_1; P_2](η')(ℓ', d')$$

$$= \lambda(\eta', d').\mathcal{F}[P_2](\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η_1(ℓ'))d'), \text{abs}(r_1)d')(η', d'))$$

$$= \lambda(\eta', d').\mathcal{F}[P_2](\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η_1(ℓ'))d'), \text{abs}(r_1)d')$$

$$= \lambda(\eta', d').(\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η_2(ℓ'))d'), \text{abs}(r_2)d')(\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η_1(ℓ'))d'), \text{abs}(r_1)d')$$

$$= \lambda(\eta', d').(\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η_2(ℓ'))d'), \text{abs}(r_2)d')(\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η_1(ℓ'))d'), \text{abs}(r_1)d')$$

$$= \lambda(\eta', d').(\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η_1(ℓ'))d') ∧ \text{abs}(η_2(ℓ'))(\text{abs}(r_1)d'), \text{abs}(r_2)(\text{abs}(r_1)d'))$$

$$= \lambda(\eta', d').(\lambdaℓ'.η'((ℓ') ∧ \text{abs}(η_1(ℓ'))d') ∧ \text{abs}(η_2(ℓ'))(\text{abs}(r_1)d'), \text{abs}(r_2)(\text{abs}(r_1)d'))$$

$$= (2)$$
• if (e) \( P_1 \) else \( P_2 \)
  Let \((\eta_1, r_1) = R[P_1] \) and \((\eta_2, r_2) = R[P_2] \). By the induction hypothesis we have

\[
\mathcal{F}[P_1] = \text{abs}_E(R[R[P_1]]) \\
= \text{abs}_E((\eta_1, r_1)) \\
= \lambda(\eta', d').(\lambda' \cdot \eta'(\ell') \land \text{abs}(\eta_1(\ell'))d', \text{abs}(r_1)d')
\]

\[
\mathcal{F}[P_2] = \text{abs}_E(R[R[P_2]]) \\
= \text{abs}_E((\eta_2, r_2)) \\
= \lambda(\eta', d').(\lambda' \cdot \eta'(\ell') \land \text{abs}(\eta_2(\ell'))d', \text{abs}(r_2)d')
\]

Now we work on \( \text{abs}_E(R[R[\text{if}(e) P_1 \text{ else } P_2]]) \).

\[
\text{abs}_E(R[R[\text{if}(e) P_1 \text{ else } P_2]]) = \text{abs}_E(\text{exp}_R(e); R((\eta_1, r_1) \land_R (\eta_2, r_2))) \\
= \text{abs}_E(\text{exp}_R(e); R((\eta_1, r_1) \land_R (\eta_2, r_2))) \\
= \lambda(\eta', d').(\lambda' \cdot \eta'(\ell') \land \text{abs}(\text{exp}_R(e)); \text{abs}(\eta_1 \land_R \eta_2)(\ell'))d', \\
(\text{abs}(\text{exp}_R(e)); \text{abs}(r_1 \land_R r_2))d') \\
= \lambda(\eta', d').(\lambda' \cdot \eta'(\ell') \land (\text{exp}(e) \land \text{abs}(\eta_1 \land_R \eta_2)(\ell'))d', \\
(\text{exp}(e); \text{abs}(r_1 \land_R r_2))d') \\
= \lambda(\eta', d').(\lambda' \cdot \eta'(\ell') \land \text{abs}(\eta_1(\ell'))(\text{exp}(e)d') \land \text{abs}(\eta_2(\ell'))(\text{exp}(e)d'), \\
\text{abs}(r_1)(\text{exp}(e)d') \land \text{abs}(r_2)(\text{exp}(e)d')) \tag{3}
\]

And \( \mathcal{F}[\text{if}(e) P_1 \text{ else } P_2] \):

\[
\mathcal{F}[\text{if}(e) P_1 \text{ else } P_2] = \lambda(\eta', d'). \text{let } (\eta'_1, d'_1) \leftarrow \mathcal{F}[P_1](\eta', \text{exp}(e)d') \\
(\eta'_2, d'_2) \leftarrow \mathcal{F}[P_2](\eta', \text{exp}(e)d') \\
in (\eta'_1, d'_1) \land (\eta'_2, d'_2) \\
= \lambda(\eta', d'). \text{let } (\eta'_1, d'_1) \leftarrow (\lambda' \cdot \eta'(\ell') \land \text{abs}(\eta_1(\ell'))d', \text{abs}(r_1)d')(\eta', \text{exp}(e)d') \\
(\eta'_2, d'_2) \leftarrow (\lambda' \cdot \eta'(\ell') \land \text{abs}(\eta_2(\ell'))d', \text{abs}(r_2)d')(\eta', \text{exp}(e)d') \\
in (\eta'_1, d'_1) \land (\eta'_2, d'_2) \\
= \lambda(\eta', d'). \text{let } (\eta'_1, d'_1) \leftarrow (\lambda' \cdot \eta'(\ell') \land \text{abs}(\eta_1(\ell'))(\text{exp}(e)d') \land \eta'(\ell') \land \text{abs}(\eta_2(\ell'))(\text{exp}(e)d'), \\
\text{abs}(r_1)(\text{exp}(e)d') \land \text{abs}(r_2)(\text{exp}(e)d')) \\
(\eta'_2, d'_2) \leftarrow (\lambda' \cdot \eta'(\ell') \land \text{abs}(\eta_2(\ell'))(\text{exp}(e)d'), \\
\text{abs}(r_2)(\text{exp}(e)d')) \\
in (\eta'_1, d'_1) \land (\eta'_2, d'_2) \\
= \lambda(\eta', d').(\lambda' \cdot \eta'(\ell') \land \text{abs}(\eta_1(\ell'))(\text{exp}(e)d') \land \eta'(\ell') \land \text{abs}(\eta_2(\ell'))(\text{exp}(e)d'), \\
\text{abs}(r_1)(\text{exp}(e)d') \land \text{abs}(r_2)(\text{exp}(e)d')) \\
= (3)
\]
\[ \mathcal{F}^R[\text{skip}] = \text{id}^R \]
\[ \mathcal{F}^R[x = e] = \lambda(\eta, r).\langle \eta, \text{asgn}^R(x, e) r \rangle \]
\[ \mathcal{F}^R[\text{break } \ell] = \lambda(\eta, r).\langle \eta[\ell \leftarrow r \&_R \eta(\ell)], \top_R \rangle \]
\[ \mathcal{F}^R[\ell : P] = \lambda(\eta, r). \text{ let } (\eta_1, r_1) \leftarrow \mathcal{F}^R[P](\eta, r) \]
\[ \text{ in } (\eta_1[\ell \leftarrow \top_R], r_1 \&_R \eta(\ell)) \]
\[ \mathcal{F}^R[P_1; P_2] = \mathcal{F}^R[P_1]; \mathcal{F}^R[P_2] \]
\[ \mathcal{F}^R[\text{if } (e) P_1 \text{ else } P_2] = \lambda(\eta, r). \text{ let } (\eta_1, r_1) \leftarrow \mathcal{F}^R[P_1](\eta, \text{exp}^R(e) r) \]
\[ \text{ in } (\eta_1, r_1) \&_R (\eta_2, r_2) \]

Figure 8: \( \mathcal{F}^R \) with break statements.

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 as given in Figure 8 where \( \text{asgn}^R \) and \( \text{exp}^R \) are exactly the same as in the previous section; \( \text{id}^R \) has the same definition but different type.

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

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

**Proof** The proof is by induction on the structure of \( P \).

- **skip**
  - For this case, \( (\eta, r) = (\top_{\text{Env}_R}, \text{id}_R) \).

\[
\text{abs}_E(\mathcal{F}^R[\text{skip}](\eta', r')) = \text{abs}_E(\text{id}^R(\eta', r'))
\]
\[
= \text{abs}_E(\langle \eta', r' \rangle)
\]
\[
= \lambda(\eta'', d').\langle \lambda \ell. \eta''(\ell) \&_R (r'; \eta(\ell)) d', \text{abs}(r')d'' \rangle
\]

And

\[
\text{abs}_E((\lambda \ell. \eta''(\ell) \&_R (r'; \top_{\text{Env}_R}(\ell)) r'; \text{id}_R))
\]
\[
= \text{abs}_E((\lambda \ell. \eta''(\ell) \&_R (r'; \top_{\text{Env}_R}(\ell)))(\ell))d'', \text{abs}(r'; \text{id}_R)d''
\]
\[
= \lambda(\eta'', d'')(\lambda \ell. \eta''(\ell) \&_R (r'; \top_{\text{Env}_R}(\ell)))(\ell)d'', \text{abs}(r'; \text{id}_R)d''
\]
\[
= \lambda(\eta'', d'').\langle \lambda \ell. \eta''(\ell) \&_R \text{abs}(\eta''(\ell))d'' \&_R \top_{\text{Data}}, \text{abs}(r')d'' \rangle
\]
\[
= \lambda(\eta'', d'').\langle \lambda \ell. \eta''(\ell) \&_R \text{abs}(\eta''(\ell))d'' \rangle d'', \text{abs}(r')d''
\]
\[
= (1)
\]
• $x = e$

  For this case, $(\eta, r) = (\top_{Env_R}, \text{asgn}_R(x, e))$.

  $\text{abs}_E(\mathcal{F}^R[x = e][\eta'], r')) = \text{abs}_E((\eta', \text{asgn}_R(x, e)(r')))$

  $= \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \land \text{abs}(\eta'(\ell))d', \text{abs}((\lambda\ell.\eta'(\ell))d'))$

  $= \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \land \text{abs}(\eta'(\ell))d', \text{abs}(r' : \text{asgn}_R(x, e))d')$

  (2)

• break $\ell$

  For this case, $(\eta, r) = (\top_{Env_R}[\ell \mapsto \text{id}_R], \top_R)$.

