Staging Static Analyses for Program Generation (Extended Version)

BARIS AKTEMUR, SAM KAMIN and MICHAEL KATELMAN
University of Illinois at Urbana-Champaign

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. Our framework is defined on abstract syntax trees (AST), because program fragments appear as AST's. We give a translation from source-level code to an intermediate representation (IR) and show that our staging methodology is applicable at the IR-level, too.

Categories and Subject Descriptors: F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; D.3.4 [**Programming Languages**]: Processors—*Code generation*

General Terms: Languages, Performance

Additional Key Words and Phrases: runtime program generation, program 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 meta-program) [Kamin et al. 2003; Oiwa et al. 2001; Poletto et al. 1997; Czarnecki et al. 2004]. 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

Authors' address: 201 N. Goodwin, Urbana, IL 61801 USA
Email addresses: {aktemur,kamin,katelman}@cs.uiuc.edu
Partial support for this work was received from NSF under grant CCR-0306221. This is the extended version of the paper published in GPCE 2006 with the same title.

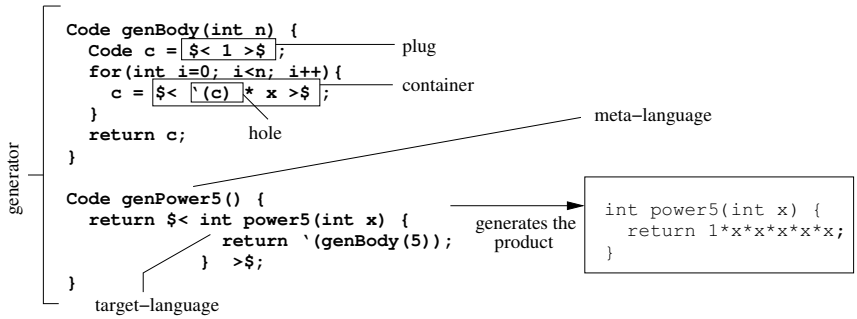

Fig. 1. Terminology for program generators. When a fragment fills in a hole, we call it a *plug*. Note that a fragment is never used as a plug until all of its holes have been filled.

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 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. We give a translation from the source program to an intermediate representation in Section 7 to show that our framework is also applicable to IR-level code. The proofs of the main theorems given in the paper are available at the Appendix.

The contributions of this paper are four-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. (4) We show that our framework for staging analyses is also applicable at the intermediate representation (IR) level by giving a translation from source code to IR-level code.

This paper is an expanded version of [Kamin et al. 2006].

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 [Sharir and Pnueli 1981] introduced the idea of summarizing the analysis of an entire procedure. Rountev, Kagan and Marlowe [Rountev et al. 2006] 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 [Reps et al. 1995] 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 [Kramer et al. 1994] partition programs and analyze each partition to produce a summary of its effect on the program as a whole.

In *hybrid* analysis [Marlowe and Ryder 1990], 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* [Smith et al. 2003], 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. [Chambers 2002]. 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 [Aho et al. 1986] 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 AST-based static analysis, as in standard control flow graph-based analysis, each node is, in the end, assigned a value from a lattice *Data* of dataflow values. However, in the AST case, the assignment is performed by a traversal of the tree (rather than by a worklist algorithm), possibly including multiple traversals of some subtrees. Thus, each node has input data (received from its neighbor to the left or right, depending upon whether we are considering a forward or backward analysis) and output data. A key difference is that the AST contains nodes that represent entire subtrees, so that the calculation of output data from input data may be the composition of many smaller calculations. Whereas in a CFG, the function from input data to output data given by any one node is relatively small, in an AST it can be very large. (AST's do, of course, contain those "small nodes" as well; they just have *more* nodes overall.)

Given a (hole-free) subtree, taken out of context, we cannot say what its value is because we do not know its input data. We do know the function from *Data* to *Data* that it represents. Now suppose we have representations for functions that arise in a particular analysis. Then we can handle staging of the analysis like this: For all AST's, calculate this function for every *maximal hole-free subtree*. This leaves a prefix of the original AST, with some subtrees pruned and replaced by function representations. (For hole-free plugs, the entire tree is replaced by its function representation.) At run time, the code-generating code associated with each fragment [Kamin et al. 2003] is accompanied by the fragment's representation tree. When the fragments are combined to form the entire program, the static analysis can be performed on the combined tree. Time is saved because there is no need to traverse the program's entire AST, and also because there may be optimizations applicable to the function representations.

The staging process is illustrated in Figure 2.

In this section, we present our framework in three steps. The first framework covers the language without break statements; the second adds break statements; and the third — the full framework — adds the feature of assigning a dataflow value to each node rather than just to the root. For each of these three frameworks, 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

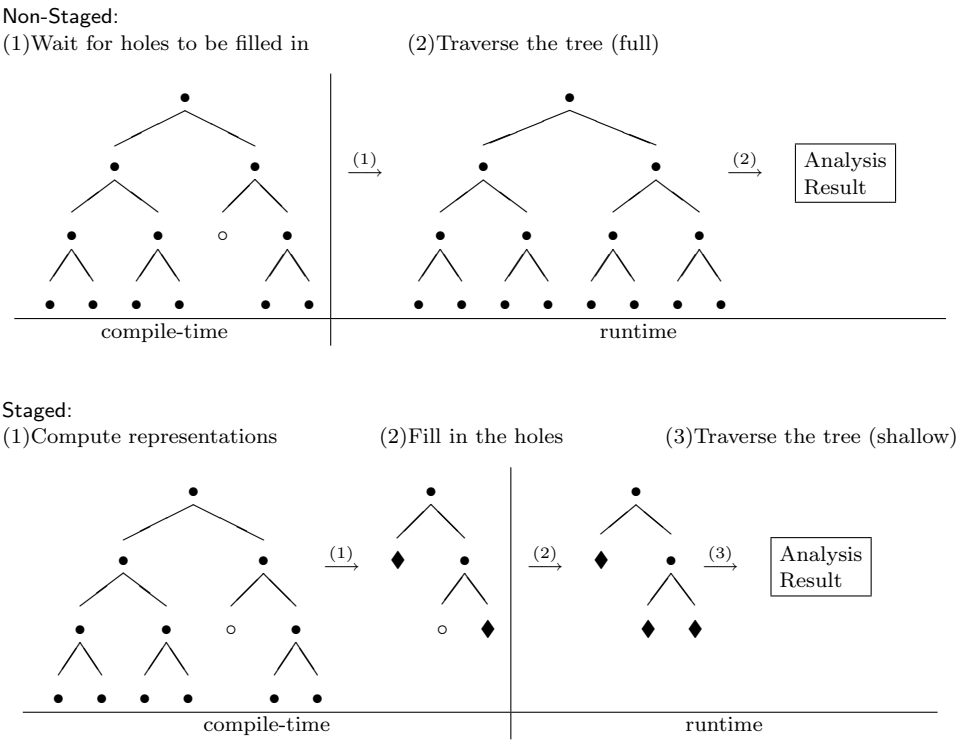

Fig. 2. Staging a data flow analysis. • is a regular AST node, ∘ is a hole, and ⧫ is a representation.

$e \in \text{Exp}$
$x \in \text{Var}$
$\ell \in \text{Label}$
$P \in \text{Pgm} ::= x = e \mid \mathsf{skip} \mid \mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2 \mid P_1; P_2$
$\mid \mathsf{while}(e)\,\mathsf{do}\ P \mid \ell : P \mid \mathsf{break}\ \ell$

Fig. 3. The language treated in this paper

dataflow values only for the root of an AST; the final framework calculates values at each node within an AST.

Figure 3 shows the abstract syntax of the language we treat in this paper. We use a Java-like language for concrete syntax. 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 $Data \to Data$ (confined to functions that preserve $\top_{Data}$).

3.1 Simple Control Structures

Our first framework (Figure 4) treats a subset of the full language, programs with only sequencing and conditionals. $\mathcal{F}$ assigns an element of *DFFun* to every program.

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

Fig. 4. First framework.

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:

$$id : DFFun$$
$$asgn : \mathrm{Var} \times \mathrm{Exp} \to DFFun$$
$$exp : \mathrm{Exp} \to DFFun$$
$$; : DFFun \times DFFun \to DFFun$$
$$\wedge : DFFun \times DFFun \to DFFun$$

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) at 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.

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

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

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

Definition 3.1. Suppose R is a set with the following values and functions ($\top_R$ is not used until the next subsection):

$$\begin{array}{lll} \top_R & : & R \\ id_R & : & R \\ asgn_R & : & \mathrm{Var} \times \mathrm{Exp} \to R \end{array} \qquad \begin{array}{lll} exp_R & : & \mathrm{Exp} \to R \\ ;_R & : & R \times R \to R \\ \wedge_R & : & R \times R \to R \end{array}$$

R is an *adequate representation* of a dataflow problem if there is a homomorphism

$$\mathbf{abs} : R \to DFFun$$

$$\mathcal{R}[\![\mathsf{skip}]\!] = id_R$$
$$\mathcal{R}[\![x = e]\!] = asgn_R(x, e)$$
$$\mathcal{R}[\![P_1; P_2]\!] = \mathcal{R}[\![P_1]\!] ;_R \mathcal{R}[\![P_2]\!]$$
$$\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = exp_R(e) ;_R (\mathcal{R}[\![P_1]\!] \wedge_R \mathcal{R}[\![P_2]\!])$$

Fig. 5. Representation function for the first framework.

from $(R, \top_R, id_R, asgn_R, exp_R, ;_R, \wedge_R)$ to $(DFFun, \top_{DFFun}, id, asgn, exp, ;, \wedge)$. Specifically, this requires

$$\begin{array}{l}\mathbf{abs}(\top_R) = \top_{DFFun} = \lambda d.\top_{Data}\\ \mathbf{abs}(id_R) = id\\ \mathbf{abs}(asgn_R(x,e)) = asgn(x,e)\\ \mathbf{abs}(exp_R(e)) = exp(e)\\ \mathbf{abs}(r ;_R r') = \mathbf{abs}(r); \mathbf{abs}(r')\\ \mathbf{abs}(r \wedge_R r') = \mathbf{abs}(r) \wedge \mathbf{abs}(r')\end{array}$$

Define $\mathcal{R} : \mathrm{Pgm} \to R$ to be the function in Figure 5.

THEOREM 3.2. *If R is an adequate representation, then for all P,* $\mathbf{abs}(\mathcal{R}[\![P]\!]) = \mathcal{F}[\![P]\!]$.

PROOF. A trivial structural induction. □

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. The operations on this representation are[1]

$$\begin{array}{l}id_R = (\emptyset, \emptyset)\\ asgn_R(x,e) = (\{x\}, vars(e))\\ exp_R(e) = (\emptyset, vars(e))\\ (D,U) ;_R (D',U') = (D \cup D', U \cup (U' \setminus D))\\ (D,U) \wedge_R (D',U') = (D \cap D', U \cup U')\end{array}$$

The **abs** function is defined as

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

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

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

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

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

Fig. 6. $\mathcal{F}^R$ for the first framework.

```
// ({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 3, we included while statements in our language. They can be defined using a maximal fixpoint in the usual way:

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

If we were to include $\mathcal{R}[\![\mathsf{while}(e)\ \mathsf{do}\ P]\!]$ in Figure 5, we would define it as $while_R(\mathcal{R}[\![e]\!], \mathcal{R}[\![P]\!])$, where $while_R$ is a function specific to each analysis. We will not mention this further, however, because in each of our examples, the function $while_R$ is not very interesting: $while_R(r_1, r_2)$ is either $r_1 ;_R r_2 ;_R r_1$ or $r_1 ;_R r_2 ;_R r_1 ;_R r_2 ;_R r_1$. That is, only a fixed number of iterations of the loop body is required.

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, when applied to the full framework, to be more efficient; see page 13. The difference is that $\mathcal{R}$ is a purely bottom-up algorithm, while $\mathcal{F}$ is more top-down; the situation is similar to the use of an accumulator parameter in functional programs, which can turn a quadratic algorithm into a linear one [Ireland and Bundy 1999].

Define $\mathcal{F}^R : \mathrm{Pgm} \to R \to R$ to be the function in Figure 6, with the relevant operations defined as follows:

$$id^R = id$$
$$asgn^R(x, e) = \lambda r.r ;_R asgn_R(x, e)$$
$$exp^R(e) = \lambda r.r ;_R exp_R(x, e)$$
$$f \wedge^R g = \lambda r.fr \wedge_R gr$$

Definition 3.3. Two representation values, r and r', are equivalent, denoted $r \equiv r'$, if $\mathbf{abs}(r) = \mathbf{abs}(r')$.

THEOREM 3.4. *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 by induction on the structure of P. Details provided in the Appendix. □

COROLLARY 3.5. $\mathcal{F}^R[\![P]\!]id_R \equiv \mathcal{R}[\![P]\!]$.

$$\mathcal{F}[\![\mathsf{skip}]\!] = id$$

$$\mathcal{F}[\![x = e]\!] = \lambda(\eta, d).(\eta, asgn(x, e)(d))$$

$$\mathcal{F}[\![\mathsf{break}\ \ell;]\!] = \lambda(\eta, d).(\eta[\ell \mapsto d \wedge \eta(\ell)], \top_{Data})$$

$$\mathcal{F}[\![\ell : P]\!] = \lambda(\eta, d).\ \mathrm{let}\ (\eta_1, d_1) \leftarrow \mathcal{F}[\![P]\!](\eta, d)\ \mathrm{in}\ (\eta_1[\ell \mapsto \top_{Data}], d_1 \wedge \eta_1(\ell))$$

$$\mathcal{F}[\![P_1; P_2]\!] = \mathcal{F}[\![P_1]\!]; \mathcal{F}[\![P_2]\!]$$

$$\mathcal{F}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = \lambda(\eta, d).\ \mathrm{let}\ (\eta_1, d_1) \leftarrow \mathcal{F}[\![P_1]\!](\eta, exp(e)(d))\ (\eta_2, d_2) \leftarrow \mathcal{F}[\![P_2]\!](\eta, exp(e)(d))\ \mathrm{in}\ (\eta_1, d_1) \wedge (\eta_2, d_2)$$

Fig. 7. Framework with break statements

Proof.

$$\begin{aligned}\mathbf{abs}(\mathcal{F}^R[\![P]\!]id_R) &= \mathbf{abs}(id_R ;_R \mathcal{R}[\![P]\!]) \\ &= \mathbf{abs}(id_R); \mathbf{abs}(\mathcal{R}[\![P]\!]) \\ &= id; \mathbf{abs}(\mathcal{R}[\![P]\!]) \\ &= \mathbf{abs}(\mathcal{R}[\![P]\!])\end{aligned}$$

□

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, as depicted in Figure 2. The first stage calculates values of R, using $\mathcal{F}^R$, and the second, run-time, stage uses $\mathcal{F}$ to complete the analysis.

3.2 Break Statements

We expand our analysis now to labelled statements and break-to-label statements. 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 the paper, 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 $Env = \mathrm{Label} \to Data$. Now the incoming and outgoing values are pairs:

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

The extended analysis is shown in Figure 7. *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.

To explain Figure 7: 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 $\mathsf{break}\ \ell$ statement prior to reaching P; since there may be more than one, $\eta(\ell)$ gives a conservative approximation by taking

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

Fig. 8. Representation for framework of Figure 7

the meet of all those paths. Thus, if P is $\mathsf{break}\ \ell$, then d is incorporated into the outgoing environment by taking $d \wedge \eta(\ell)$. Furthermore, the "normal exit" from P is $\top_{Data}$, which ensures that any statement directly following P will be ignored (since, for any statement Q, $\mathcal{F}[\![Q]\!]$ preserves $\top_{Data}$ in its second argument). Now consider labelled statements. $\mathcal{F}[\![\ell : P]\!](\eta, d)$ first calculates $\mathcal{F}[\![P]\!](\eta, d)$. A normal exit from $\ell : P$ can be a normal exit from P or a break to ℓ, so we take the meet of these two values. Furthermore, the binding of ℓ in the environment is reset to $\top_{Data}$, since a subsequent statement could be labelled ℓ.

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 $Env \times Data \to Env \times Data$. Define $Env_R = \text{Label} \to R$. Then

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

Figure 8 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:

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

we have the following theorem.

THEOREM 3.6. *If R is adequate, then for any legal program P, $\mathbf{abs}_E(\mathcal{R}[\![P]\!]) = \mathcal{F}[\![P]\!]$.*

$$\mathcal{F}^R[\![\mathsf{skip}]\!] = id^R$$

$$\mathcal{F}^R[\![x = e]\!] = \lambda(\eta, r).(\eta, asgn^R(x, e)r)$$

$$\mathcal{F}^R[\![\mathsf{break}\ \ell;]\!] = \lambda(\eta, r).(\eta[\ell \mapsto r \wedge_R \eta(\ell)], \top_R)$$

$$\mathcal{F}^R[\![\ell : P]\!] = \lambda(\eta, r).\ \mathrm{let}\ (\eta_1, r_1) \leftarrow \mathcal{F}^R[\![P]\!](\eta, r)\ \mathrm{in}\ (\eta_1[\ell \mapsto \top_R], r_1 \wedge_R \eta_1(\ell))$$

$$\mathcal{F}^R[\![P_1; P_2]\!] = \mathcal{F}^R[\![P_1]\!]; \mathcal{F}^R[\![P_2]\!]$$

$$\mathcal{F}^R[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = \lambda(\eta, r).\ \mathrm{let}\ (\eta_1, r_1) \leftarrow \mathcal{F}^R[\![P_1]\!](\eta, exp^R(e)r)\ (\eta_2, r_2) \leftarrow \mathcal{F}^R[\![P_2]\!](\eta, exp^R(e)r)\ \mathrm{in}\ (\eta_1, r_1) \wedge_R (\eta_2, r_2)$$

Fig. 9. $\mathcal{F}^R$ with break statements.

PROOF. The proof is by induction on the structure of P. Details are provided in the Appendix. □

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

$$\mathcal{F}^R : \mathrm{Pgm} \to E_R \to E_R$$

is defined as given in Figure 9 where $asgn^R$ and exp^R are exactly the same as in the previous section; id^R has the same definition but different type.

THEOREM 3.7. *Let P be a legal program, and $(\eta, r) = \mathcal{R}[\![P]\!]$. Then, for all η' and r', as long as $\eta'(L) = \top_R$ for any label L that occurs in P, we have*

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

PROOF. The proof is by induction on the structure of P. Details are provided in the Appendix. □

COROLLARY 3.8. $\mathcal{F}^R[\![P]\!](\top_{Env_R}, id_R) \equiv \mathcal{R}[\![P]\!]$.

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

$$\begin{aligned}\mathcal{F}^R[\![P]\!](\top_{Env_R}, id_R) &\equiv (\lambda\ell'.\top_{Env_R}(\ell') \wedge_R (id_R ;_R \eta(\ell')), id_R ;_R r)\\ &= (\lambda\ell'.\top_R \wedge_R (id_R ;_R \eta(\ell')), id_R ;_R r)\end{aligned}$$

which means

$$\begin{aligned}&\mathbf{abs}_E(\mathcal{F}^R[\![P]\!](\top_{Env_R}, id_R)) = \mathbf{abs}_E((\lambda\ell'.\top_R \wedge_R (id_R ;_R \eta(\ell')),\ id_R ;_R r))\\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\top_R \wedge_R (id_R ;_R \eta(\ell')))d'', \mathbf{abs}(id_R ;_R r)d'')\\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\top_R)d'' \wedge (\mathbf{abs}(id_R); \mathbf{abs}(\eta(\ell')))d'', (\mathbf{abs}(id_R); \mathbf{abs}(r))d'')\\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta(\ell'))d'', \mathbf{abs}(r)d'')\\ &= \mathbf{abs}_E((\eta, r))\\ &= \mathbf{abs}_E(\mathcal{R}[\![P]\!])\end{aligned}$$

□

Again, $\equiv$ can be replaced by $=$ for all the analyses we present in this paper.

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 ↦ (∅, ∅)}, ⊤)
}
```

Note that in the topmost node, `w` is in the defined set for normal exit even though it is not defined in both branches of the if-statement. This is because the flow reaches the end of the if-statement only if the then-branch where `w` is defined is taken.

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? Etc.

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

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

We also change the type of *asgn*:

$$asgn : Node \times \mathrm{Var} \times \mathrm{Exp} \to DFFun$$

for cases (such as reaching definitions) where $Node$ is contained within $Data$. In cases such as uninitialized variables, the first argument is ignored. The full forward analysis is shown in Figure 10.

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

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

The abstraction function becomes:

$$\mathbf{abs}_F : F_R \to (NodeMap \times Env \times Data \to NodeMap \times Env \times Data)$$
$$\mathbf{abs}_F(\varphi_R, \eta_R, r) \;=\; \lambda(\varphi', \eta', d').(\varphi' \cup (\lambda n.\mathbf{abs}(\varphi_R(n))d'),\ \lambda\ell.\eta'(\ell) \wedge \mathbf{abs}(\eta_R(\ell))d',\ \mathbf{abs}(r)d')$$

Representations are calculated by function $\mathcal{R}$ as given in Figure 11.

THEOREM 3.9. *If R is adequate, then for any legal program P, $\mathbf{abs}_F(\mathcal{R}[\![P]\!]) = \mathcal{F}[\![P]\!]$.*

PROOF. The proof is similar to the proof for the intermediate framework (Theorem 3.6). □

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

$$\mathcal{F}[\![n : x = e]\!] = \lambda(\varphi,\eta,d).\text{let } d' \leftarrow asgn(n,x,e)(d) \text{ in } (\varphi[n \mapsto d'],\eta,d')$$

$$\mathcal{F}[\![n : \mathsf{break}\ \ell]\!] = \lambda(\varphi,\eta,d).(\varphi[n \mapsto \top_{Data}],\eta[\ell \mapsto d \wedge \eta(\ell)],\top_{Data})$$

$$\mathcal{F}[\![n : (\ell : (n_1 : P))]\!] = \lambda(\varphi,\eta,d).\ \text{let } (\varphi_1,\eta_1,d_1) \leftarrow \mathcal{F}[\![n_1 : P]\!](\varphi,\eta,d) \text{ in } (\varphi_1[n \mapsto d_1 \wedge \eta_1(\ell)],\eta_1[\ell \mapsto \top_{Data}],d_1 \wedge \eta_1(\ell))$$

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

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

Fig. 10. Forward analysis framework

$$\mathcal{R}[\![n : \mathsf{skip}]\!] = (\{n \mapsto id_R\},\top_{Env_R},id_R)$$

$$\mathcal{R}[\![n : x = e]\!] = (\{n \mapsto asgn_R(n,x,e)\},\top_{Env_R},asgn_R(n,x,e))$$

$$\mathcal{R}[\![n : \mathsf{break}\ \ell]\!] = (\{n \mapsto \top_R\},\top_{Env_R}[\ell \mapsto id_R],\top_R)$$

$$\mathcal{R}[\![n : (\ell : (n_1 : P))]\!] = \text{let } (\varphi,\eta,r) \leftarrow \mathcal{R}[\![n_1 : P]\!] \text{ in } (\varphi[n \mapsto r \wedge_R \eta(\ell)],\eta[\ell \mapsto \top_R],r \wedge_R \eta(\ell))$$

$$\mathcal{R}[\![n : (n_1 : P_1;\quad n_2 : P_2)]\!] = \text{let } (\varphi_1,\eta_1,r_1) \leftarrow \mathcal{R}[\![n_1 : P_1]\!],\ (\varphi_2,\eta_2,r_2) \leftarrow \mathcal{R}[\![n_2 : P_2]\!] \text{ in } (\lambda n'.\ \text{if } \varphi_1(n') \text{ defined then } \varphi_1(n') \text{ if } \varphi_2(n') \text{ defined then } r_1;_R\,\varphi_2(n') \text{ if } n' = n \text{ then } r_1;_R\,r_2\ ,\ \eta_1 \wedge_R (r_1;_R\,\eta_2),\ r_1;_R\,r_2)$$

$$\mathcal{R}[\![n : \mathsf{if}(e)\ n_1 : P_1\ \mathsf{else}\ n_2 : P_2]\!] = \text{let } (\varphi_1,\eta_1,r_1) \leftarrow \mathcal{R}[\![n_1 : P_1]\!],\ (\varphi_2,\eta_2,r_2) \leftarrow \mathcal{R}[\![n_2 : P_2]\!] \text{ in } (exp_R(e);_R\,((\varphi_1 \cup \varphi_2)[n \mapsto (r_1 \wedge_R r_2)]),\ exp_R(e);_R\,(\eta_1 \wedge_R \eta_2),\ exp_R(e);_R\,(r_1 \wedge_R r_2))$$

Fig. 11. Representation for framework of Figure 10.

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

THEOREM 3.10. *Let P be a legal program and $(\varphi,\eta,r) = \mathcal{R}[\![P]\!]$. Then for all φ',η' and r', as long as $\eta'(L) = \top_R$ for any label L that occurs in P, we have*

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

PROOF. The proof is similar to the proof for the intermediate framework (Theorem 3.7). □

The importance of $\mathcal{F}^R$ can now be explained. $\mathcal{R}$ calculates the node map φ bottom-up. Suppose $\mathcal{R}[\![P]\!] = (\varphi,\eta,r)$, and consider $\varphi(n)$, where n is a node in P. $\varphi(n)$ says how to calculate a data value at n given input data at P; that is, it represents the computation from the start of P to n. In calculating $\mathcal{R}[\![P_1;P_2]\!]$, the subcomputation $\mathcal{R}[\![P_2]\!]$ returns a node map φ_2 representing computations *within* P_2. The result for $P_1;P_2$ has to give values for nodes *in* P_2 that represent the

```
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;
     }
```

Fig. 12. The example program with numbered nodes.

computation *starting at* P_1. Thus, it not only produces values for each node in P_1, but also calculates *new* values for every node in P_2. Extending this reasoning to a list of statements $P_1; \ldots; P_n$, we see that values for all the nodes in P_n will be calculated n times, for all the nodes in P_{n-1} $n-1$ times, etc. Thus, the complexity of $\mathcal{R}[\![P]\!]$ is quadratic in the size of the P. $\mathcal{F}$ uses, in effect, an accumulator, passing φ through the entire tree, and thus calculates a value for each node just once.

Our previous example with numbered nodes is in Figure 12. We show the value of $\mathcal{R}[\![P]\!]$ 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:

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

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 [Sharir and Pnueli 1981]). 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. In general, we have the chance to query the data of only selected nodes without analyzing the entire tree, which can have a salutary effect on the run-time performance of the analysis. 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 else-branch ends in a break, control can only reach the end of this statement by taking the then-branch.

Again, staging is not fundamentally different in this more complicated framework. One new wrinkle is that a single plug cannot be used to fill in 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 (RD)

The reaching definitions (RD) 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. Representations for this analysis have been given in [Marlowe and Ryder 1990; Reps et al. 1995; Rountev et al. 2006].

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

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

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

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

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

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

$$\begin{array}{l} id_R = (\emptyset, \emptyset) \\ asgn_R(n, x, e) = (\{x\}, \{n\}) \\ exp_R(e) = (\emptyset, \emptyset) \\ (K_1, G_1);_R (K_2, G_2) = (K_1 \cup K_2, G_2 \cup (G_1 \setminus K_2)) \\ (K_1, G_1) \wedge_R (K_2, G_2) = (K_1 \cap K_2, G_1 \cup G_2) \end{array}$$

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

where $G \setminus K = \{n \in G \mid n \text{ is not the definition of some } x \in K\}$.

THEOREM 4.1. *R for* RD *is an exact representation.*

PROOF. To show that a representation is exact (i.e. **abs** is an isomorphism between R and *DFFun*), we need to prove two claims:

(1) R is adequate (i.e. **abs** defines a homomorphism)
(2) **abs** is injective. (i.e. $\forall r_1, r_2 \in R, r_1 \neq r_2 \Rightarrow \mathbf{abs}(r_1) \neq \mathbf{abs}(r_2)$)

Claim 1: R for RD is adequate.
Proof:

- $\mathbf{abs}(\top_R) = \lambda D.\top_{Data}$ holds by definition.
- $\mathbf{abs}(id_R) = \mathbf{abs}((\emptyset, \emptyset)) = \lambda D.\emptyset \cup (D \setminus \emptyset) = \lambda D.D = id$
- $\mathbf{abs}(asgn_R(n, x, e)) = \mathbf{abs}((\{x\}, \{n\})) = \lambda D.\{n\} \cup (D \setminus \{x\}) = \lambda D.\{n\} \cup (D \setminus D_x) = asgn(n, x, e)$
- $\mathbf{abs}(exp_R(e)) = \mathbf{abs}((\emptyset, \emptyset)) = \lambda D.\emptyset \cup (D \setminus \emptyset) = \lambda D.D = exp(e)$

$$
\begin{aligned}
\cdot\ \mathbf{abs}((K_1, G_1);_R (K_2, G_2)) &= \mathbf{abs}((K_1 \cup K_2, G_2 \cup (G_1 \setminus K_2))) \\
&= \lambda D.G_2 \cup (G_1 \setminus K_2) \cup (D \setminus (K_1 \cup K_2)) \\
&= \lambda D.G_2 \cup (G_1 \setminus K_2) \cup ((D \setminus K_1) \cap (D \setminus K_2)) \qquad (1)
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{abs}((K_1, G_1)); \mathbf{abs}((K_2, G_2)) &= (\lambda D.G_1 \cup (D \setminus K_1)); (\lambda D.G_2 \cup (D \setminus K_2)) \\
&= \lambda D.G_2 \cup ((G_1 \cup (D \setminus K_1)) \setminus K_2) \\
&= \lambda D.G_2 \cup (G_1 \setminus K_2) \cup ((D \setminus K_1) \setminus K_2) \\
&= \lambda D.G_2 \cup (G_1 \setminus K_2) \cup ((D \setminus K_1) \cap (D \setminus K_2)) \\
&= (1)
\end{aligned}
$$

$$
\begin{aligned}
\cdot\ \mathbf{abs}((K_1, G_1) \wedge_R (K_2, G_2)) &= \mathbf{abs}((K_1 \cap K_2, G_1 \cup G_2)) \\
&= \lambda D.G_1 \cup G_2 \cup (D \setminus (K_1 \cap K_2)) \qquad (2)
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{abs}((K_1, G_1)) \wedge \mathbf{abs}((K_2, G_2)) &= (\lambda D.G_1 \cup (D \setminus K_1)) \wedge (\lambda D.G_2 \cup (D \setminus K_2)) \\
&= \lambda D.G_1 \cup (D \setminus K_1) \cup G_2 \cup (D \setminus K_2) \\
&= \lambda D.G_1 \cup G_2 \cup (D \setminus (K_1 \cap K_2)) \\
&= (2)
\end{aligned}
$$

Therefore, R for RD is adequate.

Claim 2: **abs** is injective.

Proof: By contradiction. 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 $\mathbf{abs}(r_1) = \mathbf{abs}(r_2)$. 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 $D \in Data$,

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

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 $n \in Node$ 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 impossible because G_1 does not include n.

□

4.2 Available Expressions (AE)

Available expressions (AE) 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.

$$E \in Data = \mathcal{P}(\mathrm{Exp}) \cup \{\top\}$$

Sets in $Data$ are ordered by set inclusion.

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

where E_x is the set of expressions in E that contain x, and $sub(e)$ 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 R value (K, G) represents that G is the set of expressions made available by a statement, and K is the set of variables defined by that statement (so that the statement kills any expressions containing those variables).

$$\begin{array}{l} id_R = (\emptyset, \emptyset) \\ asgn_R(n, x, e) = (\{x\}, \{e' \mid e' \in sub(e), x \notin vars(e')\} \\ exp_R(e) = (\emptyset, \{e' \mid e' \in sub(e)\}) \\ (K_1, G_1);_R (K_2, G_2) = (K_1 \cup K_2, G_2 \cup (G_1 \setminus K_2)) \\ (K_1, G_1) \wedge_R (K_2, G_2) = (K_1 \cup K_2, G_1 \cap G_2) \end{array}$$

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

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

However, this is not an adequate representation for the analysis. Consider the statement: if $(cond)$ $\{a = \ldots; \ldots = a + b\}$ 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 representation of 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 4.2. For set S, $S_{Annot} = \{s_{must} \mid s \in S\} \cup \{s_{sur} \mid s \in S\}$. Also define the operation "$\cdot$" on annotations: $must \cdot must = must$ and otherwise $\alpha \cdot \alpha' = sur$, where α, α' are annotations.