  $\text{abs}_E(\mathcal{F}^R[\text{break $\ell$}][\eta'], r')) = \text{abs}_E((\eta'[\ell \mapsto r' \land \top_R(\ell)], \top_R))$

  $= \lambda(\eta'', d').(\lambda\ell.\eta'(\ell) \land \text{abs}(\eta'(\ell))d', \text{abs}(\top_R)d')$

  $= \lambda(\eta'', d').(\lambda\ell.\eta'(\ell) \land \text{abs}(\eta'(\ell))d', \text{abs}(r' : \text{asgn}_R(x, e))d')$

  (3)

And

  $\text{abs}_E((\lambda\ell.\eta'(\ell) \land R (r' : \top_{Env_R}[\ell \mapsto \text{id}_R](\ell)), r' : \text{asgn}_R(x, e)))$

  $= \lambda(\eta'', d').(\lambda\ell.\eta''(\ell) \land \text{abs}(\lambda\ell.\eta''(\ell) \land R (r' : \top_{Env_R}[\ell \mapsto \text{id}_R](\ell)))d', \text{abs}(\text{asgn}_R(x, e))d')$

  (4)

• $\ell : P$

  Let $(\eta, r) = \mathcal{R}[P]$. By the induction hypothesis, we have

  $\text{abs}_E(\mathcal{F}^R[P][\eta'], r'))$

  $= \text{abs}_E((\lambda\ell.\eta'(\ell) \land R (r' : \lambda\ell.\eta'(\ell)), r' : R r))$

  $= \lambda(\eta'', d').(\lambda\ell.\eta''(\ell) \land \text{abs}(\lambda\ell.\eta''(\ell) \land R (r' : \lambda\ell.\eta'(\ell)))d', \text{abs}(r' : R r)d')$

  (4)

Let $(\eta_1, r_1) = \mathcal{F}^R[P][\eta', r']$. Then we get

  $\text{abs}_E(\mathcal{F}^R[P][\eta', r']) = \text{abs}_E((\eta_1, r_1))$

  $= \lambda(\eta'', d').(\lambda\ell.\eta''(\ell) \land \text{abs}(\eta_1(\ell))d', \text{abs}(r_1)d')$

  (5)
Since \((4) = (5)\), we obtain
\[
\text{abs}(r' : R r) d'' = \text{abs}(r_1) d''
\]
and
\[
\eta''(\ell') \land \text{abs}(\eta'(\ell') \land R (r' : R \eta(\ell'))) d'' = \eta''(\ell') \land \text{abs} (\eta_1(\ell')) d''
\]
When \(\ell' = \ell\), using the legality condition for all the programs, we get
\[
\eta''(\ell') \land \text{abs}(\eta'(\ell') \land R (r' : R \eta(\ell'))) d'' = \eta''(\ell') \land \text{abs} (\eta_1(\ell')) d''
\Rightarrow \top_{\text{Data}} \land \text{abs}(\top_{\text{R}} \land R (r' : R \eta(\ell'))) d'' = \top_{\text{Data}} \land \text{abs} (\eta_1(\ell')) d''
\Rightarrow \text{abs}(r' : R \eta(\ell')) d'' = \text{abs} (\eta_1(\ell')) d''
\]
Now we work on \(\ell : P\):
\[
\begin{align*}
\text{abs}_{E}(\mathcal{F}^R[\ell : P](\eta', r)) &= \text{abs}_{E}(\eta_1[\ell \mapsto \top_{\text{R}}], r_1 \land R \eta_1(\ell))) \\
&= \lambda(\eta'', d''). (\lambda \ell. \eta''(\ell') \land \text{abs} (\eta_1[\ell \mapsto \top_{\text{R}}](\ell')) d'', \text{abs}(r_1 \land R \eta_1(\ell)) d'') \\
&= \lambda(\eta'', d''). (\lambda \ell. \{ \eta''(\ell') \land \text{abs}(\top_{\text{R}}) d'' \text{ if } \ell = \ell' \} \land \text{abs}(\eta_1(\ell')) d'' \text{ if } \ell \neq \ell' , \\
&\quad \text{abs}(r_1) d'' \land \text{abs} (\eta_1(\ell')) d'') \\
&= \lambda(\eta'', d''). (\lambda \ell. \{ \eta''(\ell') \land \text{abs}(\eta_1(\ell')) d'' \text{ if } \ell = \ell' \} \land \text{abs}(r_1 \land R \eta(\ell)) d'') \\
&= \lambda(\eta'', d''). (\lambda \ell. \{ \eta''(\ell') \land \text{abs}(\eta_1(\ell')) d'' \text{ if } \ell = \ell' \} \land \text{abs}(r' : R \eta(\ell)) d'')
\end{align*}
\]
And using the fact that \(R[\ell : P] = (\eta[\ell \mapsto \top_{\text{R}}], r \land R \eta(\ell))\), we have
\[
\begin{align*}
\text{abs}_{E}(\mathcal{F}^R[\ell : P](\eta' : R \eta)[\ell \mapsto \top_{\text{R}}](\ell'), r' : R (\ell' \land R \eta(\ell))) &= \lambda(\eta'', d''). (\lambda \ell. \eta''(\ell') \land \text{abs}((\lambda \ell'. \eta''(\ell') \land R (r' : R \eta[\ell \mapsto \top_{\text{R}}](\ell')) d''), \\
&\quad \text{abs}(r' : R (\ell' \land R \eta(\ell))) d'') \\
&= \lambda(\eta'', d''). (\lambda \ell. \{ \eta''(\ell') \land \text{abs}(\top_{\text{R}}) d'' \text{ if } \ell = \ell' \} \land \text{abs}(\eta(\ell') \land R (r' : R \eta(\ell')) d'') \text{ if } \ell \neq \ell' , \\
&\quad \text{abs}(r' : R (\ell' \land R \eta(\ell))) d'') \\
&= \lambda(\eta'', d''). (\lambda \ell. \{ \eta''(\ell') \land \text{abs}(\top_{\text{R}}) d'' \text{ if } \ell = \ell' \} \land \text{abs}(\eta(\ell') \land R (r' : R \eta(\ell')) d'') \text{ if } \ell \neq \ell' , \\
&\quad \text{abs}(r' : R (\ell' \land R \eta(\ell))) d'') \\
&= \lambda(\eta'', d''). (\lambda \ell. \{ \eta''(\ell') \land \text{abs}(\eta(\ell') \land R (r' : R \eta(\ell')) d'') \text{ if } \ell = \ell' \} \land \text{abs}(r' : R (\ell' \land R \eta(\ell))) d'') \\
&= \lambda(\eta'', d''). (\lambda \ell. \{ \eta''(\ell') \land \text{abs}(\eta(\ell') \land R (r' : R \eta(\ell')) d'') \text{ if } \ell = \ell' \} \land \text{abs}(r' : R (\ell' \land R \eta(\ell))) d'') \\
&= (6)
\end{align*}
\]
\( P_1; P_2 \)

Let \((\eta_1, r_1) = \mathcal{R}[P_1], (\eta_2, r_2) = \mathcal{R}[P_2], (\eta_3, r_3) = \mathcal{F}^R[P_1](\eta', r'), \) and \((\eta_4, r_4) = \mathcal{F}^R[P_2](\eta_3, r_3). \) By the induction hypothesis, we have

\[
\text{abs}_E(\mathcal{F}^R[P_1](\eta', r')) = \text{abs}_E((\eta_3, r_3)) = \lambda(\eta'', d'').(\lambda\ell.\eta''(\ell') \land \text{abs}(\eta_3(\ell'))d''), \text{ abs}(r_3)d''
\]

and

\[
\text{abs}_E(\mathcal{F}^R[P_1](\eta', r')) = \text{abs}_E((\lambda'\ell.\eta''(\ell') \land_R (r'; R \eta_1(\ell'))), r'; R r_1))
\]

\[
= \lambda(\eta'', d'').(\lambda\ell.\eta''(\ell') \land \text{abs}((\lambda\ell.\eta''(\ell') \land_R (r'; R \eta_1(\ell')))(\ell'))d''), \text{ abs}(r'; R r_1)d''
\]

\[
= \lambda(\eta'', d'').(\lambda\ell.\eta''(\ell') \land \text{abs}(\eta_3(\ell') \land_R (r'; R \eta_1(\ell'))))d''), \text{ abs}(r'; R r_1)d''
\]

Similarly, for \( P_2 \)

\[
\text{abs}_E(\mathcal{F}^R[P_2](\eta_3, r_3)) = \text{abs}_E((\eta_3, r_3)) = \lambda(\eta'', d'').(\lambda\ell.\eta''(\ell') \land \text{abs}(\eta_3(\ell'))d''), \text{ abs}(r_3)d''
\]

and

\[
\text{abs}_E(\mathcal{F}^R[P_2](\eta_3, r_3)) = \text{abs}_E((\lambda'\ell.\eta''(\ell') \land_R (r_a ; R \eta_2(\ell'))), r_a ; R r_2))
\]

\[
= \lambda(\eta'', d'').(\lambda\ell.\eta''(\ell') \land \text{abs}((\lambda\ell.\eta''(\ell') \land_R (r_a ; R \eta_2(\ell')))(\ell'))d''), \text{ abs}(r_a ; R r_2)d''
\]

\[
= \lambda(\eta'', d'').(\lambda\ell.\eta''(\ell') \land \text{abs}(\eta_3(\ell') \land_R (r_a ; R \eta_2(\ell'))))d''), \text{ abs}(r_a ; R r_2)d''
\]