Our analysis uses the set Exp_{Annot}. The annotation *sur* stands for the case when there is a path in the fragment that lets the incoming expression *sur*vive. The annotation *must* stands for the case when there is no such path, so that the statement itself *must* define the expression if it is to be available. The dot operation encapsulates the notion that an expression can survive a conditional statement as long as it can survive at least one of the branches.
Then, this analysis is defined as follows:

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

$$\begin{array}{l} id_R = (\emptyset, \emptyset) \\ asgn_R(n, x, e) = (\{x\}, \{e'_{must} \mid e' \in sub(e), x \notin vars(e')\} \\ exp_R(e) = (\emptyset, \{e'_{must} \mid e' \in sub(e)\}) \end{array}$$

$$
\begin{array}{l}
(K_1, G_1);_R (K_2, G_2) = (K_1 \cup K_2, \\
\quad \{e_{must} \,|\, e_{must} \in G_2\} \cup \\
\quad \{e_\alpha \,|\, e_{sur} \in G_2,\ e_\alpha \in G_1\} \cup \\
\quad \{e_{sur} \,|\, e_{sur} \in G_2,\ e_\alpha \notin G_1,\ vars(e) \cap K_1 = \emptyset\} \cup \\
\quad \{e_\alpha \,|\, e_\alpha \in G_1,\ e'_\alpha \notin G_2,\ vars(e) \cap K_2 = \emptyset\}) \\
(K_1, G_1) \wedge_R (K_2, G_2) = (K_1 \cup K_2, \\
\quad \{e_{\alpha\cdot\alpha'} \,|\, e_\alpha \in G_1,\ e'_\alpha \in G_2\} \cup \\
\quad \{e_{sur} \,|\, e_\alpha \in G_1,\ e'_\alpha \notin G_2,\ vars(e) \cap K_2 = \emptyset\} \cup \\
\quad \{e_{sur} \,|\, e_\alpha \in G_2,\ e'_\alpha \notin G_1,\ vars(e) \cap K_1 = \emptyset\})
\end{array}
$$

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

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 $(a+b)$ is in the incoming available set, then it will be in the resulting available set.

THEOREM 4.3. *R for* AE *is an exact representation.*

PROOF. The proof is similar to the proof for RD in Section 4.1. □

4.3 Reaching Definitions II (RD2)

Using annotations, we give an alternative representation for reaching definitions. We will call this analysis RD2. Here we annotate sets of definitions of a variable; a *must* subscript indicates that the set includes all possible definitions of the variable, while a *sur* subscript indicates that there is some path in this statement through which a previous definition of the variable might survive.

Let $N \in \mathcal{P}(Node)$ in the following definitions.

$$S \in R = (\mathrm{Var} \to \mathcal{P}(Node)_{Annot}) \cup \{\top_R\}$$

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

$$
\begin{array}{l}
S_1;_R S_2 = \lambda x.\,\mathrm{let}\ N_\alpha \leftarrow S_1(x), N'_{\alpha'} \leftarrow S_2(x) \\
\qquad\qquad \mathrm{in\ if}\ \alpha' = must\ \mathrm{then}\ N'_{\alpha'}\ \mathrm{else}\ (N \cup N')_\alpha \\
S_1 \wedge_R S_2 = \lambda x.\mathrm{let}\ N_\alpha \leftarrow S_1(x), N'_{\alpha'} \leftarrow S_2(x) \\
\qquad\qquad \mathrm{in}\ (N \cup N')_{\alpha\cdot\alpha'}
\end{array}
$$

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

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

where $D \in Data = \mathcal{P}(Node) \cup \{\top\}$ as before.

THEOREM 4.4. *R for* RD2 *is an exact representation.*

PROOF. The proof is similar to the proof for RD in Section 4.1. □

4.4 Constant Propagation (CP)

The framework can be instantiated for constant propagation (CP) with the following definitions. For simplicity we consider only integers as constant values, and assume that the expressions in the language are arithmetic operations. A graph-based representation for this analysis can be found in [Reps et al. 1995; Sagiv et al. 1996]. That representation requires that the set of program variables be available to construct the representation graphs. By the nature of our context, we cannot, and do not, make such an assumption.

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

Function values in $Data$ are ordered under the usual pointwise ordering.

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

where $isConstant(e, M)$ returns true if the expression e can be shown to have a constant value based on the values kept in the constant map M, and $consVal(e, M)$ returns that constant value[2].

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.

$$\begin{aligned} R &= Var \to CS_{Annot} \\ CS &= P(Exp \cup \{\perp\}) \end{aligned}$$

Implicitly, a $C \in CS$ is normalized to $\{\perp\}$ if it contains $\perp$ or two distinct integers.

As in the previous cases, the annotations are used to preserve information in conditionals. A *must* annotation on a set of expressions indicates that the variable they define is definitely assigned one of those expressions; a *sur* annotation indicates that some other definition may apply to that variable (but may, of course, assign the same value to it that these expressions do).

$$\begin{aligned} id_R &= \lambda v.\emptyset_{sur} \\ asgn_R(n, x, e) &= (\lambda v.\emptyset_{sur})[x \mapsto \{e\}_{must}] \\ exp_R(e) &= \lambda v.\emptyset_{sur} \end{aligned}$$

$$\begin{aligned} M_1 \wedge_R M_2 &= \lambda x.M_1(x) \wedge_R M_2(x) \\ &= \lambda x.\text{let } C_\alpha \leftarrow M_1(x), C'_{\alpha'} \leftarrow M_2(x) \\ &\qquad \text{in } (C \cup C')_{\alpha \cdot \alpha'} \end{aligned}$$

[2]Precise definitions of *isConstant* and *consVal* depend on the kind of constant propagation chosen (e.g. literal, copy, linear, or non-linear constant propagation [Sagiv et al. 1996]).

$$M_1;_R M_2 = \lambda x.\mathit{semicolon}(M_1, M_1(x), M_2(x))$$
$$\mathit{semicolon}(M, C_\alpha, C'_{must}) = \mathit{update}(M, C')_{must}$$
$$\mathit{semicolon}(M, C_\alpha, C'_{sur}) = (\mathit{update}(M, C') \cup C)_\alpha$$

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, if $M(y) = \{w, z\}$ and $C = \{y+1\}$, $\mathit{update}(M, C)$ returns $\{w+1, z+1\}$.

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

$$\begin{aligned}\mathbf{abs}(M) = \lambda S.\lambda x.&\text{let } C_{must} \leftarrow \mathit{semicolon}(S, S(x)_{must}, M(x))\\ &\text{in if } C = \{i\} \text{ then } i \text{ else } \bot\end{aligned}$$

THEOREM 4.5. *R for* CP *is an exact representation.*

PROOF. We provide the sketch of the proof here. We first show that R is adequate.

- $\mathbf{abs}(\top_R) = \lambda M.\top_{Data}$ holds by definition.
- $$\begin{aligned}\mathbf{abs}(\mathit{id}_R) = \mathbf{abs}(\lambda v.\emptyset_{sur}) &= \lambda S.\lambda v.\,\text{let } C_{must} \leftarrow \mathit{semicolon}(S, S(v)_{must}, \emptyset_{sur})\\ &\qquad\text{in if } C = \{i\} \text{ then } i \text{ else } \bot\\ &= \lambda S.\lambda v.\,\text{let } C_{must} \leftarrow S(v)_{must}\\ &\qquad\text{in if } C = \{i\} \text{ then } i \text{ else } \bot\\ &= \lambda S.\lambda v.S(v)\\ &= \mathit{id}\end{aligned}$$
- $$\begin{aligned}\mathbf{abs}(\mathit{asgn}_R(n, x, e)) &= \mathbf{abs}(\lambda v.\emptyset_{sur}[x \mapsto \{e\}_{must}])\\ &= \lambda S.\lambda v.\,\text{let } C_{must} \leftarrow \mathit{semicolon}(S, S(v)_{must}, (\lambda v.\emptyset_{sur}[x \mapsto \{e\}_{must}])(v))\\ &\qquad\text{in if } C = \{i\} \text{ then } i \text{ else } \bot\\ &= \lambda S.\lambda v.\begin{cases} S(v) & \text{if } v \neq x\\ \mathit{update}(S, \{e\}) & \text{if } v = x\end{cases}\\ &= \lambda S.S[x \mapsto \text{if } \mathit{isConstant}(e, S) \text{ then } \mathit{consVal}(e, M) \text{ else } \bot]\\ &= \mathit{asgn}(n, x, e)\end{aligned}$$
- $\mathbf{abs}(\mathit{exp}_R(e)) = \mathbf{abs}(\lambda v.\emptyset_{sur}) = \lambda S.S = \mathit{exp}(e)$
- $$\begin{aligned}\mathbf{abs}(M_1 \wedge_R M_2) &= \mathbf{abs}(\lambda v.M_1(v) \wedge_R M_2(v))\\ &= \lambda S.\lambda v.\,\text{let } C_{must} \leftarrow \mathit{semicolon}(S, S(v)_{must}, M_1(v) \wedge_R M_2(v))\\ &\qquad\text{in if } C = \{i\} \text{ then } i \text{ else } \bot\\ &= (1)\end{aligned}$$

  and

  $$\begin{aligned}\mathbf{abs}(M_1) \wedge \mathbf{abs}(M_2) = &\left(\begin{array}{l}\lambda S.\lambda v.\text{let } C_{must} \leftarrow \mathit{semicolon}(S, S(v)_{must}, M_1(v))\\ \qquad\text{in if } C = \{i\} \text{ then } i \text{ else } \bot\end{array}\right) \wedge\\ &\left(\begin{array}{l}\lambda S.\lambda v.\text{let } C_{must} \leftarrow \mathit{semicolon}(S, S(v)_{must}, M_2(v))\\ \qquad\text{in if } C = \{i\} \text{ then } i \text{ else } \bot\end{array}\right)\\ = &\ (2)\end{aligned}$$

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

$= (3)$

and

$$\mathbf{abs}(M_1);\mathbf{abs}(M_2) = \begin{pmatrix}\lambda S.\lambda v.\text{let } C_{must} \leftarrow semicolon(S, S(v)_{must}, M_1(v)) \\ \quad \text{in if } C = \{i\} \text{ then } i \text{ else } \bot\end{pmatrix}; \begin{pmatrix}\lambda S.\lambda v.\text{let } C_{must} \leftarrow semicolon(S, S(v)_{must}, M_2(v)) \\ \quad \text{in if } C = \{i\} \text{ then } i \text{ else } \bot\end{pmatrix}$$
$$= (4)$$

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 RD (Section 4.1). □

4.5 Loop Invariants (LI)

We take the definition of a loop invariant as given in [Aho et al. 1986]:

> A statement inside a loop L is invariant if all the operands of the statement either are constant, have all their reaching definitions outside L, or have exactly one reaching definition, and that definition is an invariant statement in L.

As a simplification we will compute the invariance information of a statement only with respect to the innermost loop that surrounds the statement. We assume that there exists a function $loop(P)$ to obtain that innermost loop. We also assume that the reaching definitions have been computed and are available for use in LI: $\mathsf{RD}(n, y)$ gives the definitions of y that reach the node n; $\mathsf{RD}_R(n)$ gives the RD representation for the node n. (Alternatively, RD can be computed on-the-fly.)

Data is defined as a map containing the invariance information:

$$I \in Data = (Node \nrightarrow Bool) \cup \{\top\}$$

Data is ordered as follows:

$$I \sqsubseteq I' \text{ if } \forall n.I'(n) \text{ is undefined or } I'(n) \sqsubseteq I(n) \text{ in the boolean lattice, where false } \sqsubset \text{ true}$$

The definitions of *exp* and *asgn* are

$$exp(e) = id$$
$$asgn(n, x, e) = \lambda I.I[n \mapsto \forall y \in vars(e).\mathsf{isInv}(n, y, I)]$$

where isInv is defined as

$$\mathsf{isInv}(n, y, I) = loop(n) \text{ is defined and } ((\forall d \in \mathsf{RD}(n, y)\,.\,d \text{ is not contained in } loop(n)) \vee (\exists d\,.\,\mathsf{RD}(n, y) = \{d\} \text{ and } I(d)))$$

isInv directly follows from the definition of loop invariants: Invariance of a node is dependent on (1) the reaching definitions of a variable, or (2) a single node if there is only one reaching definition. This also hints at the definition of a representation:

$$R = (Node \twoheadrightarrow IV) \cup \{\top_R\}$$
$$IV = \mathcal{P}(Var \cup Node \cup \{\text{true}, \text{false}\})$$

The invariance information we keep per node, called IV, is a set that contains variables, nodes, true, or false, where a variable stands for the dependence (1), and a node stands for the dependence (2). The intuition is that if the IV set of a statement contains a node, that node must become invariant for the statement to be invariant; if the set contains a variable, the reaching definitions of that variable must eventually satisfy the conditions for making the statement an invariant. If the available information is enough to conclude that the statement is not invariant, the set contains false; if the only item in the set is true, the statement is invariant. In other words, the IV is used to delay the computation of invariance of a statement. As more information becomes available, IV is updated. Below are the necessary definitions.

$$exp_R(e) = id_R$$
$$asgn_R(x, e, n) = \{n \mapsto vars(e) \cup \{\text{true}\}\}$$
$$M_1 \wedge_R M_2 = \lambda n.\, M_1(n) \cup M_2(n)$$
$$M_1 ;_R M_2 = M_1 \uplus fix\big(\lambda M_2'.\lambda n\,.\,\{\mathsf{update}_n(s, M_1 \uplus M_2') | s \in M_2(n)\}\big)$$

where $\uplus$ is domain disjoint union of functions, and update is defined as

$$\begin{array}{l}
\mathsf{update}_n(\text{true}, M) = \text{true} \\
\mathsf{update}_n(\text{false}, M) = \text{false} \\
\mathsf{update}_n(n', M) = \text{if } M(n') = \{\text{true}\} \text{ //}n' \textit{ is invariant} \\
\qquad\qquad \text{then true} \\
\qquad\qquad \text{else if false} \in M(n') \text{ //}n' \textit{ is not invariant} \\
\qquad\qquad\quad \text{then false} \\
\qquad\qquad\quad \text{else } n' \text{ //}\textit{cannot update yet, so keep } n' \\
\mathsf{update}_n(x, M) = \text{let } (K, G) \leftarrow \mathsf{RD}_R(n) \text{ in} \\
\qquad \text{if there are no definitions of } x \text{ in } G \\
\qquad \text{then } x \text{ //}\textit{cannot update yet, so keep } x \\
\qquad \text{else if there are multiple definitions of } x \text{ in } G \\
\qquad\quad \text{then if all the definitions are outside } \textit{loop}(n) \text{ then true} \\
\qquad\qquad \text{else false} \\
\qquad \text{else if there is a single definition } d \text{ of } x \text{ in } G \\
\qquad\quad \text{then if } d \text{ is outside } \textit{loop}(n) \text{ then true} \\
\qquad\qquad \text{else if } x \in K \\
\qquad\qquad\quad \text{then if } M(d) = \{\text{true}\} \text{ then true else } d \\
\qquad\qquad\quad \text{else false}
\end{array}$$

Finally, we give the definition of the **abs** function

$$\mathbf{abs}(M) = \lambda I.\, I \wedge I' \text{ where } I' \text{ is}$$

$$fix\Big(\begin{array}{l}\lambda I_2.\lambda n.\ \text{let } B \leftarrow \{\mathsf{isInv}_R(n, s, I \wedge I_2) | s \in M(n)\} \\ \qquad\qquad \text{in } B = \{\text{true}\}\end{array}\Big)$$

and isInv_R is a function that returns a boolean value:

$$\mathcal{F}[\![n : \text{int } x]\!] = \lambda(\varphi, \eta, d).\, \text{let } d' \leftarrow \mathit{intDecl}(n, x)(d) \\ \text{in } (\varphi[n \mapsto d'], \eta, d')$$

$$\mathcal{F}[\![n : \{n_1 : P\}]\!] = \lambda(\varphi, \eta, d).\, \text{let } \eta' \leftarrow \mathit{map}(\mathit{beginScope}(n), \eta), \\ (\varphi_1, \eta_1, d_1) \leftarrow \mathcal{F}[\![n_1 : P]\!](\varphi, \eta', \mathit{beginScope}(n)(d)) \\ \text{in let } d' \leftarrow \mathit{endScope}(d_1) \\ \text{in } (\varphi_1[n \mapsto d'], \mathit{map}(\mathit{endScope}, \eta_1), d')$$

Fig. 13. Framework extended with declarations and scope.

$$\mathcal{R}[\![n : \text{int } x]\!] = \mathit{intDecl}_R(n, x)$$
$$\mathcal{R}[\![n : \{n_1 : P\}]\!] = \mathit{endScope}_R(\mathcal{R}[\![n_1 : P]\!]))$$

Fig. 14. Representation for declarations and scope. Because $\mathcal{R}$ is strictly bottom-up, there is no $\mathit{beginScope}_R$.

$$\mathsf{isInv}_R(n, \text{true}, I) = \mathit{loop}(n) \text{ is defined}$$
$$\mathsf{isInv}_R(n, \text{false}, I) = \text{false}$$
$$\mathsf{isInv}_R(n, n', I) = \mathit{loop}(n) \text{ is defined and } I(n')$$
$$\mathsf{isInv}_R(n, x, I) = \mathsf{isInv}(n, x, I)$$

Note that there is recursion in the definitions of $;_R$ and **abs**. The recursion terminates because it is not possible to have in a valid program two nodes which are solely dependent on each other, or a node whose invariance is only dependent on itself.

4.6 Type Checking (TC)

To discuss type checking, we will extend the language with declarations and scope. The framework definitions for these constructs are in Figure 13 and 14. (A fuller discussion of this analysis can be found in [Katelman 2006].)

To be concrete, we will assume simple typing — no polymorphism, either ad-hoc or parametric — and a Java-like rule for scope: declarations are scoped, but the same name cannot be declared twice within one method.

Before proceeding to the technical development, we give some intuition for the representation for this analysis; for this discussion, we will ignore scope. The question, as for all these analyses, is: what might a fragment contain that could affect the analysis when the fragment is placed in context? For one thing, the fragment might contain declarations, which would affect type-checking for all subsequent statements; we will record these in a type environment. Beyond that, it may contain uses which might lead to type errors. These are of three kinds: (1) uses which constrain a variable to be of a particular type (e.g. `x+1`); (2) uses which constrain a variable to be of the same type as another variable (e.g. `x = y`); and (3) declarations, which entail that there cannot be another declaration for the same name, either before or after this one. (Note that we need to remember that a variable has been declared, even when the actual declaration goes out of scope.) Thus, the representation for this analysis contains a type environment and a set of "obligations," which are constraints imposed by this fragment:

$$R = TySt \times Oblg$$
$$\Omega \in Oblg = \mathcal{P}((Var \times Type) \cup Var^2 \cup Var) \cup \{\mathbf{error}\}$$

The obligations represent the constraints listed above. ($TySt$ is a stack of type environments; the stack is needed to account for scope.)

We can define $asgn_R$, exp_R, and $intDecl_R$:

$$asgn_R(n, x, e) = (\,[\,], mkOblg(x, e))$$
$$exp_R(e) = (\,[\,], mkOblg(e, \mathbf{bool}))$$
$$intDecl_R(n, x) = ([(\star, \{x \mapsto \mathbf{int}\})]\,, \{x\})$$

The $\star$ represents the bottom of the environment stack. exp_R records the type of an expression as **bool** just because exp_R is only applied to expressions that appear as conditions. *mkOblg* is defined by:

$$mkOblg(x, y) = (x, y)$$
$$mkOblg(x, e_1 \oplus e_2) = mkOblg(e_1, ltype(\oplus)) \;\sqcup\; mkOblg(e_2, rtype(\oplus)) \sqcup (x, type(\oplus))$$
$$mkOblg(x, T) = (x, T)$$
$$mkOblg(e_1 \oplus e_2, T) = \text{if } type(\oplus) = T \text{ then } mkOblg(e_1, ltype(\oplus)) \;\sqcup\; mkOblg(e_2, rtype(\oplus)) \text{ else } \mathbf{error}$$

$\oplus$ 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 now present the formal development of this analysis. 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. $\star$ denotes the initial frame of the stack.

$$\Gamma \in Data = TySt \cup \{\mathbf{error}\} \cup \{\top\}$$
$$TySt = ((Node \cup \{\star\}) \times TyEnv)^*$$
$$TyEnv = Var \twoheadrightarrow Type$$
$$Type = \{\mathbf{int}, \mathbf{bool}\}$$

$$asgn(n, x, e) = \lambda\Gamma.\,\text{if } type(x, \Gamma) = type(e, \Gamma) \text{ then } \Gamma \text{ else } \mathbf{error}$$
$$intDecl(n, x) = \lambda\Gamma.\,\text{if } type(x, \Gamma) \text{ is defined then } \mathbf{error} \text{ else } add(\Gamma, x, \mathbf{int})$$
$$exp(e) = \lambda\Gamma.\,\text{if } type(e, \Gamma) = \mathbf{bool} \text{ then } \Gamma \text{ else } \mathbf{error}$$
$$beginScope = \lambda n.\lambda\Gamma.\,(n, \emptyset) :: \Gamma$$
$$endScope = \lambda\Gamma.\,\text{let } (n, \gamma) :: \Gamma' \leftarrow \Gamma \text{ in } \Gamma'$$

Below are $beginScope_R$ and $endScope_R$. In the definition of $\mathcal{R}$ for scoped statements, there is no occurrence of $beginScope_R$ because $\mathcal{R}$ is strictly bottom-up; we never "enter" into scopes, we only "exit" from them. We still give a definition for $beginScope_R$ because it is used in $\mathcal{F}^R$.

$$beginScope_R = \lambda n.\lambda(\Gamma, \Omega).\,(beginScope(n)\Gamma,\, \Omega)$$
$$endScope_R = \lambda(\Gamma, \Omega).(endScope(\Gamma), \Omega)$$

$$\mathcal{B}[\![\mathsf{skip}]\!] = id$$

$$\mathcal{B}[\![x = e]\!] = \lambda(\eta, d).(\eta, asgn(x, e)(d))$$

$$\mathcal{B}[\![\mathsf{break}\ \ell]\!] = \lambda(\eta, d).(\eta, \eta(\ell))$$

$$\mathcal{B}[\![\ell : P]\!] = \lambda(\eta, d).\ \text{let}\ (\eta', d') \leftarrow \mathcal{B}[\![P]\!](\eta[\ell \mapsto d], d)\ \text{in}\ (\eta'[\ell \mapsto \top_{Data}], d')$$

$$\mathcal{B}[\![P_1; P_2]\!] = \mathcal{B}[\![P_2]\!]; \mathcal{B}[\![P_1]\!]$$

$$\mathcal{B}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = \lambda(\eta, d).\ \text{let}\ (\eta_1, d_1) \leftarrow \mathcal{B}[\![P_1]\!](\eta, d)\ (\eta_2, d_2) \leftarrow \mathcal{B}[\![P_2]\!](\eta, d)\ \text{in}\ (\eta, exp(e)(d_1 \wedge d_2))$$

Fig. 15. Intermediate framework for backward analysis.

We define the semicolon operation as

$$(\Gamma, \Omega);_R (\Gamma', \Omega') = (concatenate(\Gamma, \Gamma'),\ sequence(\Omega, \Omega', \Gamma))$$

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

$$sequence(\Omega_1, \Omega_2, \Gamma) = \{\omega \mid \omega \in \Omega_1 \text{ or, } \omega \in \Omega_2 \text{ and } \omega \text{ not satisfied by } \Gamma\}$$

Note that as more information becomes available with the $;_r$ operation, satisfied obligations are removed.

For meet we have

$$(\Gamma, \Omega) \wedge_R (\Gamma', \Omega') = (longestCommonSuffix(\Gamma, \Gamma'), \Omega \cup \Omega')$$

Finally, in the **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.

$$\mathbf{abs}(\Gamma_R, \Omega_R) = \lambda\Gamma.\ \text{let}\ (\Gamma', \Omega') \leftarrow (\Gamma, \emptyset);_R (\Gamma_R, \Omega_R)\ \text{in if}\ \Omega' = \emptyset\ \text{then}\ \Gamma'\ \text{else}\ \mathbf{error}$$

THEOREM 4.6. *R for* TC *is an exact representation.*

PROOF. This proof follows the same structure as the corresponding proof for CP, and is omitted. □

5. BACKWARD ANALYSIS FRAMEWORK

We can define a similar framework for backwards analysis, although break statements significantly complicate matters. We directly start with the intermediate framework here. It is presented in Figure 15, and the representation is in Figure 16. The abstraction function is

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

THEOREM 5.1. *For a legal program P, if the DFFun functions are distributive (i.e. $f(d \wedge d') = f(d) \wedge f(d')$), then $\mathbf{abs}_E(\mathcal{R}[\![P]\!]) = \mathcal{B}[\![P]\!]$.*

PROOF. The proof is by induction on the structure of P. Details are provided in the Appendix. □

$$\mathcal{R}[\![\mathsf{skip}]\!] = (\top_{Env_R},\ id_R)$$

$$\mathcal{R}[\![x = e]\!] = (\top_{Env_R},\ asgn_R(x, e))$$

$$\mathcal{R}[\![\mathsf{break}\ \ell]\!] = (\top_{Env_R}[\ell \mapsto id_R],\ \top_R)$$

$$\mathcal{R}[\![\ell : P]\!] = \text{let } (\eta, r) \leftarrow \mathcal{R}[\![P]\!] \text{ in } (\eta[\ell \mapsto \top_R],\ r \wedge_R \eta(\ell))$$

$$\mathcal{R}[\![P_1; P_2]\!] = \text{let } (\eta_1, r_1) \leftarrow \mathcal{R}[\![P_1]\!],\ (\eta_2, r_2) \leftarrow \mathcal{R}[\![P_2]\!] \text{ in } (\eta_1 \wedge_R (\eta_2 \ ;_R\ r_1),\ r_2 \ ;_R\ r_1)$$

$$\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = (\mathcal{R}[\![P_1]\!] \wedge_R \mathcal{R}[\![P_2]\!]) \ ;_R\ exp_R(e)$$

Fig. 16. Representation for framework of Figure 15.

$$\mathcal{B}[\![n : \mathsf{skip}]\!] = id$$

$$\mathcal{B}[\![n : x = e]\!] = \lambda(\varphi, \eta, d).(\varphi[n \mapsto asgn(x, e)(d)], \eta, asgn(x, e)(d))$$

$$\mathcal{B}[\![n : \mathsf{break}\ \ell]\!] = \lambda(\varphi, \eta, d).(\varphi[n \mapsto \eta(\ell)], \eta, \eta(\ell))$$

$$\mathcal{B}[\![n : (\ell : n_1 : P)]\!] = \lambda(\varphi, \eta, d).\ \text{let } (\varphi', \eta', d') \leftarrow \mathcal{B}[\![n_1 : P]\!](\varphi, \eta[\ell \mapsto d], d) \text{ in } (\varphi'[n \mapsto d'], \eta'[\ell \mapsto \top_{Data}], d')$$

$$\mathcal{B}[\![n : (n_1 : P_1; n_2 : P_2)]\!] = \lambda(\varphi, \eta, d).\ \text{let } (\varphi', \eta', d') \leftarrow (\mathcal{B}[\![n_2 : P_2]\!]; \mathcal{B}[\![n_1 : P_1]\!])(\varphi, \eta, d) \text{ in } (\varphi'[n \mapsto d'], \eta', d')$$

$$\mathcal{B}[\![n : \mathsf{if}(e)\ n_1 : P_1\ \mathsf{else}\ n_2 : P_2]\!] = \lambda(\varphi, \eta, d).\ \text{let } (\varphi_1, \eta_1, d_1) \leftarrow \mathcal{B}[\![n_1 : P_1]\!](\varphi, \eta, d),\ (\varphi_2, \eta_2, d_2) \leftarrow \mathcal{B}[\![n_2 : P_2]\!](\varphi, \eta, d) \text{ in } ((\varphi_1 \cup \varphi_2)[n \mapsto exp(e)(d_1 \wedge d_2)], \eta, exp(e)(d_1 \wedge d_2))$$

Fig. 17. Full framework for backward analysis.

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 10). 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 $\mathcal{R}[\![P_1; P_2]\!]$ in the forward representation function of Figure 11 updates the node-map for each node in P_1 and P_2, $\mathcal{R}[\![L : P]\!]$ and $\mathcal{R}[\![P_1; P_2]\!]$ in the full backward representation function update each mapping in their node-maps as well. Full versions of $\mathcal{B}$ and $\mathcal{R}$ are given in Figures 17 and 18, respectively. In Figure 18, *closeLabel* is defined as

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

The **abs** function for the full backward framework is defined as

$$\begin{aligned}
&\mathbf{abs}_F(\varphi, \eta, r) = \\
&\quad \lambda(\varphi', \eta', d').\ \text{let } \varphi'' \leftarrow \lambda n.\ \text{let } (\bar{\eta}, \bar{r}) \leftarrow \varphi(n) \\
&\qquad\qquad \text{in } \mathbf{abs}(\bar{r})(d') \wedge \textstyle\bigwedge_{\ell \in \text{Label}} \mathbf{abs}(\bar{\eta}(\ell))(\eta'(\ell)) \\
&\quad \text{in } (\varphi' \cup \varphi'',\ \eta',\ \mathbf{abs}(r)(d') \wedge \textstyle\bigwedge_{\ell \in \text{Label}} \mathbf{abs}(\eta(\ell))(\eta'(\ell)))
\end{aligned}$$

Theorem 5.2. *For a legal program P, if the DFFun functions are distributive (i.e. $f(d \wedge d') = f(d) \wedge f(d')$), then $\mathbf{abs}_F(\mathcal{R}[\![P]\!]) = \mathcal{B}[\![P]\!]$.*

$$\mathcal{R}[\![n : \mathsf{skip}]\!] = (\{n \mapsto (\top_{Env_R}, id_R)\},\ \top_{Env_R},\ id_R)$$

$$\mathcal{R}[\![n : x = e]\!] = (\{n \mapsto (\top_{Env_R}, asgn_R(x,e))\},\ \top_{Env_R},\ asgn_R(x,e))$$

$$\mathcal{R}[\![n : \mathsf{break}\ \ell]\!] = (\{n \mapsto (\top_{Env_R}[\ell \mapsto id_R], \top_R)\},\ \top_{Env_R}[\ell \mapsto id_R],\ \top_R)$$

$$\begin{aligned}\mathcal{R}[\![n : (\ell : n_1 : P)]\!] = \text{let } &(\varphi, \eta, r) \leftarrow \mathcal{R}[\![n_1 : P]\!] \\ \text{in } &(closeLabel(\ell,\ \varphi[n \mapsto (\eta, r)]),\ \eta[\ell \mapsto \top_R],\ r \wedge_R \eta(\ell))\end{aligned}$$

$$\begin{aligned}\mathcal{R}[\![n : (n_1 : P_1; n_2 : P_2)]\!] = \text{let } &(\varphi_1, \eta_1, r_1) \leftarrow \mathcal{R}[\![n_1 : P_1]\!],\ (\varphi_2, \eta_2, r_2) \leftarrow \mathcal{R}[\![n_2 : P_2]\!] \\ \text{in } &(\lambda n'.\ \text{if } \varphi_2(n') \text{ defined then } \varphi_2(n') \\ &\quad \text{if } \varphi_1(n') \text{ defined then let } (\eta', r') \leftarrow \varphi_1(n') \\ &\qquad \text{in } (\eta' \wedge_R (\eta_2 ;_R r'),\ r_2 ;_R r') \\ &\quad \text{if } n' = n \text{ then } (\eta_1 \wedge_R (\eta_2 ;_R r_1),\ r_2 ;_R r_1), \\ &\quad \eta_1 \wedge_R (\eta_2 ;_R r_1),\ r_2 ;_R r_1)\end{aligned}$$

$$\begin{aligned}\mathcal{R}[\![n : \mathsf{if}(e)\ n_1 : P_1\ \mathsf{else}\ n_2 : P_2]\!] = \text{let } &(\varphi_1, \eta_1, r_1) \leftarrow \mathcal{R}[\![n_1 : P_1]\!],\ (\varphi_2, \eta_2, r_2) \leftarrow \mathcal{R}[\![n_2 : P_2]\!] \\ \text{in } &((\varphi_1 \cup \varphi_2)[n \mapsto (r_1 \wedge_R r_2);_R exp_R(e)], \\ &(\eta_1 \wedge_R \eta_2);_R exp_R(e), \\ &(r_1 \wedge_R r_2);_R exp_R(e))\end{aligned}$$

Fig. 18. Representation for framework of Figure 17.

Proof. The proof is similar to the proof for the intermediate framework (Theorem 5.1). □

5.1 Live Variables (LV)

Data is defined as

$$L \in Data = (\mathcal{P}(\mathrm{Var})) \cup \{\top\}$$

and is ordered by reverse set inclusion.