These give us the equalities

\[
\eta''(\ell') \land \text{abs}(\eta_3(\ell'))d'' = \eta''(\ell') \land \text{abs}(\eta_3(\ell') \land_R (r'; R \eta_1(\ell'))))d''
\]

\[
\text{abs}(r_3)d'' = \text{abs}(r'; R r_1)d''
\]

\[
\eta''(\ell') \land \text{abs}(\eta_3(\ell'))d'' = \eta''(\ell') \land \text{abs}(\eta_3(\ell') \land_R (r_a ; R \eta_2(\ell'))))d''
\]

\[
\text{abs}(r_3)d'' = \text{abs}(r_a ; R r_2)d''
\]

Now, returning to \( P_1; P_2, \) we have

\[
\text{abs}_E(\mathcal{F}^R[P_1; P_2](\eta', r')) = \text{abs}_E(\mathcal{F}^R[P_2](\mathcal{F}^R[P_1](\eta', r')))
\]

\[
= \text{abs}_E(\mathcal{F}^R[P_2](\eta_3, r_3)) = \lambda(\eta'', d'').(\lambda\ell.\eta''(\ell') \land \text{abs}(\eta_3(\ell'))d''), \text{ abs}(r_3)d''
\]

\[
= \lambda(\eta'', d'').(\lambda\ell.\eta''(\ell') \land \text{abs}(\eta_3(\ell') \land_R (r_a ; R \eta_2(\ell'))))d''), \text{ abs}(r_a ; R r_2)d''
\]

\[
= \lambda(\eta'', d'').(\lambda\ell.\eta''(\ell') \land \text{abs}(\eta_3(\ell') \land_R (r_a ; R \eta_2(\ell')))(\ell'))d'' \land \text{abs}(\eta_2(\ell'))(\text{abs}(r'; R r_1)d''), \text{ abs}(r_a ; R r_2)d''
\]

\[
\text{abs}(r_3)d'' = \text{abs}(r_a ; R r_2)d''
\]

for the left-hand-side of the equivalence. And using the fact that \( \mathcal{R}[P_1; P_2] = (\eta_1 \land_R (r_1 ; R \eta_2), r_1 ; R r_2), \) for the right-hand-side of the equivalence we have

\[
17
\]
By the induction hypothesis, we have
\[ abs((\lambda \ell \cdot \eta(\ell')) \land_R (r' : R (\eta_1 \land_R (r_1 : R \eta_2))(\ell'))), r' : R (r_1 : R r_2)) = \lambda(\eta', d') \cdot (\lambda \ell \cdot \eta''(\ell') \land abs((\lambda \ell \cdot \eta(\ell') \land_R (r' : R (\eta_1 \land_R (r_1 : R \eta_2))(\ell')))(\ell')) d' \]
\[ abs(r' : R (r_1 : R r_2)) d'' \]
\[ = \lambda(\eta', d') \cdot (\lambda \ell \cdot \eta''(\ell') \land abs((\lambda \ell \cdot \eta(\ell') \land_R (r' : R (\eta_1 \land_R (r_1 : R \eta_2))(\ell')))(\ell')) d' \]
\[ abs(r' : R (r_1 : R r_2)) d'' \]
\[ = \lambda(\eta', d') \cdot (\lambda \ell \cdot \eta''(\ell') \land abs((\lambda \ell \cdot \eta(\ell') \land_R (r' : R (\eta_1 \land_R (r_1 : R \eta_2))(\ell')))(\ell')) d' \]
\[ abs(r' : R (r_1 : R r_2)) d'' \]
\[ = \lambda(\eta', d') \cdot (\lambda \ell \cdot \eta''(\ell') \land abs((\lambda \ell \cdot \eta(\ell') \land_R (r' : R (\eta_1 \land_R (r_1 : R \eta_2))(\ell')))(\ell')) d' \]
\[ abs(r' : R (r_1 : R r_2)) d'' \]
\[ (7) \]

\[ if (e) P_1 \text{ else } P_2 \]

Let \((\eta_1, r_1) = R[P_1], (\eta_2, r_2) = R[P_2], (\eta_b, r_a) = R[R[P_1](\eta', \exp_R(e)r')], \) and \((\eta_b, r_b) = R[R[P_2](\eta', \exp_R(e)r')].\)

By the induction hypothesis, we have

\[ abs_E(\mathcal{F}^R[P_1](\eta', \exp_R(e)r')) = abs_E(\mathcal{F}^R[P_2](\eta', \exp_R(e)r')) \]

and

\[ abs_E(\mathcal{F}^R[P_1](\eta', \exp_R(e)r')) = abs_E(\mathcal{F}^R[P_2](\eta', \exp_R(e)r')) \]

Similarly, for \(P_2\)

\[ abs_E(\mathcal{F}^R[P_2](\eta', \exp_R(e)r')) = \lambda(\eta', d') \cdot (\lambda \ell \cdot \eta''(\ell') \land abs((\lambda \ell \cdot \eta(\ell') \land_R (r' : R (\eta_1 \land_R (r_1 : R \eta_2))(\ell')))(\ell')) d' \]

and

\[ abs_E(\mathcal{F}^R[P_2](\eta', \exp_R(e)r')) = \lambda(\eta', d') \cdot (\lambda \ell \cdot \eta''(\ell') \land abs((\lambda \ell \cdot \eta(\ell') \land_R (r' : R (\eta_1 \land_R (r_1 : R \eta_2))(\ell')))(\ell')) d' \]

These give us the equalities

\[ \eta''(\ell') \land abs(\eta_b(\ell')) d'' = \eta''(\ell') \land abs(\eta_b(\ell')) \]
\[ abs(r_a) d'' = abs(r' : R \exp_R(e); R r_1) d'' \]
\[ \eta''(\ell') \land abs(\eta_b(\ell')) d'' = \eta''(\ell') \land abs(\eta_b(\ell')) \]
\[ abs(r_b) d'' = abs(r' : R \exp_R(e); R r_2) d'' \]

18
Now, returning to if(e) $P_1$ else $P_2$, we have

$$\text{abs}_E(\mathcal{F}^R[\text{if}(e) \; P_1 \; \text{else} \; P_2](\eta', r')) = \text{abs}_E((\eta \land_R \eta_0 \land_R r_0 \land_R r_0))$$

$$= \lambda(\eta', d'), (\lambda' . \eta''(\ell') \land \text{abs}(\eta_0(\ell') \land_R \eta_0(\ell'))d', \text{abs}(r_0 \land_R r_0 \land_R r_0)d'')$$

$$= \lambda(\eta', d'), (\lambda' . \eta''(\ell') \land \text{abs}(\eta_0(\ell') \land_R \lambda_0 \exp_R(e); R \eta_1(\ell'))d' \land \text{abs}(\eta'(\ell') \land_R r_0 \exp_R(e); R \eta_2(\ell'))d'')$$

$$\quad \text{abs}(r_0 \exp_R(e); R r_1)d'' \land \text{abs}(r_0 \exp_R(e); R r_2)d'')$$

$$= \lambda(\eta', d'), (\lambda' . \eta''(\ell') \land \text{abs}(\eta'(\ell'))d'' \land \text{abs}(r_0 \exp_R(e); R \eta_1(\ell'))d'' \land \text{abs}(r_0 \exp_R(e); R \eta_2(\ell'))d'')$$

$$\quad \text{abs}(r_0 \exp_R(e); R r_1)d'' \land \text{abs}(r_0 \exp_R(e); R r_2)d'')$$

$$= \lambda(\eta', d'), (\lambda' . \eta''(\ell') \land \text{abs}(\eta'(\ell'))d'' \land \text{abs}(r_0 \exp_R(e); R \eta_1(\ell'))d'' \land \text{abs}(r_0 \exp_R(e); R \eta_2(\ell'))d'')$$

$$\quad \text{abs}(r_0 \exp_R(e); R r_1)d'' \land \text{abs}(r_0 \exp_R(e); R r_2)d'')$$  (8)

And using the fact that $\mathcal{R}[\text{if}(e) \; P_1 \; \text{else} \; P_2] = (\exp_R(e); R (\eta \land_R \eta_2), \exp_R(e); R (r_1 \land_R r_2))$,

$$\text{abs}_E((\lambda' . \eta''(\ell') \land_R (r_0 \exp_R(e); R (\eta \land_R \eta_2)))) = \lambda(\eta', d'), (\lambda' . \eta''(\ell') \land \text{abs}(\eta'(\ell'))d'' \land \text{abs}(r_0 \exp_R(e); R \eta_1(\ell'))d'' \land \text{abs}(r_0 \exp_R(e); R \eta_2(\ell'))d'')$$

$$\quad \text{abs}(r_0 \exp_R(e); R r_1)d'' \land \text{abs}(r_0 \exp_R(e); R r_2)d'')$$  (8)