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

$$\begin{aligned}&R = \mathcal{P}(\mathrm{Var})^2 \\ &asgn_R(n,x,e) = (\{x\}, vars(e)) \\ &exp_R(e) = (\emptyset, vars(e))\end{aligned}$$

Definitions of id_R, $;_R$, $\wedge_R$ and **abs** are the same as in RD (Section 4.1).
Note that LV is a distributive analysis.

5.2 Very Busy Expressions (VBE)

The definitions, except the following, are the same as in AE.

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

Note that VBE is a distributive analysis.

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*.

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

	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 ([Kamin 2004])	1.10	1.07	1.31	1.24	0.93	1.17	1.64	1.26	1.05
Fib2 ([Kamin 2004])	1.23	1.16	0.67	1.48	0.99	1.18	2.02	1.47	1.05
Sort ([Kamin et al. 2000])	1.48	1.21	1.92	1.64	1.08	1.59	1.86	1.29	1.66
Huffman ([Kamin 2004])	1.11	1.29	0.30	1.04	0.93	1.02	1.31	1.30	0.95
Marshalling 1 ([Aktemur et al. 2005])	12.37	3.93	28.27	34.83	15.42	9.34	49.64	18.92	12.04
Marshalling 2 ([Aktemur et al. 2005])	2.01	1.75	16.01	1.83	1.33	1.86	2.59	2.27	1.47

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 I, 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. Benchmarking was done on a Linux machine with 1.5 GHz CPU and 1GB memory.

We briefly describe the benchmarks used in Table I.

—**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 [Kamin 2004].

—**Sort** is a generator that produces a sort function by inlining the comparison operation [Kamin et al. 2000].

—**Huffman** is a generator that turns a Huffman tree into a sequence of conditional statements [Kamin 2004].

—**Marshalling 1** is part of a program that produces customized serializers in Java [Aktemur et al. 2005]; 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. STAGING AT THE IR LEVEL

In the type of program generation we are considering — what we have called *fragment-oriented program generation* in [Aktemur and Kamin] — fragments are written in source code. Approaches vary in, among other ways, whether they allow fragments to be translated to intermediate code. In Jumbo [Kamin et al. 2003], for example, it is not generally possible to do this; because fragments are nearly arbitrary, a fragment could introduce a declaration that would affect the translation of all subsequent code. In most other systems [Poletto et al. 1997; Smith et al. 2003; Oiwa et al. 2001; Taha et al.], restrictions on fragments ensure that the context of any fragment is sufficiently known at initial compile time to permit generation of intermediate code. Furthermore, some of the analyses we have presented in staged form are the types of analyses normally performed on the intermediate representation rather than the source.

The question then arises as to whether our approach to staging is applicable in cases when fragments are presented in IR form. One could adopt a control-flow graph (CFG) approach, as in [Katelman 2006], in which CFG's with multiple in- and out-edges are spliced together. However, here we will present a simpler approach, in which we use a subset of our source language as the IR. (This amounts to the same thing as the CFG approach, where the labelled statements and breaks correspond to the labelling of in- and out-edges of the CFG fragment.) Specifically, we provide a translation from our source language to a subset of the source language which can reasonably be considered as an intermediate representation. The most interesting part of the translation is the short-circuit evaluation of boolean expressions.

The IR subset of our source language has these restrictions:

—All expressions are limited to at most one operator.

—All while statements have the form $\mathsf{while(true)\,do}\ \{...\}$. We will, in fact, write this as $\mathsf{loop}\,\{...\}$.

—All if statements have the form $\mathsf{if}(\mathit{variable})\ \mathsf{break}\ L;\ \mathsf{else\ break}\ L';$.

Thus, there is no unstructured goto — loops provide the only backward branches — but the code is otherwise flattened. We would argue that this is a reasonable candidate for an IR in the sense that it is just as amenable to translation to target machine code as an ordinary 3-address, CFG-based IR. It cannot express arbitrary CFG's, but it is not clear whether that is a disadvantage; it corresponds strictly to reducible CFG's [Aho et al. 1986], which is a large class often thought to contain all the useful CFG's. We are not prepared to make any stronger arguments about the utility of this IR. We simply say that it is not *prima facie* unsuitable as an IR.

We now present the translation of source code to this IR. The idea of short-circuit compilation of boolean expressions is explained in numerous compiler textbooks

[Aho et al. 1986; Cooper and Torczon 2004]. Our translation consists of three schemes:

$[\![S]\!]$: translation of statement S; returns a program.

$[\![e]\!]$: translation of non-boolean expression e; returns a program and a variable name, the latter giving the location of the value of the expression.

$[\![e]\!]_{L_t,L_f}$: translation of a boolean expression, in a context in which either break L_t or break L_f should be executed, depending upon the value of e.

Statements:[3]

$$
\begin{aligned}
[\![x = e]\!] &= \text{let } (C, v) \leftarrow [\![e]\!] \\
&\quad \text{in } C; x = v;
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathsf{while}(e)\ \mathsf{do}\ S]\!] &= L : \mathsf{loop}\,\{ \\
&\qquad\qquad L' : \ \{\ [\![e]\!]_{L',L}\ \} \\
&\qquad\qquad [\![S]\!] \\
&\qquad \}
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathsf{if}(e)\ S\ \mathsf{else}\ S']\!] = L_1 : \ \{ & \\
& L_2 : \{ \\
& \qquad L_3 : \ \{\ [\![e]\!]_{L_2,L_3}\ \} \\
& \qquad [\![S']\!] \\
& \qquad \mathsf{break}\ L_1 \\
& \} \\
& [\![S]\!] \\
\}
\end{aligned}
$$

where L, L', L_1, L_2, and L_3 are fresh labels.

Non-boolean expressions:

$$
\begin{aligned}
[\![n]\!] &= (t = n\,,\ t) \\
[\![x]\!] &= (\epsilon\,,\ x) \\
[\![e_1 \oplus e_2]\!] &= \text{let } (C_1, t_1) \leftarrow [\![e_1]\!] \\
&\qquad\quad (C_2, t_2) \leftarrow [\![e_2]\!] \\
&\quad \text{in } (C_1; C_2; t = t_1 \oplus t_2\,,\ t)
\end{aligned}
$$

where t in the first and third rules is a fresh variable.

Boolean expressions:

$$
\begin{aligned}
[\![\mathsf{true}]\!]_{L_t,L_f} &= \mathsf{break}\ L_t; \\
[\![\mathsf{false}]\!]_{L_t,L_f} &= \mathsf{break}\ L_f;
\end{aligned}
$$

[3]Various simple optimizations can be applied to avoid things like translating `x = y;` to `temp = y; x = temp;`, or generating statements of the form L : break L;. For our current purposes, these would only complicate matters without altering the basic point we are making.

$$
\begin{aligned}
[\![e_1 \oslash e_2]\!]_{L_t,L_f} = \ & \text{let } (C_1, t_1) \leftarrow [\![e_1]\!] \\
& \quad\ (C_2, t_2) \leftarrow [\![e_2]\!] \\
& \text{in } C_1; \\
& \quad\ C_2; \\
& \quad\ t = t_1 \oslash t_2; \\
& \quad\ \mathsf{if}(t) \text{ break } L_t; \text{ else break } L_f;
\end{aligned}
$$

$$
[\![!e]\!]_{L_t,L_f} = [\![e]\!]_{L_f,L_t}
$$

$$
\begin{aligned}
[\![e_1 \ || \ e_2]\!]_{L_t,L_f} = \ & L : \{[\![e_1]\!]_{L_t,L}\} \\
& [\![e_2]\!]_{L_t,L_f}
\end{aligned}
$$

$$
\begin{aligned}
[\![e_1 \&\& e_2]\!]_{L_t,L_f} = \ & L : \{[\![e_1]\!]_{L,L_f}\} \\
& [\![e_2]\!]_{L_t,L_f}
\end{aligned}
$$

8. CONCLUSIONS

We have presented a framework for static analysis of ASTs, including break statements, that allows the analysis 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 presented representations for several data-flow analyses, namely reaching definitions, available expressions, constant propagation, loop invariance, live variables, very busy expressions, and also type checking. We provided experimental results to demonstrate that staging can achieve significant runtime performance improvement. We also presented a translation from source code to intermediate code that shows our methodology for staging can be applied on IR-level code as well.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers of GPCE '06, where an earlier version of this work appeared, for their helpful comments.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

Aktemur, B., Jones, J., Kamin, S., and Clausen, L. 2005. Optimizing marshalling by run-time program generation. In *GPCE '05: Proceedings of the 4th international conference on Generative programming and component engineering*, R. Glück and M. R. Lowry, Eds. Lecture Notes in Computer Science, vol. 3676. Springer, 221–236.

Aktemur, B. and Kamin, S. Writing specialized libraries. Under submission. Available at `http://loome.cs.uiuc.edu/pubs.html`.

Chambers, C. 2002. Staged compilation. In *PEPM '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*. ACM, New York, NY, USA, 1–8.

Cooper, K. D. and Torczon, L. 2004. *Engineering a Compiler*. Morgan Kaufmann.

Czarnecki, K., O'Donnell, J. T., Striegnitz, J., and Taha, W. 2004. Dsl implementation in metaocaml, template haskell, and c++. In *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle*. Lecture Notes in Computer Science, vol. 3016. 51–72.

Ireland, A. and Bundy, A. 1999. Automatic verification of functions with accumulating parameters. *J. Funct. Program. 9,* 2, 225–245.

Kamin, S. 2004. Program generation considered easy. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM Press, 68–79.

Kamin, S., Aktemur, B., and Katelman, M. 2006. Staging static analyses for program generation. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*. ACM, New York, NY, USA, 1–10.

Kamin, S., Callahan, M., and Clausen, L. 2000. Lightweight and generative components-1: Source-level components. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*. Springer-Verlag, 49–64.

Kamin, S., Clausen, L., and Jarvis, A. 2003. Jumbo: run-time code generation for java and its applications. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 48–56.

Katelman, M. 2006. Staged static analyses and run-time program generation. M.S. thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA.

Kramer, R., Gupta, R., and Soffa, M. L. 1994. The combining dag: A technique for parallel data flow analysis. *IEEE Trans. Parallel Distrib. Syst. 5,* 8, 805–813.

Marlowe, T. J. and Ryder, B. G. 1990. An efficient hybrid algorithm for incremental data flow analysis. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 184–196.

Oiwa, Y., Masuhara, H., and Yonezawa, A. 2001. Dynjava: Type safe dynamic code generation in java. In *The 3rd JSSST Workshop on Programming and Programming Languages (PPL2001)*.

Poletto, M., Engler, D. R., and Kaashoek, M. F. 1997. tcc: a system for fast, flexible, and high-level dynamic code generation. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*. ACM Press, 109–121.

Reps, T., Horwitz, S., and Sagiv, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 49–61.

Rountev, A., Kagan, S., and Marlowe, T. 2006. Interprocedural dataflow analysis in the presence of large libraries. In *International Conference on Compiler Construction*. LNCS 3923. 2–16.

Sagiv, M., Reps, T., and Horwitz, S. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science 167,* 1-2, 131–170.

Sharir, M. and Pnueli, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Englewood Cliffs, NJ, 189–233.

Smith, F., Grossman, D., Morrisett, G., Hornof, L., and Jim, T. 2003. Compiling for template-based run-time code generation. *Journal of Functional Programming 13,* 3, 677–708.

Taha, W., Calcagno, C., Leroy, X., and Pizzi, E. Metaocaml. `http://www.metaocaml.org/`.

A. PROOFS OF THEOREMS

PROOF OF THEOREM 3.4. The proof is by induction on the structure of P.

Case 1. $P = \mathsf{skip}$

$$\begin{aligned}
\mathbf{abs}(\mathcal{F}^R[\![\mathsf{skip}]\!]r) &= \mathbf{abs}(id^R(r)) \\
&= \mathbf{abs}(r) \\
&= \mathbf{abs}(r)\ ;\ id \\
&= \mathbf{abs}(r)\ ;\ \mathbf{abs}(id_R) \\
&= \mathbf{abs}(r)\ ;\ \mathbf{abs}(\mathcal{R}[\![\mathsf{skip}]\!]) \\
&= \mathbf{abs}(r\ ;_R \mathcal{R}[\![\mathsf{skip}]\!])
\end{aligned}$$

Case 2. $P = x = e$

$$\begin{aligned}
\mathbf{abs}(\mathcal{F}^R[\![x = e]\!]r) &= \mathbf{abs}(asgn^R(x,e)(r)) \\
&= \mathbf{abs}(r\ ;_R asgn_R(x,e)) \\
&= \mathbf{abs}(r\ ;_R \mathcal{R}[\![x = e]\!])
\end{aligned}$$

Case 3. $P = P_1; P_2$

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

$$\begin{aligned}
\mathbf{abs}(\mathcal{F}^R[\![P_1]\!]r) &= \mathbf{abs}(r\ ;_R \mathcal{R}[\![P_1]\!]) \\
\mathbf{abs}(\mathcal{F}^R[\![P_2]\!]r) &= \mathbf{abs}(r\ ;_R \mathcal{R}[\![P_2]\!])
\end{aligned}$$

Now we work on $P_1; P_2$:

$$\begin{aligned}
\mathbf{abs}(\mathcal{F}^R[\![P_1; P_2]\!]r) &= \mathbf{abs}(\mathcal{F}^R[\![P_2]\!](\mathcal{F}^R[\![P_1]\!]r)) \\
&= \mathbf{abs}((\mathcal{F}^R[\![P_1]\!]r)\ ;_R \mathcal{R}[\![P_2]\!]) && (1) \\
&= \mathbf{abs}((\mathcal{F}^R[\![P_1]\!]r)); \mathbf{abs}(\mathcal{R}[\![P_2]\!]) \\
&= \mathbf{abs}(r\ ;_R \mathcal{R}[\![P_1]\!]); \mathbf{abs}(\mathcal{R}[\![P_2]\!]) && (2) \\
&= \mathbf{abs}(r); \mathbf{abs}(\mathcal{R}[\![P_1]\!]); \mathbf{abs}(\mathcal{R}[\![P_2]\!]) \\
&= \mathbf{abs}(r); \mathbf{abs}(\mathcal{R}[\![P_1]\!]\ ;_R \mathcal{R}[\![P_2]\!]) \\
&= \mathbf{abs}(r); \mathbf{abs}(\mathcal{R}[\![P_1; P_2]\!]) \\
&= \mathbf{abs}(r\ ;_R \mathcal{R}[\![P_1; P_2]\!])
\end{aligned}$$

Induction hypothesis is used to derive (1) and (2).

Case 4. $P = \mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2$

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

$$\begin{aligned}
\mathbf{abs}(\mathcal{F}^R[\![P_1]\!]r) &= \mathbf{abs}(r\ ;_R \mathcal{R}[\![P_1]\!]) \\
\mathbf{abs}(\mathcal{F}^R[\![P_2]\!]r) &= \mathbf{abs}(r\ ;_R \mathcal{R}[\![P_2]\!])
\end{aligned}$$

Now we work on $\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2$:

$$\begin{aligned}
&\mathbf{abs}(\mathcal{F}^R[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!]r) = \mathbf{abs}((exp^R(e)\ ;\ (\mathcal{F}^R[\![P_1]\!] \wedge^R \mathcal{F}^R[\![P_2]\!]))r) \\
&= \mathbf{abs}((\mathcal{F}^R[\![P_1]\!] \wedge^R \mathcal{F}^R[\![P_2]\!])(r\ ;_R exp_R(e))) \\
&= \mathbf{abs}(\mathcal{F}^R[\![P_1]\!](r\ ;_R exp_R(e)) \wedge_R \mathcal{F}^R[\![P_2]\!](r\ ;_R exp_R(e)))
\end{aligned}$$

$$
\begin{aligned}
&= \mathbf{abs}(\mathcal{F}^R[\![P_1]\!](r ;_R exp_R(e))) \wedge \mathbf{abs}(\mathcal{F}^R[\![P_2]\!](r ;_R exp_R(e))) \\
&= \mathbf{abs}((r ;_R exp_R(e)) ;_R \mathcal{R}[\![P_1]\!]) \wedge \mathbf{abs}((r ;_R exp_R(e)) ;_R \mathcal{R}[\![P_2]\!]) \qquad (3) \\
&= \mathbf{abs}(r ;_R exp_R(e)); \mathbf{abs}(\mathcal{R}[\![P_1]\!]) \ \wedge \ \mathbf{abs}(r ;_R exp_R(e)); \mathbf{abs}(\mathcal{R}[\![P_2]\!]) \\
&= \mathbf{abs}(r); \mathbf{abs}(exp_R(e)); \mathbf{abs}(\mathcal{R}[\![P_1]\!]) \ \wedge \ \mathbf{abs}(r); \mathbf{abs}(exp_R(e)); \mathbf{abs}(\mathcal{R}[\![P_2]\!]) \\
&= \mathbf{abs}(r); \mathbf{abs}(exp_R(e)); (\mathbf{abs}(\mathcal{R}[\![P_1]\!]) \ \wedge \ \mathbf{abs}(\mathcal{R}[\![P_2]\!])) \\
&= \mathbf{abs}(r); \mathbf{abs}(exp_R(e)); (\mathbf{abs}(\mathcal{R}[\![P_1]\!] \wedge_R \mathcal{R}[\![P_2]\!])) \\
&= \mathbf{abs}(r); \mathbf{abs}(exp_R(e) ;_R (\mathcal{R}[\![P_1]\!] \wedge_R \mathcal{R}[\![P_2]\!])) \\
&= \mathbf{abs}(r); \mathbf{abs}(\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!]) \\
&= \mathbf{abs}(r ;_R (\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!]))
\end{aligned}
$$

Induction hypothesis is used to derive (3).