**Corollary** $\mathcal{F}^R[P](\top_{\text{Env}}, \text{id}_R) = \mathcal{R}[P]$.

**Proof** Let $\mathcal{R}[P] = (\eta, r)$. Then, by the theorem above,

$$\mathcal{F}^R[P](\top_{\text{Env}}, \text{id}_R) \equiv (\lambda' . \top_{\text{Env}}(\ell') \land_R (\text{id}_R \land_R \eta(\ell')) \land_R (\text{id}_R \land_R r))$$

$$= (\lambda' . \top_{\text{Env}}(\ell') \land_R (\text{id}_R \land_R \eta(\ell')) \land_R (\text{id}_R \land_R r))$$

which means

$$\text{abs}_E(\mathcal{F}^R[P](\top_{\text{Env}}, \text{id}_R)) = \text{abs}_E((\lambda' . \top_{\text{Env}}(\ell') \land_R (\text{id}_R \land_R \eta(\ell')) \land_R (\text{id}_R \land_R r)))$$

$$= \lambda(\eta', d'), (\lambda' . \eta''(\ell') \land \text{abs}(\top_{\text{Env}}(\ell') \land_R (\text{id}_R \land_R \eta(\ell')) \land_R (\text{id}_R \land_R r))d'') \land \text{abs}(\text{id}_R \land_R (\text{id}_R \land_R r))d''$$

$$= \lambda(\eta', d'), (\lambda' . \eta''(\ell') \land \text{abs}(\top_{\text{Env}}(\ell') \land_R (\text{id}_R \land_R \eta(\ell')) \land_R (\text{id}_R \land_R r))d'') \land \text{abs}(\text{id}_R \land_R (\text{id}_R \land_R r))d''$$

$$= \lambda(\eta', d'), (\lambda' . \eta''(\ell') \land \text{abs}(\eta(\ell'))d'' \land \text{abs}(r)d'')$$

$$= \text{abs}_E((\eta, r))$$

$$= \text{abs}_E(\mathcal{R}[P])$$
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 $\mathcal{F}^R[P]((\top \triangleright Env_R, (\emptyset, \emptyset))$ for each node $P$.

```plaintext
// ((L \mapsto \{(x, y), \{x, z\}\}), ((x, w, y), \{x, z\}))
y = x; // (\emptyset, (\{y\}, \{x\}))
if (z > 10) // ((L \mapsto ((\{x\}, \{z\})), ((x, w), \{x, y, z\}))
  \{ w = 15; // (\emptyset, (\{w\}, \emptyset))
    x = x + y + w; // (\emptyset, (\{x, y, w\}))
  \}
else
  \{ // ((L \mapsto (\{x\}, \emptyset)), \top)
    x = 0; // (\emptyset, (\{x\}, \emptyset))
    break L; // ((L \mapsto (\emptyset, \emptyset)), \top)
  }
```

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 $NodeMap = Node \triangleright Data$ (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 $asgn$:

$$asgn : Node \times Var \times Exp \rightarrow DFFun$$

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 9.

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

$$F_R = (Node \triangleright R) \times Env_R \times R$$
The intermediate framework in Section 3.2, page 10.

\( \mathcal{F}[n] \) is defined as follows:

\[ \mathcal{F}[n : \text{skip}] = \lambda (\varphi, \eta, d). (\varphi[n \mapsto d], \eta, d) \]

\[ \mathcal{F}[n : x = e;] = \lambda (\varphi, \eta, d). \text{let } d' \leftarrow \text{assign}(n, x, e)(d) \]

\[ \mathcal{F}[n : \text{break } \ell;] = \lambda (\varphi, \eta, d). (\varphi[n \mapsto d'], \eta, d') \]

\[ \mathcal{F}[n : (\ell : (m_1 : P))] = \lambda (\varphi, \eta, d). \text{let } (\varphi_1, \eta_1, d_1) \leftarrow \mathcal{F}[m_1 : P](\varphi, \eta, d) \]

\[ \mathcal{F}[n] \text{ is similar to, but notationally more complex than, the corresponding proof for } \mathcal{F}[P]. \]

\[ \text{Proof: } \]

\[ \text{The proof is similar to, but notationally more complex than, the corresponding proof for the intermediate framework in Section 3.2, page 10.} \]

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

\[ \text{Theorem: } \mathcal{F}^R [P](\varphi', \eta', r') \equiv (\varphi' \cup (\lambda n. \text{abs}(\varphi_R(n))[\text{abs}(r)]) \land (\lambda\ell. \text{abs}(\eta_R)(\ell))) \land R \]

\[ \text{Proof: } \]

Our previous example with numbered nodes is in Figure 11. We show the value of function \( \mathcal{R}(P) \) only at the top node. The environment and data values are just as in Section 3.2: \( \{ L \mapsto (\{ y, \}, \{ z \}) \} \) and \( \{ (x, w, y), (x, z) \} \), respectively. The node map is:

\[ \{ n_1 \mapsto (\{ x, w, y \}, \{ x, z \}), \quad n_2 \mapsto (\{ y \}, \{ x \}), \quad n_3 \mapsto (\{ x, w, y \}, \{ x, z \}) \]

\[ n_4 \mapsto (\{ x, w, y \}, \{ x, z \}), \quad n_5 \mapsto \top_R, \quad n_6 \mapsto (\{ w, y \}, \{ x, z \}) \]

\[ n_7 \mapsto (\{ x, w, y \}, \{ x, z \}), \quad n_8 \mapsto (\{ x, y \}, \{ x, z \}), \quad n_9 \mapsto \top_R \]

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
\( \mathcal{R}[n : \text{skip}] = (\{ n \mapsto id_R \}, \top_{\text{Env}_R}, id_R) \)

\( \mathcal{R}[n : x = e] = (\{ n \mapsto \text{asgn}_R(n, x, e) \}, \top_{\text{Env}_R}, \text{asgn}_R(n, x, e)) \)

\( \mathcal{R}[n : \text{break } \ell] = (\{ n \mapsto \top_R \}, \top_{\text{Env}_R} \| \ell \mapsto id_R, \top_R) \)

\( \mathcal{R}[n : (\ell : n_1 : P)] = \text{let } (\phi, \eta, r) \leftarrow \mathcal{R}[P] \text{ in } (\text{exp}_R(e); R (\phi_1 \cup \phi_2) \| n \mapsto id_R) R (\eta_1 \wedge_R \eta_2), \text{exp}_R(e); R (r_1 \wedge_R r_2)) \)

\( \mathcal{R}[n : \text{if } (e) n_1 : P_1 \text{ else } n_2 : P_2] = \text{let } (\phi_1, \eta_1, r_1) \leftarrow \mathcal{R}[P_1], (\phi_2, \eta_2, r_2) \leftarrow \mathcal{R}[P_2] \text{ in } \text{exp}_R(e); R (\phi_1 \wedge_R \phi_2) \| n \mapsto \top_R) R (\eta_1 \wedge_R \eta_2), \text{exp}_R(e); R (r_1 \wedge_R r_2) \)

\[ \text{Data} = \mathcal{P}(\text{Node}) \cup \{ \top \} \]
Sets in \textit{Data} are ordered by reverse inclusion, with \(\emptyset\) being the element just below \(\top\). The operations are

\[
\begin{align*}
\text{asgn}(n, x, e) &= \lambda D. (D \setminus D_x) \cup \{n\} \\
\text{exp}(e) &= \lambda D. D
\end{align*}
\]

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

\[
\begin{align*}
\text{id}_R &= (\emptyset, \emptyset) \\
\text{asgn}_R(n, x, e) &= (\{x\}, \{n\}) \\
\text{exp}_R(e) &= (\emptyset, \emptyset) \\
(K_1, G_1); R \quad (K_2, G_2) &= (K_1 \cup K_2, G_2 \cup (G_1 \setminus K_2)) \\
(K_1, G_1) \land R \quad (K_2, G_2) &= (K_1 \cap K_2, G_1 \cup G_2)
\end{align*}
\]

\[
\text{abs}(K, G) = \lambda D. G \cup (D \setminus K)
\]

where \(G \setminus K = \{n \in G \mid n\) is the definition of some \(x \in K\).

\textbf{Theorem} \(R\) for reaching definitions \(I\) is an exact representation.

\textbf{Proof} In order to show that a representation is exact (i.e. there is an isomorphism between \(R\) and \textit{DFFun} defined by \textit{abs}), we need to prove two claims:

1. \(R\) is adequate (i.e. \textit{abs} defines a homomorphism)
2. No two different representations represent the same \textit{DFFun} function. (i.e. \(\forall r_1, r_2 \in R, r_1 \neq r_2 \Rightarrow \text{abs}(r_1) \neq \text{abs}(r_2)\))
Claim 1: $R$ for reaching definitions I is adequate.

Proof:

- $\text{abs}(\top_R) = \lambda D. \top_{\text{Data}}$ holds by definition.
- $\text{abs}(\text{id}_R) = \text{abs}((\emptyset, \emptyset)) = \lambda D. \emptyset \cup (D \setminus \emptyset) = \lambda D. D = \text{id}$
- $\text{abs}(\text{asgn}_R(n, x, e)) = \text{abs}([(x, [n])] = \lambda D. \{n\} \cup (D \setminus \{x\}) = \lambda D. \{n\} \cup (D \setminus D_x) = \text{asgn}(n, x, e)$
- $\text{abs}(\text{exp}_R(e)) = \text{abs}((\emptyset, \emptyset)) = \lambda D. \emptyset \cup (D \setminus \emptyset) = \lambda D. D = \text{exp}(e)$
- $\text{abs}((K_1, G_1);_R (K_2, G_2)) = \text{abs}((K_1 \cup K_2, G_2 \cup (G_1 \setminus K_2)))$
  \begin{align*}
  &\quad = \lambda D. G_2 \cup (G_1 \setminus K_2) \cup (D \setminus (K_1 \cup K_2)) \\
  &\quad = \lambda D. G_2 \cup (G_1 \setminus K_2) \cup ((D \setminus K_1) \cap (D \setminus K_2)) \\
  &\quad = \text{abs}((K_1, G_1);_R (K_2, G_2)) \quad (1)
  \end{align*}

$\text{abs}((K_1, G_1);_R (K_2, G_2))$

- $\text{abs}((K_1, G_1) \wedge_R (K_2, G_2)) = \text{abs}((K_1 \cap K_2, G_1 \cup G_2))$
  \begin{align*}
  &\quad = \lambda D. G_1 \cup G_2 \cup (D \setminus (K_1 \cap K_2)) \\
  &\quad = \text{abs}((K_1, G_1) \wedge_R (K_2, G_2)) \quad (2)
  \end{align*}

Therefore, $R$ for reaching definitions I is adequate.