□

Proof of Theorem 3.6. The proof is by induction on the structure of P.

Case 1. $P = \mathsf{skip}$

$$
\begin{aligned}
\mathbf{abs}_E(\mathcal{R}[\![\mathsf{skip}]\!]) &= \mathbf{abs}_E((\top_{Env_R},\ id_R)) \\
&= \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \wedge \mathbf{abs}(\top_{Env_R}(\ell))d',\ \mathbf{abs}(id_R)d') \\
&= \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \wedge \mathbf{abs}(\top_R)d',\ id(d')) \\
&= \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \wedge (\lambda d.\top_{Data})d',\ d') \\
&= \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \wedge \top_{Data},\ d') \\
&= \lambda(\eta', d').(\lambda\ell.\eta'(\ell),\ d') \\
&= \lambda(\eta', d').(\eta',\ d') \\
&= \mathcal{F}[\![\mathsf{skip}]\!]
\end{aligned}
$$

Case 2. $P = x = e$

$$
\begin{aligned}
\mathbf{abs}_E(\mathcal{R}[\![x = e]\!]) &= \mathbf{abs}_E((\top_{Env_R},\ asgn_R(x, e))) \\
&= \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \wedge \mathbf{abs}(\top_{Env_R}(\ell))d',\ \mathbf{abs}(asgn_R(x, e))d') \\
&= \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \wedge \mathbf{abs}(\top_R)d',\ asgn(x, e)(d')) \\
&= \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \wedge (\lambda d.\top_{Data})d',\ asgn(x, e)(d')) \\
&= \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \wedge \top_{Data},\ asgn(x, e)(d')) \\
&= \lambda(\eta', d').(\lambda\ell.\eta'(\ell),\ asgn(x, e)(d')) \\
&= \lambda(\eta', d').(\eta',\ asgn(x, e)(d')) \\
&= \mathcal{F}[\![x = e]\!]
\end{aligned}
$$

Case 3. $P = \mathsf{break}\ \ell$

$$
\begin{aligned}
\mathbf{abs}_E(\mathcal{R}[\![\mathsf{break}\ \ell]\!]) &= \mathbf{abs}_E((\top_{Env_R}[\ell \mapsto id_R], \top_R)) \\
&= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\top_{Env_R}[\ell \mapsto id_R](\ell'))d',\ \mathbf{abs}(\top_R)d') \\
&= \lambda(\eta', d').(\Big(\lambda\ell'.\begin{cases} \eta'(\ell) \wedge \mathbf{abs}(id_R)d' & \text{if } \ell = \ell' \\ \eta'(\ell') \wedge \mathbf{abs}(\top_R)d' & \text{if } \ell \neq \ell' \end{cases}\Big), \top_{Data})
\end{aligned}
$$

$$
\begin{aligned}
&= \lambda(\eta', d').(\Big(\lambda\ell'.\begin{cases}\eta'(\ell) \wedge d' & \text{if } \ell = \ell' \\ \eta'(\ell') & \text{if } \ell \neq \ell'\end{cases}\Big), \top_{Data}) \\
&= \lambda(\eta', d').(\eta'[\ell \mapsto \eta'(\ell) \wedge d'],\ \top_{Data}) \\
&= \mathcal{F}[\![\mathsf{break}\ \ell]\!]
\end{aligned}
$$

Case 4. $P = \ell : P'$

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

$$
\begin{aligned}
\mathcal{F}[\![P']\!] &= \mathbf{abs}_E(\mathcal{R}[\![P']\!]) \\
&= \mathbf{abs}_E((\eta, r)) \\
&= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta(\ell'))d',\ \mathbf{abs}(r)d')
\end{aligned}
$$

Now we work on $\mathbf{abs}_E(\mathcal{R}[\![\ell : P']\!])$. Note that because we require all the programs to be legal, the incoming environment has ℓ mapped to $\top_{Data}$.

$$
\begin{aligned}
&\mathbf{abs}_E(\mathcal{R}[\![\ell : P']\!]) = \mathbf{abs}_E((\eta[\ell \mapsto \top_R], r \wedge_R \eta(\ell))) \\
&= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta[\ell \mapsto \top_R](\ell'))d',\ \mathbf{abs}(r \wedge_R \eta(\ell))d') \\
&= \lambda(\eta', d').(\lambda\ell'.\begin{cases}\eta'(\ell) \wedge \mathbf{abs}(\top_R)d' & \text{if } \ell = \ell' \\ \eta'(\ell') \wedge \mathbf{abs}(\eta(\ell'))d' & \text{if } \ell \neq \ell'\end{cases},\ \mathbf{abs}(r \wedge_R \eta(\ell))d') \\
&= \lambda(\eta', d').(\lambda\ell'.\begin{cases}\top_{Data} \wedge \top_{Data} & \text{if } \ell = \ell' \\ \eta'(\ell') \wedge \mathbf{abs}(\eta(\ell'))d' & \text{if } \ell \neq \ell'\end{cases},\ \mathbf{abs}(r \wedge_R \eta(\ell))d') \\
&= \lambda(\eta', d').(\lambda\ell'.\begin{cases}\top_{Data} & \text{if } \ell = \ell' \\ \eta'(\ell') \wedge \mathbf{abs}(\eta(\ell'))d' & \text{if } \ell \neq \ell'\end{cases},\ \mathbf{abs}(r \wedge_R \eta(\ell))d') \qquad (4)
\end{aligned}
$$

And $\mathcal{F}[\![\ell : P']\!]$:

$$
\begin{aligned}
&\mathcal{F}[\![\ell : P']\!] = \lambda(\eta', d').\ \text{let } (\eta_1, d_1) \leftarrow \mathcal{F}[\![P']\!](\eta', d') \\
&\qquad\qquad \text{in } (\eta_1[\ell \mapsto \top_{Data}],\ d_1 \wedge \eta_1(\ell)) \\
&= \lambda(\eta', d').\ \text{let } (\eta_1, d_1) \leftarrow (\lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta(\ell'))d',\ \mathbf{abs}(r)d'))(\eta', d') \\
&\qquad\qquad \text{in } (\eta_1[\ell \mapsto \top_{Data}],\ d_1 \wedge \eta_1(\ell)) \\
&= \lambda(\eta', d').\ \text{let } (\eta_1, d_1) \leftarrow (\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta(\ell'))d',\ \mathbf{abs}(r)d') \\
&\qquad\qquad \text{in } (\eta_1[\ell \mapsto \top_{Data}],\ d_1 \wedge \eta_1(\ell)) \\
&= \lambda(\eta', d').((\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta(\ell'))d')[\ell \mapsto \top_{Data}], \\
&\qquad\qquad \mathbf{abs}(r)d' \wedge (\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta(\ell'))d')(\ell)) \\
&= \lambda(\eta', d').((\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta(\ell'))d')[\ell \mapsto \top_{Data}],\ \mathbf{abs}(r)d' \wedge \eta'(\ell) \wedge \mathbf{abs}(\eta(\ell))d') \\
&= \lambda(\eta', d').(\lambda\ell'.\begin{cases}\top_{Data} & \text{if } \ell = \ell' \\ \eta'(\ell') \wedge \mathbf{abs}(\eta(\ell'))d' & \text{if } \ell \neq \ell'\end{cases},\ \mathbf{abs}(r)d' \wedge \top_{Data} \wedge \mathbf{abs}(\eta(\ell))d') \\
&= \lambda(\eta', d').(\lambda\ell'.\begin{cases}\top_{Data} & \text{if } \ell = \ell' \\ \eta'(\ell') \wedge \mathbf{abs}(\eta(\ell'))d' & \text{if } \ell \neq \ell'\end{cases},\ \mathbf{abs}(r \wedge_R \eta(\ell))d') \\
&= (4)
\end{aligned}
$$

Case 5. $P = P_1; P_2$

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

$$
\begin{aligned}
\mathcal{F}[\![P_1]\!] &= \mathbf{abs}_E(\mathcal{R}[\![P_1]\!]) \\
&= \mathbf{abs}_E((\eta_1, r_1))
\end{aligned}
$$

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

$$\begin{aligned}\mathcal{F}[\![P_2]\!] &= \mathbf{abs}_E(\mathcal{R}[\![P_2]\!]) \\ &= \mathbf{abs}_E((\eta_2, r_2)) \\ &= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_2(\ell'))d',\ \mathbf{abs}(r_2)d')\end{aligned}$$

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

$$\begin{aligned}&\mathbf{abs}_E(\mathcal{R}[\![P_1; P_2]\!]) = \mathbf{abs}_E((\eta_1 \wedge_R (r_1 ;_R \eta_2), r_1 ;_R r_2)) \\ &= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}((\eta_1 \wedge_R (r_1 ;_R \eta_2))(\ell'))d',\ \mathbf{abs}(r_1 ;_R r_2)d') \quad (5)\end{aligned}$$

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

$$\begin{aligned}&\mathcal{F}[\![P_1; P_2]\!] = \lambda(\eta', d').(\mathcal{F}[\![P_1]\!]; \mathcal{F}[\![P_2]\!])(\eta', d') \\ &= \lambda(\eta', d').\mathcal{F}[\![P_2]\!](\mathcal{F}[\![P_1]\!](\eta', d')) \\ &= \lambda(\eta', d').\mathcal{F}[\![P_2]\!]((\lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))d',\ \mathbf{abs}(r_1)d'))(\eta', d')) \\ &= \lambda(\eta', d').\mathcal{F}[\![P_2]\!](\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))d',\ \mathbf{abs}(r_1)d') \\ &= \lambda(\eta', d').(\lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_2(\ell'))d',\ \mathbf{abs}(r_2)d'))(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))d',\ \mathbf{abs}(r_1)d') \\ &= \lambda(\eta', d').(\lambda\ell'.(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))(d'))(\ell') \wedge \mathbf{abs}(\eta_2(\ell'))(\mathbf{abs}(r_1)d'),\ \mathbf{abs}(r_2)(\mathbf{abs}(r_1)d')) \\ &= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))(d') \wedge \mathbf{abs}(\eta_2(\ell'))(\mathbf{abs}(r_1)d'),\ \mathbf{abs}(r_1 ;_R r_2)d') \\ &= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}((\eta_1 \wedge_R (r_1 ;_R \eta_2))(\ell'))d',\ \mathbf{abs}(r_1 ;_R r_2)d') \\ &= (5)\end{aligned}$$

Case 6. $P = \mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2$

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

$$\begin{aligned}\mathcal{F}[\![P_1]\!] &= \mathbf{abs}_E(\mathcal{R}[\![P_1]\!]) \\ &= \mathbf{abs}_E((\eta_1, r_1)) \\ &= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))d',\ \mathbf{abs}(r_1)d')\end{aligned}$$

$$\begin{aligned}\mathcal{F}[\![P_2]\!] &= \mathbf{abs}_E(\mathcal{R}[\![P_2]\!]) \\ &= \mathbf{abs}_E((\eta_2, r_2)) \\ &= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_2(\ell'))d',\ \mathbf{abs}(r_2)d')\end{aligned}$$

Now we work on $\mathbf{abs}_E(\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!])$.

$$\begin{aligned}&\mathbf{abs}_E(\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!]) = \mathbf{abs}_E(exp_R(e) ;_R ((\eta_1, r_1) \wedge_R (\eta_2, r_2))) \\ &= \mathbf{abs}_E(exp_R(e) ;_R ((\eta_1, r_1) \wedge_R (\eta_2, r_2))) \\ &= \mathbf{abs}_E((exp_R(e) ;_R (\eta_1 \wedge_R \eta_2), exp_R(e) ;_R (r_1 \wedge_R r_2))) \\ &= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}((exp_R(e) ;_R (\eta_1 \wedge_R \eta_2))(\ell'))d', \\ &\qquad \mathbf{abs}(exp_R(e) ;_R (r_1 \wedge_R r_2))d') \\ &= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(exp_R(e) ;_R (\eta_1(\ell') \wedge_R \eta_2(\ell')))d', \\ &\qquad \mathbf{abs}(exp_R(e) ;_R (r_1 \wedge_R r_2))d') \quad (6)\end{aligned}$$

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

$$\mathcal{F}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = \lambda(\eta', d').\ \mathsf{let}\ (\eta'_1, d'_1) \leftarrow \mathcal{F}[\![P_1]\!](\eta', exp(e)d')$$

$$
\begin{array}{rl}
& \qquad\qquad (\eta_2', d_2') \leftarrow \mathcal{F}[\![P_2]\!](\eta', exp(e)d') \\
& \qquad\qquad \text{in } (\eta_1', d_1') \wedge (\eta_2', d_2') \\
= & \lambda(\eta', d').\ \text{let } (\eta_1', d_1') \leftarrow (\lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))d',\ \mathbf{abs}(r_1)d'))(\eta', exp(e)d') \\
& \qquad\qquad (\eta_2', d_2') \leftarrow (\lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_2(\ell'))d',\ \mathbf{abs}(r_2)d'))(\eta', exp(e)d') \\
& \qquad\qquad \text{in } (\eta_1', d_1') \wedge (\eta_2', d_2') \\
= & \lambda(\eta', d').\ \text{let } (\eta_1', d_1') \leftarrow (\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))(exp(e)d'),\ \mathbf{abs}(r_1)(exp(e)d')) \\
& \qquad\qquad (\eta_2', d_2') \leftarrow (\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_2(\ell'))(exp(e)d'),\ \mathbf{abs}(r_2)(exp(e)d')) \\
& \qquad\qquad \text{in } (\eta_1', d_1') \wedge (\eta_2', d_2') \\
= & \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))(exp(e)d') \wedge \eta'(\ell') \wedge \mathbf{abs}(\eta_2(\ell'))(exp(e)d'), \\
& \qquad \mathbf{abs}(r_1)(exp(e)d') \wedge \mathbf{abs}(r_2)(exp(e)d')) \\
= & \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))(exp(e)d') \wedge \mathbf{abs}(\eta_2(\ell'))(exp(e)d'), \\
& \qquad \mathbf{abs}(exp_R(e);_R (r_1 \wedge_R r_2))d') \\
= & \ (6)
\end{array}
$$

□

Proof of Theorem 3.7. The proof is by induction on the structure of P.