Claim 1: No two different representations represent the same $\text{DFFun}$ function.

Proof: Let $r_1 = (K_1, G_1), r_2 = (K_2, G_2)$ and $r_1 \neq r_2$, which implies $K_1 \neq K_2$ and/or $G_1 \neq G_2$. Assume $\text{abs}(r_1) = \text{abs}(r_2)$. Then we have

$$\lambda D. G_1 \cup (D \setminus K_1) = \lambda D. G_2 \cup (D \setminus K_2)$$

which means, for all $D \in \text{Data}$,

$$G_1 \cup (D \setminus K_1) = G_2 \cup (D \setminus K_2)$$

Now there are two cases to consider: $K_1 = K_2$ and $K_1 \neq K_2$.

- For the first case, take $D$ to be the empty set. Then we get $G_1 = G_2$. But this conflicts with our initial assumption.
For the second case, without loss of generality, assume \( K_1 \setminus K_2 \neq \emptyset \). We can pick \( D \) to be \( \{n\} \) for some \( n \in \text{Node} \) such that \( n : x = e, x \in (K_1 \setminus K_2) \) and \( n \notin G_1 \). Then we end up with the equality

\[
G_1 \cup \emptyset = G_2 \cup \{n\}
\]

which is a conflict because \( G_1 \) does not include \( n \).

Therefore, there is always a value which conflicts our initial assumption, meaning representations uniquely represent functions.

□

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.

\[
\text{Data} = \mathcal{P}(\text{Exp}) \cup \{\top\}
\]

Sets in Data are ordered by set inclusion.

\[
\begin{align*}
\text{asgn}(n, x, e) &= \lambda E. (E \cup \{e' \mid e' \in \text{sub}(e)\}) \setminus E_x \\
\text{exp}(e) &= \lambda E. E \cup \{e' \mid e' \in \text{sub}(e)\}
\end{align*}
\]

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

The following seems an obvious representation.

\[
R = (\mathcal{P}(\text{Var}) \times \mathcal{P}(\text{Exp})) \cup \{\top\}
\]

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).

\[
\begin{align*}
id_R &= (\emptyset, \emptyset) \\
\text{asgn}_R(n, x, e) &= (\{x\}, \{e' \mid e' \in \text{sub}(e), x \notin \text{vars}(e')\} \\
\text{exp}_R(e) &= (\emptyset, \{e' \mid e' \in \text{sub}(e)\}) \\
(K_1, G_1) \land_R (K_2, G_2) &= (K_1 \cup K_2, G_2 \cup (G_1 \setminus K_2)) \\
(K_1, G_1) \setminus_R (K_2, G_2) &= (K_1 \cup K_2, G_1 \cap G_2) \\
\text{abs}(K, G) &= \lambda E. G \cup (E \setminus K)
\end{align*}
\]

where \( G \setminus K = \{e \in G \mid \text{none of the variables in } e \text{ occur in } K\} \).

However, this is not an adequate representation for the analysis. Consider the statement: if \((\text{cond}) \{a = \ldots; \ldots = a + b\} \text{ else } \{\} \). 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 $S$, $S_{\text{Annot}} = \{s_{\text{must}} | s \in S\} \cup \{s_{\text{sur}} | s \in S\}$. Also define the operation $\cdot$ on annotations: $\text{must}.\text{must} = \text{must}$ and otherwise $a.a' = \text{sur}$.

Then, this analysis is defined as follows:

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

$id_R = (\emptyset, \emptyset)$

$\text{asrn}_R(n, x, e) = (\{x\}, \{e_{\text{must}} | e' \in \text{sub}(e), x \notin \text{vars}(e')\})$

$\text{exp}_R(e) = (\emptyset, \{e_{\text{must}} | e' \in \text{sub}(e)\})$

$\text{abs}(K, G) = \lambda E.\{e | e_{\text{must}} \in G\} \cup$

$\{e | e_{\text{sur}} \in G, e \in E\} \cup$

$\{e | e \in E, e \notin G, \text{vars}(e) \cap K = \emptyset\}$

$\text{is} = \{s_{\text{must}} | s \in S\} \cup \{s_{\text{sur}} | s \in S\}$. Also define the operation $\cdot$ on annotations: $\text{must}.\text{must} = \text{must}$ and otherwise $a.a' = \text{sur}$.

The most interesting case is in the definition of semicolon, when $e_{\text{sur}} \in G_2$ 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 $\{(a), \{(a + b)_{\text{must}}\}\}$, and the false branch gives $\{(\emptyset, \emptyset)\}$. The meet of these values is $\{(a), \{(a + b)_{\text{must}}\}\}$. This value summarizes the effect of the if statement correctly: if $(a + b)_{\text{must}}$ 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.

**Theorem** $R$ for available expressions is an exact representation.

**Proof** The proof is similar to the corresponding proof for reaching definitions I in Section 4.1, but involves more cases to handle because of the annotations. \qed

### 4.3 Reaching Definitions II

Using annotations, we give an alternative representation for reaching definitions.
\[
R = (\text{Var} \rightarrow \mathcal{P}(\text{Node}_{\text{Annot}})) \cup \{\top_R\}
\]

\[
id_R = \lambda v. \emptyset
\]

\[
\text{asgn}(n, x, e) = (\lambda v. \emptyset)[x \mapsto \{n\}_{\text{must}}]
\]

\[
\text{exp}(e) = \lambda v. \emptyset
\]

\[
S_1; R S_2 = \lambda x. \text{let } p_m \leftarrow S_1(x), q_n \leftarrow S_2(x)
\]

\[
in \text{ if } n = \text{must} \text{ then } q_n \text{ else } (p \cup q)_m
\]

\[
S_1 \land_R S_2 = \lambda x. \text{let } p_m \leftarrow S_1(x), q_n \leftarrow S_2(x)
\]

\[
in (p \cup q)_{m,n}
\]

We assume that \(S(x)\) defaults to \(\emptyset_{\text{sur}}\). Finally, the abstraction function is

\[
\text{abs}(S) = \lambda D. \{n \in D \mid x = e \text{ and } S(x) = p_{\text{sur}}\} \cup \{n \in p \mid x = e \text{ and } S(x) = p_{\text{must}}\}
\]

**Theorem** \(R\) for reaching definitions II is an exact representation.

**Proof** The proof is similar to the corresponding proof for reaching definitions I in Section 4.1, but involves more cases to handle because of the annotations. \(\square\)

### 4.4 Constant Propagation

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

\[
\text{Data} = (\text{Var} \rightarrow \mathbb{Z}_{\text{sur}}) \cup \{\top_R\}
\]

Function values in \(\text{Data}\) are ordered under the usual pointwise ordering.

\[
\text{asgn}(n, x, e) = \lambda M. \text{if } \text{isConstant}(e, M) \text{ then } M[x \mapsto \text{consVal}(e, M)]
\]

\[
\text{else } M[x \mapsto \bot]
\]

\[
\text{exp}(e) = \lambda M. M
\]

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.

\[
R = \text{Var} \rightarrow \text{CS}_{\text{Annot}}
\]

\[
\text{CS} = \mathcal{P}(\text{Exp} \cup \{\bot\})
\]

We assume that, for all \(C \in \text{CS}\), if \(\bot \in C\) then \(C = \{\bot\}\); if there exist two integers \(i_1, i_2 \in C\) such that \(i_1 \neq i_2\) then \(C = \{\bot\}\). In the following definitions, \(M_1\) and \(M_2 \in \text{Data}\), \(C\) and \(C' \in \text{CS}, m\) and \(n \in \text{Annot}\).
We provide the sketch of the proof here. We first show that

\begin{align*}
\text{Theorem } R \text{ for constant propagation is an exact representation.}
\end{align*}

\textbf{Proof} We provide the sketch of the proof here. We first show that } R \text{ is adequate.}

\begin{itemize}
\item \(\text{abs}(\top_R) = \lambda M. \top_{\mathit{Data}} \) holds by definition.
\item \(\text{abs}(\text{id}_R) = \text{abs}(\lambda v. \emptyset_{\mathit{sur}}) = \lambda S. \lambda v. \text{let } C_{\mathit{must}} \leftarrow \text{semicolon}(S, S(v)_{\mathit{must}}, \emptyset_{\mathit{sur}})\)
\end{itemize}

\begin{align*}
\text{in if } C = \{i\} \text{ then } i \text{ else } \bot \\
\text{abs}(\text{asgn}_R(n, x, e)) = \text{abs}(\lambda v. \emptyset_{\mathit{sur}}[x \mapsto \{e\}_{\mathit{must}}]) \\
= \lambda S. \lambda v. \text{let } C_{\mathit{must}} \leftarrow \text{semicolon}(S, S(v)_{\mathit{must}}, (\lambda v. \emptyset_{\mathit{sur}}[x \mapsto \{e\}_{\mathit{must}}])(v))\)
\end{align*}

\begin{itemize}
\item \(\text{in if } C = \{i\} \text{ then } i \text{ else } \bot \\
\text{abs}(\text{exp}_R(e)) = \text{abs}(\lambda v. \emptyset_{\mathit{sur}}) = \lambda S. S = \text{exp}(e)
\end{itemize}

\begin{align*}
\text{abs}(M_1 \land_R M_2) = \lambda S. \lambda v. \text{let } C_{\mathit{must}} \leftarrow \text{semicolon}(S, S(v)_{\mathit{must}}, M_1(v) \land_R M_2(v))
\end{align*}

\begin{itemize}
\item \(\text{in if } C = \{i\} \text{ then } i \text{ else } \bot \\
= (1)
\end{itemize}

\begin{align*}
\text{abs}(M_1) \land \text{abs}(M_2) = \begin{cases} \\
\lambda S. \lambda v. \text{let } C_{\mathit{must}} \leftarrow \text{semicolon}(S, S(v)_{\mathit{must}}, M_1(v)) \\
\text{in if } C = \{i\} \text{ then } i \text{ else } \bot \\
\lambda S. \lambda v. \text{let } C_{\mathit{must}} \leftarrow \text{semicolon}(S, S(v)_{\mathit{must}}, M_2(v)) \\
\text{in if } C = \{i\} \text{ then } i \text{ else } \bot \\
= (2)
\end{cases}
\end{align*}
Showing that (1) = (2) is a straightforward case analysis based on the annotations of the values obtained from $M_1(v)$ and $M_2(v)$.

- \[ \text{abs}(M_1; R M_2) = \text{abs}(\lambda v. \text{semicolon}(M_1, M_1(v), M_2(v))) \]
  \[ = \lambda S. \lambda v. \text{let } C_{\text{must}} \leftarrow \text{semicolon}(S, S_{\text{must}}, \text{semicolon}(M_1, M_1(v), M_2(v))) \]
  \[ \text{in } \text{if } C = \{i\} \text{ then } i \text{ else } \bot \]
  \[ = (3) \]

and

- \[ \text{abs}(M_1) \cdot \text{abs}(M_2) = \left( \lambda S. \lambda v. \text{let } C_{\text{must}} \leftarrow \text{semicolon}(S, S_{\text{must}}, M_1(v)) \right) ; \]
  \[ \left( \lambda S. \lambda v. \text{let } C_{\text{must}} \leftarrow \text{semicolon}(S, S_{\text{must}}, M_2(v)) \right) \]
  \[ \text{in } \text{if } C = \{i\} \text{ then } i \text{ else } \bot \]
  \[ = (4) \]

Showing that (3) = (4) is a straightforward case analysis based on the annotations of the values obtained from $M_1(v)$ and $M_2(v)$.

Next step of the proof requires showing that the representations uniquely represent functions. This part in essence follows the same principles of the corresponding proof of reaching definitions I (Section 4.1). □

### 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:

- \[ F[n : \text{int}] = \lambda (\varphi, \eta, d). \text{let } d' \leftarrow \text{intDecl}(n, x)(d) \]
  \[ \text{in } (\varphi[n \mapsto d'], \eta, d') \]
- \[ F[n : \{n_1 : P\}] = \lambda (\varphi, \eta, d). \text{let } \eta' \leftarrow \text{map}(\text{beginScope}(n), \eta), \]
  \[ (\varphi_1, \eta_1, d_1) \leftarrow F[n_1 : P](\varphi, \eta', \text{beginScope}(n))(d) \]
  \[ \text{in } \text{let } d' \leftarrow \text{endScope}(d_1) \]
  \[ \text{in } (\varphi_1[n \mapsto d'], \text{map}(\text{endScope}, \eta_1), d') \]

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 of the same length, ordering is done pairwise among the type environments in the frames.

- \[ \text{Data} = \text{TySt} \cup \{\text{error}\} \]
- \[ \text{TySt} = (\{\text{Node} \cup \{\star\}\} \times \text{TyEv})^* \]
- \[ \text{TyEv} = \{\text{Var} \otimes \text{Type}\} \]
- \[ \text{Type} = \{\text{int, bool}\} \]
- \[ \text{asgn}(n, x, e) = \lambda \Gamma. \text{if } \text{type}(x, \Gamma) = \text{type}(e, \Gamma) \text{ then } \Gamma \text{ else error} \]
- \[ \text{intDecl}(n, x) = \lambda \Gamma. \text{if } \text{Gamma}(x) \text{ is defined then error else add}(\Gamma, x, \text{int}) \]
- \[ \text{exp}(e) = \lambda \Gamma. \text{if } \text{type}(e, \Gamma) = \text{bool} \text{ then } \Gamma \text{ else error} \]
- \[ \text{beginScope} = \lambda n. \lambda \Gamma. \Gamma', (n, \epsilon) \]
- \[ \text{endScope} = \lambda \Gamma. \text{let } [\Gamma', (n, \gamma)] \rightarrow \Gamma \text{ in } \Gamma' \]

The star in $\text{TySt}$ 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 $\text{Oblg}$ which can carry the three kinds of obligations.

$$R = \text{TySt} \times \text{Oblg}$$

$$\text{Oblg} = \mathcal{P}(\text{Var}^2 \cup (\text{Var} \times \text{Type}) \cup \text{Var}) \cup \{\text{error}\}$$

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

\[
\text{asgn}_R(n, x, e) = (\emptyset, \text{mkOblg}(x, e))
\]

\[
\text{exp}_R(e) = (\emptyset, \text{mkOblg}(e, \text{bool}))
\]

\[
\text{intDecl}_R(n, x) = ([(\ast, e[x \rightarrow \text{int}])], \{x\})
\]

where $\text{mkOblg}$ is an overloaded function defined by:

\[
\text{mkOblg}(x, y) = (x, y)
\]

\[
\text{mkOblg}(x, e_1 \oplus e_2) = \text{mkOblg}(e_1, \text{ltype}(\oplus)) \cup \text{mkOblg}(e_2, \text{rtype}(\oplus)) \cup (x, \text{type}(\oplus))
\]

\[
\text{mkOblg}(x, T) = (x, T)
\]

\[
\text{mkOblg}(e_1 \oplus e_2, T) = \begin{cases} 
\text{mkOblg}(e_1, \text{ltype}(\oplus)) \cup \text{mkOblg}(e_2, \text{rtype}(\oplus)) & \text{if type}(\oplus) = T \\
\text{error} & \text{else}
\end{cases}
\]

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

We define the meet and semicolon operations as

\[
(\Gamma, \Delta);_R (\Gamma', \Delta') = \begin{cases} 
\text{longestCommonPrefix}(\Gamma, \Gamma') & \text{if } \Gamma' \neq \emptyset \\
\emptyset \cup \Delta' & \text{else}
\end{cases}
\]

Finally, in the $\text{abs}$ function, if the obligations imposed by the representation are not satisfied by the incoming type stack, we return error, otherwise we just sequence the incoming type stack with the stack in the representation.

$$\text{abs}(\Gamma_R, \Delta_R) = \lambda\Gamma. \begin{cases} 
\text{let } (\Gamma', \Delta') \leftarrow (\emptyset);_R (\Gamma_R, \Delta_R) & \text{if } \Delta' = \emptyset \\
\text{else error}
\end{cases}$$
\[ B[\text{skip}] = \text{id} \]
\[ B[x = e] = \lambda(\eta, d). \text{asgn}(x, e)(d) \]
\[ B[\text{break } \ell] = \lambda(\eta, d). (\eta, \eta(\ell)) \]
\[ B[\ell : P] = \lambda(\eta, d). \text{let } (\eta', d') \leftarrow B[P][\eta[\ell \mapsto d], d] \text{ in } (\eta'[\ell \mapsto \top_{\text{Data}}], d') \]
\[ B[P_1; P_2] = B[P_2]; B[P_1] \]
\[ B[\text{if}(e) \ P_1 \text{ else } P_2] = \lambda(\eta, d). \text{let } (\eta_1, d_1) \leftarrow B[P_1][\eta, d] \]
\[ (\eta_2, d_2) \leftarrow B[P_2][\eta, d] \]
\[ \text{in } (\eta, \text{exp}(e)(d_1 \land d_2)) \]

Figure 12: Backward analysis framework

\[ R[\text{skip}] = (\top_{\text{Env}_R}, \text{id}_R) \]
\[ R[x = e] = (\top_{\text{Env}_R}, \text{asgn}_R(x, e)) \]
\[ R[\text{break } \ell] = (\top_{\text{Env}_R[\ell \mapsto \text{id}_R]}, \top_R) \]
\[ R[\ell : P] = \text{let } (\eta, r) \leftarrow R[P] \]
\[ \text{in } (\eta[\ell \mapsto \top_R], r \land_R \eta(\ell)) \]
\[ R[P_1; P_2] = \text{let } (\eta_1, r_1) \leftarrow R[P_1], (\eta_2, r_2) \leftarrow R[P_2] \]
\[ \text{in } (\eta_1 \land_R (\eta_2 ;_R r_1), r_2 ;_R r_1) \]
\[ R[\text{if}(e) \ P_1 \text{ else } P_2] = (R[P_1] \land_R R[P_2]) ;_R \text{exp}_R(e) \]

Figure 13: Representation for framework of Figure 12.