Case 1. $P = \mathsf{skip}$

For this case, we have $(\eta, r) = (\top_{Env_R}, id_R) = \mathcal{R}[\![\mathsf{skip}]\!]$.

$$
\begin{array}{rcll}
\mathbf{abs}_E(\mathcal{F}^R[\![\mathsf{skip}]\!](\eta', r')) & = & \mathbf{abs}_E(id^R(\eta', r')) & \\
& = & \mathbf{abs}_E((\eta', r')) & \\
& = & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'', \mathbf{abs}(r')d'') & (1)
\end{array}
$$

And

$$
\begin{array}{rl}
& \mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \top_{Env_R}(\ell')), r' ;_R id_R)) \\
= & \mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \top_R), r' ;_R id_R)) \\
= & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \top_R))(\ell'))d'', \mathbf{abs}(r' ;_R id_R)d'') \\
= & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \top_R))d'', \mathbf{abs}(r')d'') \\
= & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'' \wedge \top_{Data}, \mathbf{abs}(r')d'') \\
= & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'', \mathbf{abs}(r')d'') \\
= & \ (1)
\end{array}
$$

Case 2. $P = x = e$

For this case, we have $(\eta, r) = (\top_{Env_R}, asgn_R(x, e)) = \mathcal{R}[\![x = e]\!]$.

$$
\begin{array}{rcll}
\mathbf{abs}_E(\mathcal{F}^R[\![x = e]\!](\eta', r')) & = & \mathbf{abs}_E((\eta', asgn^R(x, e)(r'))) & \\
& = & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'', \mathbf{abs}(asgn^R(x, e)(r'))d'') & \\
& = & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'', \mathbf{abs}(r' ;_R asgn_R(x, e))d'') & (2)
\end{array}
$$

And

$$
\begin{array}{rl}
& \mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \top_{Env_R}(\ell')), r' ;_R asgn_R(x, e))) \\
= & \mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \top_R), r' ;_R asgn_R(x, e)))
\end{array}
$$

$$
\begin{array}{rcl}
 & = & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (r' \;;_R \top_R))(\ell'))d'', \mathbf{abs}(r' \;;_R asgn_R(x,e))d'') \\
 & = & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' \;;_R \top_R))d'', \mathbf{abs}(r' \;;_R asgn_R(x,e))d'') \\
 & = & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'' \wedge \top_{Data}, \mathbf{abs}(r' \;;_R asgn_R(x,e))d'') \\
 & = & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'', \mathbf{abs}(r' \;;_R asgn_R(x,e))d'') \\
 & = & (2)
\end{array}
$$

Case 3. $P = \mathsf{break}\ \ell$

For this case, we have $(\eta, r) = (\top_{Env_R}[\ell \mapsto id_R], \top_R) = \mathcal{R}[\![\mathsf{break}\ \ell]\!]$.

$$
\begin{array}{l}
\mathbf{abs}_E(\mathcal{F}^R[\![\mathsf{break}\ \ell]\!](\eta', r')) = \mathbf{abs}_E((\eta'[\ell \mapsto r' \wedge_R \eta'(\ell)], \top_R)) \\
\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'[\ell \mapsto r' \wedge_R \eta'(\ell)](\ell'))d'', \mathbf{abs}(\top_R)d'') \\
\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'[\ell \mapsto r' \wedge_R \eta'(\ell)](\ell'))d'', \top_{Data}) \\
\quad = \lambda(\eta'', d'').(\Big(\lambda\ell'.\begin{cases} \eta''(\ell) \wedge \mathbf{abs}(r' \wedge_R \eta'(\ell))d'' & \text{if } \ell = \ell' \\ \eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'' & \text{if } \ell \neq \ell' \end{cases}\Big), \top_{Data}) \qquad (3)
\end{array}
$$

And

$$
\begin{array}{l}
\mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' \;;_R \top_{Env_R}[\ell \mapsto id_R](\ell')), r' \;;_R \top_R))) \\
\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (r' \;;_R \top_{Env_R}[\ell \mapsto id_R](\ell')))(\ell'))d'', \mathbf{abs}(r' \;;_R \top_R)d'') \\
\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' \;;_R \top_{Env_R}[\ell \mapsto id_R](\ell')))d'', \top_{Data}) \\
\quad = \lambda(\eta'', d'').(\Big(\lambda\ell'.\begin{cases} \eta''(\ell) \wedge \mathbf{abs}(\eta'(\ell) \wedge_R (r' \;;_R id_R))d'' & \text{if } \ell = \ell' \\ \eta''(\ell) \wedge \mathbf{abs}(\eta'(\ell) \wedge_R (r' \;;_R \top_R))d'' & \text{if } \ell \neq \ell' \end{cases}\Big), \top_{Data}) \\
\quad = \lambda(\eta'', d'').(\Big(\lambda\ell'.\begin{cases} \eta''(\ell) \wedge \mathbf{abs}(\eta'(\ell) \wedge_R r')d'' & \text{if } \ell = \ell' \\ \eta''(\ell) \wedge \mathbf{abs}(\eta'(\ell'))d'' & \text{if } \ell \neq \ell' \end{cases}\Big), \top_{Data}) \\
\quad = (3)
\end{array}
$$

Case 4. $P = \ell : P'$

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

$$
\begin{array}{l}
\mathbf{abs}_E(\mathcal{F}^R[\![P']\!](\eta', r')) \\
\quad = \mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' \;;_R \eta(\ell')), r' \;;_R r)) \\
\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (r' \;;_R \eta(\ell')))(\ell'))d'', \mathbf{abs}(r' \;;_R r)d'') \\
\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' \;;_R \eta(\ell')))d'', \mathbf{abs}(r' \;;_R r)d'') \qquad (4)
\end{array}
$$

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

$$
\begin{array}{rcl}
\mathbf{abs}_E(\mathcal{F}^R[\![P']\!](\eta', r')) & = & \mathbf{abs}_E((\eta_1, r_1)) \\
 & = & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))d'', \mathbf{abs}(r_1)d'') \qquad (5)
\end{array}
$$

Since (4) = (5), we obtain

$$\mathbf{abs}(r' \;;_R r)d'' \;=\; \mathbf{abs}(r_1)d''$$

and

$$\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' \;;_R \eta(\ell')))d'' \;=\; \eta''(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))d''$$

When $\ell' = \ell$, using the legality condition, we get

$$\eta''(\ell) \wedge \mathbf{abs}(\eta'(\ell) \wedge_R (r' \;;_R \eta(\ell)))d'' \;=\; \eta''(\ell) \wedge \mathbf{abs}(\eta_1(\ell))d''$$

$$\begin{array}{lll} \Rightarrow & \top_{Data} \wedge \mathbf{abs}(\top_R \wedge_R (r' ;_R \eta(\ell)))d'' & = \top_{Data} \wedge \mathbf{abs}(\eta_1(\ell))d'' \\ \Rightarrow & \mathbf{abs}(r' ;_R \eta(\ell))d'' & = \mathbf{abs}(\eta_1(\ell))d'' \end{array}$$

Now we work on $\ell : P'$:

$$\begin{array}{lll} & \mathbf{abs}_E(\mathcal{F}^R[\![\ell : P']\!](\eta', r')) & \\ = & \mathbf{abs}_E((\eta_1[\ell \mapsto \top_R], r_1 \wedge_R \eta_1(\ell))) & \\ = & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_1[\ell \mapsto \top_R](\ell'))d'', \mathbf{abs}(r_1 \wedge_R \eta_1(\ell))d'') & \\ = & \lambda(\eta'', d'').(\lambda\ell'. \begin{cases} \eta''(\ell) \wedge \mathbf{abs}(\top_R)d'' & \text{if } \ell = \ell' \\ \eta''(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))d'' & \text{if } \ell \neq \ell' \end{cases}, & \\ & \qquad \mathbf{abs}(r_1)d'' \wedge \mathbf{abs}(\eta_1(\ell))d'') & \\ = & \lambda(\eta'', d'').(\lambda\ell'. \begin{cases} \eta''(\ell) & \text{if } \ell = \ell' \\ \eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \eta(\ell')))d'' & \text{if } \ell \neq \ell' \end{cases}, & \\ & \qquad \mathbf{abs}(r' ;_R r)d'' \wedge \mathbf{abs}(r' ;_R \eta(\ell))d'') & \\ = & \lambda(\eta'', d'').(\lambda\ell'. \begin{cases} \eta''(\ell) & \text{if } \ell = \ell' \\ \eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \eta(\ell')))d'' & \text{if } \ell \neq \ell' \end{cases}, & \\ & \qquad \mathbf{abs}(r' ;_R (r \wedge_R \eta(\ell)))d'') \qquad (6) & \end{array}$$

And using the fact that $\mathcal{R}[\![\ell : P']\!] = (\eta[\ell \mapsto \top_R], r \wedge_R \eta(\ell))$, we have

$$\begin{array}{lll} & \mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \eta[\ell \mapsto \top_R](\ell')), r' ;_R (r \wedge_R \eta(\ell)))) & \\ = & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \eta[\ell \mapsto \top_R](\ell')))(\ell'))d'', & \\ & \qquad \mathbf{abs}(r' ;_R (r \wedge_R \eta(\ell)))d'') & \\ = & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \eta[\ell \mapsto \top_R](\ell')))d'', & \\ & \qquad \mathbf{abs}(r' ;_R (r \wedge_R \eta(\ell)))d'') & \\ = & \lambda(\eta'', d'').(\lambda\ell'. \begin{cases} \eta''(\ell) \wedge \mathbf{abs}(\eta'(\ell) \wedge_R (r' ;_R \top_R))d'' & \text{if } \ell = \ell' \\ \eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \eta(\ell')))d'' & \text{if } \ell \neq \ell' \end{cases}, & \\ & \qquad \mathbf{abs}(r' ;_R (r \wedge_R \eta(\ell)))d'') & \\ = & \lambda(\eta'', d'').(\lambda\ell'. \begin{cases} \eta''(\ell) \wedge \mathbf{abs}(\top_R \wedge_R (r' ;_R \top_R))d'' & \text{if } \ell = \ell' \\ \eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \eta(\ell')))d'' & \text{if } \ell \neq \ell' \end{cases}, & \\ & \qquad \mathbf{abs}(r' ;_R (r \wedge_R \eta(\ell)))d'') & \\ = & \lambda(\eta'', d'').(\lambda\ell'. \begin{cases} \eta''(\ell) & \text{if } \ell = \ell' \\ \eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \eta(\ell')))d'' & \text{if } \ell \neq \ell' \end{cases}, & \\ & \qquad \mathbf{abs}(r' ;_R (r \wedge_R \eta(\ell)))d'') & \\ = & (6) & \end{array}$$

Case 5. $P = P_1; P_2$

Let $(\eta_1, r_1) = \mathcal{R}[\![P_1]\!]$, $(\eta_2, r_2) = \mathcal{R}[\![P_2]\!]$, $(\eta_a, r_a) = \mathcal{F}^R[\![P_1]\!](\eta', r')$, and $(\eta_b, r_b) = \mathcal{F}^R[\![P_2]\!](\eta_a, r_a)$. By the induction hypothesis, we have

$$\begin{array}{lll} \mathbf{abs}_E(\mathcal{F}^R[\![P_1]\!](\eta', r')) & = & \mathbf{abs}_E((\eta_a, r_a)) \\ & = & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell'))d'',\ \mathbf{abs}(r_a)d'') \end{array}$$

and

$$\mathbf{abs}_E(\mathcal{F}^R[\![P_1]\!](\eta', r')) = \mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \eta_1(\ell')),\ r' ;_R r_1))$$

$$= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \eta_1(\ell')))(\ell'))d'',\ \mathbf{abs}(r' ;_R r_1)d'')$$
$$= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \eta_1(\ell')))d'',\ \mathbf{abs}(r' ;_R r_1)d'')$$

Similarly, for P_2

$$\begin{aligned}\mathbf{abs}_E(\mathcal{F}^R[\![P_2]\!](\eta_a, r_a)) &= \mathbf{abs}_E((\eta_b, r_b)) \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_b(\ell'))d'',\ \mathbf{abs}(r_b)d'')\end{aligned}$$

and

$$\mathbf{abs}_E(\mathcal{F}^R[\![P_2]\!](\eta_a, r_a)) = \mathbf{abs}_E((\lambda\ell'.\eta_a(\ell') \wedge_R (r_a ;_R \eta_2(\ell')), r_a ;_R r_2))$$
$$= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta_a(\ell') \wedge_R (r_a ;_R \eta_2(\ell')))(\ell'))d'',\ \mathbf{abs}(r_a ;_R r_2)d'')$$
$$= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell') \wedge_R (r_a ;_R \eta_2(\ell')))d'',\ \mathbf{abs}(r_a ;_R r_2)d'')$$

These give us the equalities

$$\begin{aligned}\eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell'))d'' &= \eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \eta_1(\ell')))d'' \\ \mathbf{abs}(r_a)d'' &= \mathbf{abs}(r' ;_R r_1)d'' \\ \eta''(\ell') \wedge \mathbf{abs}(\eta_b(\ell'))d'' &= \eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell') \wedge_R (r_a ;_R \eta_2(\ell')))d'' \\ \mathbf{abs}(r_b)d'' &= \mathbf{abs}(r_a ;_R r_2)d''\end{aligned}$$

Now, returning to $P_1; P_2$, we have

$$\mathbf{abs}_E(\mathcal{F}^R[\![P_1; P_2]\!](\eta', r')) = \mathbf{abs}_E(\mathcal{F}^R[\![P_2]\!](\mathcal{F}^R[\![P_1]\!](\eta', r')))$$
$$= \mathbf{abs}_E(\mathcal{F}^R[\![P_2]\!](\eta_a, r_a))$$
$$= \mathbf{abs}_E((\eta_b, r_b))$$
$$= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_b(\ell'))d'',\ \mathbf{abs}(r_b)d'')$$
$$= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell') \wedge_R (r_a ;_R \eta_2(\ell')))d'',\ \mathbf{abs}(r_a ;_R r_2)d'')$$
$$= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell'))d'' \wedge \mathbf{abs}(\eta_2(\ell'))(\mathbf{abs}(r_a)d''),\ \mathbf{abs}(r_2)(\mathbf{abs}(r_a)d''))$$
$$= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \eta_1(\ell')))d'' \wedge \mathbf{abs}(\eta_2(\ell'))(\mathbf{abs}(r' ;_R r_1)d''),$$
$$\qquad \mathbf{abs}(r_2)(\mathbf{abs}(r' ;_R r_1)d''))$$
$$= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell')) \wedge (\mathbf{abs}(r'); \mathbf{abs}(\eta_1(\ell')))d'' \wedge (\mathbf{abs}(r'); \mathbf{abs}(r_1); \mathbf{abs}(\eta_2(\ell')))d'',$$
$$\qquad (\mathbf{abs}(r'); \mathbf{abs}(r_1); \mathbf{abs}(r_2))d'') \qquad (7)$$

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

$$\mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R (\eta_1 \wedge_R (r_1 ;_R \eta_2))(\ell')), r' ;_R (r_1;_R r_2)))$$
$$= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R (\eta_1 \wedge_R (r_1 ;_R \eta_2))(\ell')))(\ell'))d'',$$
$$\qquad \mathbf{abs}(r' ;_R (r_1;_R r_2))d'')$$
$$= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R (\eta_1 \wedge_R (r_1 ;_R \eta_2))(\ell')))d'',\ \mathbf{abs}(r' ;_R (r_1;_R r_2))d'')$$
$$= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell')) \wedge (\mathbf{abs}(r'); \mathbf{abs}((\eta_1 \wedge_R (r_1 ;_R \eta_2))(\ell')))d'',$$
$$\qquad (\mathbf{abs}(r'); \mathbf{abs}(r_1); \mathbf{abs}(r_2))d'')$$
$$= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell')) \wedge (\mathbf{abs}(r'); \mathbf{abs}(\eta_1(\ell')))d'' \wedge (\mathbf{abs}(r'); \mathbf{abs}(r_1); \mathbf{abs}(\eta_2(\ell')))d'',$$
$$\qquad (\mathbf{abs}(r'); \mathbf{abs}(r_1); \mathbf{abs}(r_2))d'')$$

$= (7)$

Case 6. $P = \mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2$

Let $(\eta_1, r_1) = \mathcal{R}[\![P_1]\!]$, $(\eta_2, r_2) = \mathcal{R}[\![P_2]\!]$, $(\eta_a, r_a) = \mathcal{F}^R[\![P_1]\!](\eta', exp^R(e)r')$, and $(\eta_b, r_b) = \mathcal{F}^R[\![P_2]\!](\eta', exp^R(e)r')$. By the induction hypothesis, we have

$$\begin{aligned}\mathbf{abs}_E(\mathcal{F}^R[\![P_1]\!](\eta', exp^R(e)r')) &= \mathbf{abs}_E((\eta_a, r_a)) \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell'))d'',\ \mathbf{abs}(r_a)d'')\end{aligned}$$

and

$$\begin{aligned}&\mathbf{abs}_E(\mathcal{F}^R[\![P_1]\!](\eta', exp^R(e)r')) = \mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (exp^R(e)r' ;_R \eta_1(\ell')),\ exp^R(e)r' ;_R r_1)) \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (exp^R(e)r' ;_R \eta_1(\ell')))(\ell'))d'',\ \mathbf{abs}(exp^R(e)r' ;_R r_1)d'') \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r';_R exp_R(e);_R \eta_1(\ell')))d'',\ \mathbf{abs}(r';_R exp_R(e);_R r_1)d'')\end{aligned}$$