**Theorem** \( R \) for type checking is an exact representation.

**Proof** This proof follows the same structure of the corresponding proof for constant propagation, and is omitted. \( \square \)

## 5 Backward Analysis Framework

We can define a similar framework for backward analysis, although break statements significantly complicate matters. Due to space constraints, we only provide the intermediate framework here. It is presented in Figure 12, and the representation in Figure 13. The abstraction function is

\[ \text{abs}_E(\eta_R, r) = \lambda(\eta, d). \text{abs}(r)(d) \land \bigwedge_{\ell \in \text{Label}} \text{abs}(\eta_R(\ell))(\eta(\ell)) \]

**Theorem** For all \( P \), if the \( DFFun \) functions are distributive (i.e. \( f(d \land d') = f(d) \land f(d') \)), \( \text{abs}_E(R[P]) = B[P] \).
Proof} The proof is by induction on the structure of $P$.

- **skip**:
  \[
  \text{abs} \left( \mathcal{R}[\text{skip}] \right) = \text{abs} \left( (\top_{\text{Env}_R}, \text{id}_R) \right) \\
  = \lambda (\eta', d'). (\eta', \text{abs}(\text{id}_R) d' \land \bigwedge_{\ell' \in \text{Label}} \text{abs}(\top_{\text{Env}_R}(\ell'))(\eta'(\ell'))) \\
  = \lambda (\eta', d'). (\eta', d' \land \bigwedge_{\ell' \in \text{Label}} \top_{\text{Data}}) \\
  = \lambda (\eta', d'). (\eta', d') \\
  = B[\text{skip}]
  \]

- **$x = e$**
  \[
  \text{abs} \left( \mathcal{R}[x = e] \right) = \text{abs} \left( (\top_{\text{Env}_R}, \text{asgn}_R(x, e)) \right) \\
  = \lambda (\eta', d'). (\eta', \text{abs}(\text{asgn}_R(x, e)) d' \land \bigwedge_{\ell' \in \text{Label}} \text{abs}(\top_{\text{Env}_R}(\ell'))(\eta'(\ell'))) \\
  = \lambda (\eta', d'). (\eta', \text{asgn}(x, e) d' \land \bigwedge_{\ell' \in \text{Label}} \top_{\text{Data}}) \\
  = \lambda (\eta', d'). (\eta', \text{asgn}(x, e) d') \\
  = B[x = e]
  \]

- **break $\ell$**
  \[
  \text{abs} \left( \mathcal{R}[\text{break } \ell] \right) = \text{abs} \left( (\top_{\text{Env}_R}[\ell \mapsto \text{id}_R], \top_R) \right) \\
  = \lambda (\eta', d'). (\eta', \text{abs}(\text{id}_R) d' \land \bigwedge_{\ell' \in \text{Label}} \text{abs}(\top_{\text{Env}_R}[\ell \mapsto \text{id}_R](\ell'))(\eta'(\ell'))) \\
  = \lambda (\eta', d'). (\eta', \top_{\text{Data}} \land \text{abs}(\text{id}_R)(\eta'(\ell))) \\
  = \lambda (\eta', d'). (\eta', \eta'(\ell)) \\
  = B[\text{break } \ell]
  \]

- **$\ell : P$**
  Let $(\eta, r) = \mathcal{R}[P]$. By the induction hypothesis we have
  \[
  B[P] = \text{abs} \left( \mathcal{R}[P] \right) = \text{abs} \left( (\eta, r) \right) \\
  = \lambda (\eta', d'). (\eta', \text{abs}(r)d' \land \bigwedge_{\ell' \in \text{Label}} \text{abs}(\eta'(\ell'))(\eta'(\ell'))) \\
  \]

Now we work on $\ell : P$.

\[
\text{abs} \left( \mathcal{R}[\ell : P] \right) = \text{abs} \left( (\eta[\ell \mapsto \top_R], r \land R \eta(\ell)) \right) \\
= \lambda (\eta', d'). (\eta', \text{abs}(r \land R \eta(\ell))d' \land \bigwedge_{\ell' \in \text{Label}} \text{abs}(\eta[\ell \mapsto \top_R](\ell'))(\eta'(\ell'))) \\
= \lambda (\eta', d'). (\eta', \text{abs}(r)d' \land \text{abs}(\eta(\ell))d' \land \bigwedge_{\ell' \in \text{Label}} \text{abs}(\eta[\ell \mapsto \top_R](\ell'))(\eta'(\ell'))) \\
= \lambda (\eta', d'). (\eta', \text{abs}(r)d' \land \text{abs}(\eta(\ell))d' \land \bigwedge_{\ell' \in \text{Label}, \ell' \neq \ell} \text{abs}(\eta'(\ell'))(\eta'(\ell'))) \quad (1)
\]

32
And

\[ B[\ell : P] = \lambda(q', d'). \text{let } (\eta_1, d_1) \leftarrow B[P](\eta'[\ell \mapsto d'], d') \]

in \((\eta_1[\ell \mapsto \top_{\text{Data}}], d_1)\)

= \(\lambda(q', d'). \text{let } (\eta_1, d_1) \leftarrow (\eta'[\ell \mapsto d'], \text{abs}(r)d' \land \bigwedge_{\ell' \in \text{Label}} \text{abs}(\eta(\ell'))(\eta'[\ell \mapsto d'[\ell']))) \]

in \((\eta_1[\ell \mapsto \top_{\text{Data}}], d_1)\)

= \(\lambda(q', d'). (\eta'[\ell \mapsto \top_{\text{Data}}], \text{abs}(r)d' \land \bigwedge_{\ell' \in \text{Label}} \text{abs}(\eta(\ell'))(\eta'[\ell \mapsto d'[\ell']))) \)

= \(\lambda(q', d'). (\eta'[\ell \mapsto \top_{\text{Data}}], \text{abs}(r)d' \land \bigwedge_{\ell' \in \text{Label}, \ell' \neq \ell} \text{abs}(\eta(\ell'))(\eta'[\ell'])) \)

Because we require all the programs to be legal, incoming environment \(\eta'\) has \(\ell\) mapped to \(\top_{\text{Data}}\). So

\[ (2) = \lambda(q', d'). (\eta', \text{abs}(r)d' \land \bigwedge_{\ell' \in \text{Label}, \ell' \neq \ell} \text{abs}(\eta(\ell'))(\eta'[\ell'])) \]

= (1)

- **\(P_1; P_2\)**

Let \((\eta_1, r_1) = R[P_1]\) and \((\eta_2, r_2) = R[P_2]\). By the induction hypothesis we have

\[ B[P_1] = \text{abs}_E(R[P_1]) = \text{abs}_E((\eta_1, r_1)) = \lambda(q', d', r_1)d' \land \bigwedge_{\ell' \in \text{Label}} \text{abs}(\eta_1(d'))(\eta'(\ell')) \]

and

\[ B[P_2] = \text{abs}_E(R[P_2]) = \text{abs}_E((\eta_2, r_2)) = \lambda(q', d', r_2)d' \land \bigwedge_{\ell' \in \text{Label}} \text{abs}(\eta_2(d'))(\eta'(\ell')) \]

Now we work on \(P_1; P_2\).

\[ \text{abs}_E(R[P_1; P_2]) = \text{abs}_E((\eta_1 \land_R (\eta_2; R r_1), r_2; R r_1)) \]

= \(\lambda(q', d', (\eta', \text{abs}(r_2; R r_1)d' \land \bigwedge_{\ell' \in \text{Label}} \text{abs}(\eta_1 \land_R (\eta_2; R r_1))(\ell'))(\eta'(\ell'))) \)

= \(\lambda(q', d', (\eta', \text{abs}(r_2; R r_1)d' \land \bigwedge_{\ell' \in \text{Label}} (\text{abs}(\eta_1(\ell'))(\eta'(\ell')) \land \text{abs}(\eta_2(\ell'); R r_1)(\eta'(\ell')))) \)

(3)
And
\[ B[P_1; P_2] = \lambda(q', d'). (B[P_2]; B[P_1])(q', d') \]

We note that we used the distributivity property above.

1. if $e \in P_1$ else $P_2$

Let $(\eta_1, r_1) = R[P_1]$ and $(\eta_2, r_2) = R[P_2]$. By the induction hypothesis we have

\[ B[P_1] = \text{abs}_E(R[P_1]) = \text{abs}_E((\eta_1, r_1)) = \lambda(q', d'). (q', \text{abs}(r_1)) d' \wedge \bigwedge_{t' \in \text{Label}} \text{abs}((\eta_1)'((q')')) \]

\[ B[P_2] = \text{abs}_E(R[P_2]) = \text{abs}_E((\eta_2, r_2)) = \lambda(q', d'). (q', \text{abs}(r_2)) d' \wedge \bigwedge_{t' \in \text{Label}} \text{abs}((\eta_2)'((q')')) \]

Now we work on if $e \in P_1$ else $P_2$.

\[ \text{abs}_E(R[i f(e) P_1 else P_2]) = \text{abs}_E(((\eta_1 \land R \eta_2): R \exp_2(e), (r_1 \land R r_2): R \exp(e))) = \lambda(q', d'). (q', \exp(e)(r_1 \land R r_2) d' \wedge \bigwedge_{t' \in \text{Label}} \text{abs}((\eta_1)'((q')') \land \text{abs}((\eta_2)'((q')'))) \]

(4)