Similarly, for P_2

$$\begin{aligned}\mathbf{abs}_E(\mathcal{F}^R[\![P_2]\!](\eta', exp^R(e)r')) &= \mathbf{abs}_E((\eta_b, r_b)) \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_b(\ell'))d'',\ \mathbf{abs}(r_b)d'')\end{aligned}$$

and

$$\begin{aligned}&\mathbf{abs}_E(\mathcal{F}^R[\![P_2]\!](\eta', exp^R(e)r')) = \mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (exp^R(e)r' ;_R \eta_2(\ell')),\ exp^R(e)r' ;_R r_2)) \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (exp^R(e)r' ;_R \eta_2(\ell')))(\ell'))d'',\ \mathbf{abs}(exp^R(e)r' ;_R r_2)d'') \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r';_R exp_R(e);_R \eta_2(\ell')))d'',\ \mathbf{abs}(r';_R exp_R(e);_R r_2)d'')\end{aligned}$$

These give us the equalities

$$\begin{aligned}\eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell'))d'' &= \eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r';_R exp_R(e);_R \eta_1(\ell')))d'' \\ \mathbf{abs}(r_a)d'' &= \mathbf{abs}(r';_R exp_R(e);_R r_1)d'' \\ \eta''(\ell') \wedge \mathbf{abs}(\eta_b(\ell'))d'' &= \eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r';_R exp_R(e);_R \eta_2(\ell')))d'' \\ \mathbf{abs}(r_b)d'' &= \mathbf{abs}(r';_R exp_R(e);_R r_2)d''\end{aligned}$$

Now, returning to $\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2$, we have

$$\begin{aligned}&\mathbf{abs}_E(\mathcal{F}^R[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!](\eta', r')) = \mathbf{abs}_E((\eta_a \wedge_R \eta_b, r_a \wedge_R r_b)) \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell') \wedge_R \eta_b(\ell'))d'', \mathbf{abs}(r_a \wedge_R r_b)d'') \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r';_R exp_R(e);_R \eta_1(\ell')))d'' \wedge \\ &\qquad \mathbf{abs}(\eta'(\ell') \wedge_R (r';_R exp_R(e);_R \eta_2(\ell')))d'', \\ &\qquad \mathbf{abs}(r';_R exp_R(e);_R r_1)d'' \wedge \mathbf{abs}(r';_R exp_R(e);_R r_2)d'') \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'' \wedge \mathbf{abs}(r';_R exp_R(e);_R \eta_1(\ell'))d'' \wedge \\ &\qquad \mathbf{abs}(r';_R exp_R(e);_R \eta_2(\ell'))d'', \\ &\qquad \mathbf{abs}(r';_R exp_R(e);_R r_1)d'' \wedge \mathbf{abs}(r';_R exp_R(e);_R r_2)d'') \qquad (8)\end{aligned}$$

And using the fact that $\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = (exp_R(e);_R (\eta_1 \wedge_R \eta_2), exp_R(e);_R (r_1 \wedge_R r_2))$,

$$\begin{aligned}&\mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R (exp_R(e);_R (\eta_1(\ell') \wedge_R \eta_2(\ell')))), r' ;_R (exp_R(e);_R (r_1 \wedge_R r_2)))) \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R (exp_R(e);_R (\eta_1(\ell') \wedge_R \eta_2(\ell')))))d'',\end{aligned}$$

$$
\begin{array}{ll}
& \quad\quad\quad\quad \mathbf{abs}(r' ;_R (exp_R(e);_R (r_1 \wedge_R r_2)))d'') \\
= & \lambda(\eta'', d'').(\lambda \ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'' \wedge \mathbf{abs}(r';_R exp_R(e);_R \eta_1(\ell'))d'' \wedge \\
& \quad\quad\quad\quad \mathbf{abs}(r';_R exp_R(e);_R \eta_2(\ell'))d'', \\
& \quad\quad\quad\quad \mathbf{abs}(r';_R exp_R(e);_R r_1)d'' \wedge \mathbf{abs}(r';_R exp_R(e);_R r_2)d'') \\
= & \ (8)
\end{array}
$$

□

Proof of Theorem 5.1. The proof is by induction on the structure of P.

Case 1. $P = \mathsf{skip}$

$$
\begin{array}{l}
\mathbf{abs}_E(\mathcal{R}[\![\mathsf{skip}]\!]) = \mathbf{abs}_E((\top_{Env_R},\ id_R)) \\
\quad = \lambda(\eta', d').(\eta', \mathbf{abs}(id_R)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\top_{Env_R}(\ell'))(\eta'(\ell'))) \\
\quad = \lambda(\eta', d').(\eta',\ d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \top_{Data}) \\
\quad = \lambda(\eta', d').(\eta',\ d') \\
\quad = \mathcal{B}[\![\mathsf{skip}]\!]
\end{array}
$$

Case 2. $P = x = e$

$$
\begin{array}{l}
\mathbf{abs}_E(\mathcal{R}[\![x = e]\!]) = \mathbf{abs}_E((\top_{Env_R},\ asgn_R(x, e))) \\
\quad = \lambda(\eta', d').(\eta', \mathbf{abs}(asgn_R(x, e))d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\top_{Env_R}(\ell'))(\eta'(\ell'))) \\
\quad = \lambda(\eta', d').(\eta',\ asgn(x, e)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \top_{Data}) \\
\quad = \lambda(\eta', d').(\eta',\ asgn(x, e)d') \\
\quad = \mathcal{B}[\![x = e]\!]
\end{array}
$$

Case 3. $P = \mathsf{break}\ \ell$

$$
\begin{array}{l}
\mathbf{abs}_E(\mathcal{R}[\![\mathsf{break}\ \ell]\!]) = \mathbf{abs}_E((\top_{Env_R}[\ell \mapsto id_R],\ \top_R)) \\
\quad = \lambda(\eta', d').(\eta', \mathbf{abs}(\top_R)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\top_{Env_R}[\ell \mapsto id_R](\ell'))(\eta'(\ell'))) \\
\quad = \lambda(\eta', d').(\eta',\ \top_{Data} \wedge \mathbf{abs}(id_R)(\eta'(\ell))) \\
\quad = \lambda(\eta', d').(\eta',\ \eta'(\ell)) \\
\quad = \mathcal{B}[\![\mathsf{break}\ \ell]\!]
\end{array}
$$

Case 4. $P = \ell : P'$

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

$$
\begin{array}{l}
\mathcal{B}[\![P']\!] = \mathbf{abs}_E(\mathcal{R}[\![P']\!]) = \mathbf{abs}_E((\eta, r)) \\
\quad = \lambda(\eta', d').(\eta', \mathbf{abs}(r)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\eta(\ell'))(\eta'(\ell')))
\end{array}
$$

Now we work on $\ell : P'$.

$$\begin{aligned}
&\mathbf{abs}_E(\mathcal{R}[\![\ell : P']\!]) = \mathbf{abs}_E((\eta[\ell \mapsto \top_R], r \wedge_R \eta(\ell))) \\
&= \lambda(\eta', d').(\eta', \mathbf{abs}(r \wedge_R \eta(\ell))d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\eta[\ell \mapsto \top_R](\ell'))(\eta'(\ell'))) \\
&= \lambda(\eta', d').(\eta', \mathbf{abs}(r)d' \wedge \mathbf{abs}(\eta(\ell))d' \wedge \bigwedge_{\ell' \in \mathrm{Label}, \ell' \neq \ell} \mathbf{abs}(\eta(\ell'))(\eta'(\ell'))) \qquad (1)
\end{aligned}$$

And

$$\begin{aligned}
&\mathcal{B}[\![\ell : P']\!] = \lambda(\eta', d').\ \text{let}\ (\eta_1, d_1) \leftarrow \mathcal{B}[\![P']\!](\eta'[\ell \mapsto d'], d') \\
&\qquad\qquad \text{in}\ (\eta_1[\ell \mapsto \top_{Data}],\ d_1) \\
&= \lambda(\eta', d').\ \text{let}\ (\eta_1, d_1) \leftarrow (\eta'[\ell \mapsto d'], \mathbf{abs}(r)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\eta(\ell'))(\eta'[\ell \mapsto d'](\ell'))) \\
&\qquad\qquad \text{in}\ (\eta_1[\ell \mapsto \top_{Data}],\ d_1) \\
&= \lambda(\eta', d').(\eta'[\ell \mapsto \top_{Data}],\ \mathbf{abs}(r)d' \wedge \mathbf{abs}(\eta(\ell))d' \wedge \bigwedge_{\ell' \in \mathrm{Label}, \ell' \neq \ell} \mathbf{abs}(\eta(\ell'))(\eta'(\ell'))) \qquad (2)
\end{aligned}$$

Because we require all the programs to be legal, the incoming environment η' has ℓ mapped to $\top_{Data}$. This means that $\eta' = \eta'[\ell \mapsto \top_{Data}]$. So

$$\begin{aligned}
(2) &= \lambda(\eta', d').(\eta',\ \mathbf{abs}(r)d' \wedge \mathbf{abs}(\eta(\ell))d' \wedge \bigwedge_{\ell' \in \mathrm{Label}, \ell' \neq \ell} \mathbf{abs}(\eta(\ell'))(\eta'(\ell'))) \\
&= (1)
\end{aligned}$$

Case 5. $P = P_1; P_2$

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

$$\begin{aligned}
\mathcal{B}[\![P_1]\!] &= \mathbf{abs}_E(\mathcal{R}[\![P_1]\!]) \\
&= \mathbf{abs}_E((\eta_1, r_1)) \\
&= \lambda(\eta', d').(\eta', \mathbf{abs}(r_1)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\eta_1(\ell'))(\eta'(\ell')))
\end{aligned}$$

and

$$\begin{aligned}
\mathcal{B}[\![P_2]\!] &= \mathbf{abs}_E(\mathcal{R}[\![P_2]\!]) \\
&= \mathbf{abs}_E((\eta_2, r_2)) \\
&= \lambda(\eta', d').(\eta', \mathbf{abs}(r_2)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\eta_2(\ell'))(\eta'(\ell')))
\end{aligned}$$

Now we work on $P_1; P_2$.

$$\begin{aligned}
&\mathbf{abs}_E(\mathcal{R}[\![P_1; P_2]\!]) = \mathbf{abs}_E((\eta_1 \wedge_R (\eta_2;_R r_1), r_2;_R r_1)) \\
&= \lambda(\eta', d').(\eta', \mathbf{abs}(r_2;_R r_1)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}((\eta_1 \wedge_R (\eta_2;_R r_1))(\ell'))(\eta'(\ell'))) \\
&= \lambda(\eta', d').(\eta', \mathbf{abs}(r_2;_R r_1)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} (\mathbf{abs}(\eta_1(\ell'))(\eta'(\ell')) \wedge \mathbf{abs}(\eta_2(\ell');_R r_1)(\eta'(\ell')))) \qquad (3)
\end{aligned}$$

And

$$\begin{aligned}
&\mathcal{B}[\![P_1; P_2]\!] = \lambda(\eta', d').(\mathcal{B}[\![P_2]\!]; \mathcal{B}[\![P_1]\!])(\eta', d') \\
&= \lambda(\eta', d').\mathcal{B}[\![P_1]\!](\mathcal{B}[\![P_2]\!](\eta', d')) \\
&= \lambda(\eta', d').\mathcal{B}[\![P_1]\!](\eta', \mathbf{abs}(r_2)d' \wedge \bigwedge_{\ell' \in \text{Label}} \mathbf{abs}(\eta_2(\ell'))(\eta'(\ell'))) \\
&= \lambda(\eta', d').\big(\eta', \mathbf{abs}(r_1)\big(\mathbf{abs}(r_2)d' \wedge \bigwedge_{\ell' \in \text{Label}} \mathbf{abs}(\eta_2(\ell'))(\eta'(\ell'))\big) \wedge \big(\bigwedge_{\ell' \in \text{Label}} \mathbf{abs}(\eta_1(\ell'))(\eta'(\ell'))\big)\big) \\
&= \lambda(\eta', d').\big(\eta', \mathbf{abs}(r_2;_R r_1)d' \wedge \big(\bigwedge_{\ell' \in \text{Label}} \mathbf{abs}(\eta_2(\ell');_R r_1)(\eta'(\ell'))\big) \wedge \big(\bigwedge_{\ell' \in \text{Label}} \mathbf{abs}(\eta_1(\ell'))(\eta'(\ell'))\big)\big) \\
&= \lambda(\eta', d').(\eta', \mathbf{abs}(r_2;_R r_1)d' \wedge \bigwedge_{\ell' \in \text{Label}} (\mathbf{abs}(\eta_2(\ell');_R r_1)(\eta'(\ell')) \wedge \mathbf{abs}(\eta_1(\ell'))(\eta'(\ell')))) \\
&= (3)
\end{aligned}$$

We note that we used the distributivity property above.

Case 6. $P = \mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2$

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

$$\begin{aligned}
\mathcal{B}[\![P_1]\!] &= \mathbf{abs}_E(\mathcal{R}[\![P_1]\!]) \\
&= \mathbf{abs}_E((\eta_1, r_1)) \\
&= \lambda(\eta', d').(\eta', \mathbf{abs}(r_1)d' \wedge \bigwedge_{\ell' \in \text{Label}} \mathbf{abs}(\eta_1(\ell'))(\eta'(\ell')))
\end{aligned}$$

$$\begin{aligned}
\mathcal{B}[\![P_2]\!] &= \mathbf{abs}_E(\mathcal{R}[\![P_2]\!]) \\
&= \mathbf{abs}_E((\eta_2, r_2)) \\
&= \lambda(\eta', d').(\eta', \mathbf{abs}(r_2)d' \wedge \bigwedge_{\ell' \in \text{Label}} \mathbf{abs}(\eta_2(\ell'))(\eta'(\ell')))
\end{aligned}$$

Now we work on $\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2$.

$$\begin{aligned}
&\mathbf{abs}_E(\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!]) = \mathbf{abs}_E(((\eta_1 \wedge_R \eta_2);_R exp_R(e), (r_1 \wedge_R r_2);_R exp(e))) \\
&= \lambda(\eta', d').(\eta', \mathbf{abs}((r_1 \wedge_R r_2);_R exp(e))d' \wedge \bigwedge_{\ell' \in \text{Label}} \mathbf{abs}(((\eta_1 \wedge_R \eta_2);_R exp_R(e))(\ell'))(\eta'(\ell'))) \\
&= \lambda(\eta', d').(\eta', exp(e)(\mathbf{abs}(r_1 \wedge_R r_2)d') \wedge exp(e)\Big(\bigwedge_{\ell' \in \text{Label}} \mathbf{abs}(\eta_1(\ell') \wedge_R \eta_2(\ell'))(\eta'(\ell'))\Big)) \qquad (4)
\end{aligned}$$

Let $(\eta_1', d_1') = \mathcal{B}[\![P_1]\!](\eta', d')$ and $(\eta_2', d_2') = \mathcal{B}[\![P_2]\!](\eta', d')$. Then

$$\begin{aligned}
&\mathcal{B}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = \lambda(\eta', d').(\eta', exp(e)(d_1 \wedge d_2)) \\
&= \lambda(\eta', d').(\eta', exp(e)((\mathbf{abs}(r_1)d' \wedge \bigwedge_{\ell' \in \text{Label}} \mathbf{abs}(\eta_1(\ell'))(\eta'(\ell'))) \wedge \\
&\qquad\qquad (\mathbf{abs}(r_2)d' \wedge \bigwedge_{\ell' \in \text{Label}} \mathbf{abs}(\eta_2(\ell'))(\eta'(\ell'))))) \\
&= \lambda(\eta', d').(\eta', exp(e)(\mathbf{abs}(r_1 \wedge_R r_2)d' \wedge \bigwedge_{\ell' \in \text{Label}} \mathbf{abs}(\eta_1(\ell') \wedge_R \eta_2(\ell'))(\eta'(\ell'))))
\end{aligned}$$

$$= \lambda(\eta', d').(\eta', exp(e)(\mathbf{abs}(r_1 \wedge_R r_2)d') \wedge exp(e)\Big(\bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\eta_1(\ell') \wedge_R \eta_2(\ell'))(\eta'(\ell'))\Big))$$

$$= (4)$$

We note that we used the distributivity property above.

□