Let $(q_1', d_1') = B[P_1](q', d')$ and $(\eta_2, d_2') = B[P_2](q', d')$. Then

\[ B[i f(e) P_1 else P_2] = \lambda(q', d'). (q', \exp(e)(d_1 \land d_2)) = \lambda(q', d'). (q', \exp(e)(\text{abs}(r_1) d' \wedge \bigwedge_{t' \in \text{Label}} \text{abs}((\eta_1)'((q')'))) \land \bigwedge_{t' \in \text{Label}} \text{abs}((\eta_2)'((q')'))) \]

\[ = \lambda(q', d'). (q', \exp(e)(\text{abs}(r_1 \land R r_2) d' \wedge \bigwedge_{t' \in \text{Label}} \text{abs}((\eta_1)'((q')') \land \text{abs}((\eta_2)'((q')'))) \land \bigwedge_{t' \in \text{Label}} \text{abs}((\eta_1)'((q')'))) \]

\[ = \lambda(q', d'). (q', \exp(e)(\text{abs}(r_1 \land R r_2) d' \wedge \bigwedge_{t' \in \text{Label}} \text{abs}((\eta_1)'((q')') \land \text{abs}((\eta_2)'((q')'))) \land \bigwedge_{t' \in \text{Label}} \text{abs}((\eta_1)'((q')') \land \text{abs}((\eta_2)'((q')'))) \]

\[ = (4) \]
\( B[n : \text{skip}] = \text{id} \)
\( B[n : x = e] = \lambda(\varphi, \eta, d). (\varphi[n \mapsto \text{asgn}(x, e)(d)], \eta, \text{asgn}(x, e)(d)) \)
\( B[n : \text{break } \ell.] = \lambda(\varphi, \eta, d). (\varphi[n \mapsto \eta(\ell)], \eta, \eta(\ell)) \)
\( B[n : (\ell : n_1 : P)] = \lambda(\varphi, \eta, d). \text{let } (\varphi', \eta', d') \leftarrow B[n_1 : P](\varphi, \eta[\ell \mapsto d], d) \) in \((\varphi'[n \mapsto d'], \eta'[\ell \mapsto \top_{\text{data}}], d') \)
\( B[n : (n_1 : P_1; n_2 : P_2)] = \lambda(\varphi, \eta, d). \text{let } (\varphi', \eta', d') \leftarrow (B[n_2 : P_2]; B[n_1 : P_1])(\varphi, \eta, d) \) in \((\varphi'[n \mapsto d'], \eta', d') \)
\( B[n : \text{if } e \text{ n}_1 : P_1 \text{ else } n_2 : P_2] = \lambda(\varphi, \eta, d). \text{let } (\varphi_1, \eta_1, d_1) \leftarrow B[n_1 : P_1](\varphi, \eta, d) \) \((\varphi_2, \eta_2, d_2) \leftarrow B[n_2 : P_2](\varphi, \eta, d) \) in \(((\varphi_1 \cup \varphi_2)[n \mapsto \text{exp}(e)(d_1 \land d_2)], \eta, \text{exp}(e)(d_1 \land d_2)) \)

Figure 14: Backward analysis framework

We note that we used the distributivity property above.

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 9). However, defining \( R \) 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

\( R: \text{Pgm} \rightarrow (\text{Node} \rightarrow (\text{Env}_R \times R)) \times \text{Env}_R \times R \)

Analogous to how \( R[P_1; P_2] \) in the forward representation function of Figure 10 updates the node-map for each node in \( P_1 \) and \( P_2 \), \( R[L : P] \) and \( R[P_1; P_2] \) in the full backward representation function update each mapping in their node-maps as well. Full versions of \( B \) and \( R \) are given in Figures 14 and 15, respectively. In Figure 15, closeLabel is defined as

\( \text{closeLabel}(\ell, \varphi) = \lambda n. \text{let } (\eta, r) \leftarrow \varphi(n) \) in \((\eta[\ell \mapsto \top_R], r \land_R \eta(\ell)) \)

The \textbf{abs} function for the full backward framework is defined as

\[
\text{abs}_F(\varphi, \eta, r) = \lambda(\varphi', \eta', d'). \text{let } \varphi'' \leftarrow \lambda n. \text{let } (\tilde{\eta}, \tilde{r}) \leftarrow \varphi(n) \\
\text{in } \text{abs}(\tilde{r})(d') \land \bigwedge_{\ell \in \text{Label}} \text{abs}(\tilde{\eta}(\ell))(\eta'(\ell)) \\
\text{in } (\varphi' \cup \varphi'', \eta', \text{abs}(r)(d') \land \bigwedge_{\ell \in \text{Label}} \text{abs}(\eta(\ell))(\eta'(\ell))]
\]

**Theorem** For all \( P \), if the DFFun functions are distributive (i.e. \( f(d \land d') = f(d) \land f(d') \)), \( \text{abs}_F(R[P]) = B[P] \).

**Proof** The proof is similar to, but notationally more complex than, the corresponding proof for the intermediate framework, on page 32.
\[ R[n \text{ skip}] = \{(n \mapsto (\top_{Env}, \text{id}_R))\}, \top_{Env}, \text{id}_R \]

\[ R[n \text{ break } \ell] = \{(n \mapsto (\top_{Env}, \text{assign}_R(x, e))), \top_{Env}, \text{assign}_R(x, e)\} \]

\[ R[n : x = e] = \{(n \mapsto (\top_{Env}, \text{assign}_R(x, e))), \top_{Env}, \text{assign}_R(x, e)\} \]

\[ R[n : (\ell : n_1 : P)] = \text{let } (\varphi, \eta, r) \mapsto R[P] \]

\[ \text{in } (\text{closeLabel}(\ell, \varphi[n \mapsto (\eta, r)]), \eta[\ell \mapsto \top_{R}], r \land_R \eta(\ell)) \]

\[ R[n : (n_1 : P_1 ; n_2 : P_2)] = \text{let } (\varphi_1, \eta_1, r_1) \mapsto R[n_1 : P_1], (\varphi_2, \eta_2, r_2) \mapsto R[n_2 : P_2] \]

\[ \text{in } (\lambda n'. \text{if } \varphi_2(n') \text{ defined then } \varphi_2(n') \]

\[ \text{if } \varphi_1(n') \text{ defined then let } (\eta', r') \mapsto \varphi_1(n') \]

\[ \text{in } (\eta' \land_R (\eta_2 ; r_2 ; r'), r_2 ; r') \]

\[ \text{if } n' = n \text{ then } (\eta_1 \land_R (\eta_2 ; r_1), r_2 ; r_1), \]

\[ R[n : \text{if } (e) n_1 : P_1 \text{ else } n_2 : P_2] = \text{let } (\varphi_1, \eta_1, r_1) \mapsto R[n_1 : P_1], (\varphi_2, \eta_2, r_2) \mapsto R[n_2 : P_2] \]

\[ \text{in } ((\varphi_1 \lor \varphi_2)[n \mapsto (r_1 \land_R (r_2)); \text{R exp}_R(e)], \]

\[ (\eta_1 \land_R (\eta_2 ; r_2); \text{R exp}_R(e)), \]

\[ (r_1 \land_R (r_2); \text{R exp}_R(e)) \]

\[ \text{Figure 15: Representation for framework of Figure 14.} \]

### 5.1 Live Variables

Data is defined as

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

and is ordered by reverse set inclusion.

\[ \text{assign}(n, x, e) = \lambda L.(L \setminus \{x\}) \cup \text{vars}(e) \]

\[ \text{exp}(e) = \lambda L.L \cup \text{vars}(e) \]

\[ R = \mathcal{P}\text{(Var)}^2 \]

\[ \text{assign}_R(n, x, e) = (\{x\}, \text{vars}(e)) \]

\[ \text{exp}_R(e) = (\emptyset, \text{vars}(e)) \]

Definitions of \( \text{id}_R, ;_R, \land_R \) and \( \text{abs} \) 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.

\[ \text{assign}(n, x, e) = \lambda E.(E \setminus E_x) \cup \text{sub}(e) \]

\[ \text{assign}_R(n, x, e) = (\{x\}, \{e'_{\text{must}} \mid e' \in \text{sub}(e)\}) \]

### 6 Performance

We are interested in the \textit{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 \textit{base analysis}); the other is to
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 speed-up. 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.

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** are 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].

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

<table>
<thead>
<tr>
<th>Sample Program</th>
<th>HotSpot</th>
<th>libgcj</th>
<th>Kaffe</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RD</td>
<td>CP</td>
<td>TC</td>
</tr>
<tr>
<td>Big-plug</td>
<td>2.10</td>
<td>1.19</td>
<td>3.65</td>
</tr>
<tr>
<td>Small-plug-A</td>
<td>2.17</td>
<td>1.12</td>
<td>3.50</td>
</tr>
<tr>
<td>Small-plug-B</td>
<td>2.40</td>
<td>1.14</td>
<td>2.97</td>
</tr>
<tr>
<td>Two-plug</td>
<td>1.67</td>
<td>1.17</td>
<td>1.66</td>
</tr>
<tr>
<td>Fib1 ([7])</td>
<td>1.10</td>
<td>1.07</td>
<td>1.31</td>
</tr>
<tr>
<td>Fib2 ([7])</td>
<td>1.23</td>
<td>1.16</td>
<td>0.67</td>
</tr>
<tr>
<td>Sort ([5])</td>
<td>1.48</td>
<td>1.21</td>
<td>1.92</td>
</tr>
<tr>
<td>Huffman ([7])</td>
<td>1.11</td>
<td>1.29</td>
<td>0.30</td>
</tr>
<tr>
<td>Marshalling 1 ([2])</td>
<td>12.37</td>
<td>3.93</td>
<td>28.27</td>
</tr>
<tr>
<td>Marshalling 2 ([2])</td>
<td>2.01</td>
<td>1.75</td>
<td>16.01</td>
</tr>
</tbody>
</table>

We use our staged analysis.

37
• **Marshalling 1** is part of a program that produces customized serializers in Java [2]; characteristics much like Big-plug.

• **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 of GPCE ’06 for their helpful comments.

References


