December 2018

Waddle - Always-canonical Intermediate Representation

Eric Drew Fritz
University of Wisconsin-Milwaukee

WADDLE – ALWAYS-CANONICAL INTERMEDIATE REPRESENTATION

by

Eric Fritz

A Dissertation Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

in Engineering

at

The University of Wisconsin-Milwaukee

December 2018

ABSTRACT

WADDLE – ALWAYS-CANONICAL INTERMEDIATE REPRESENTATION

by

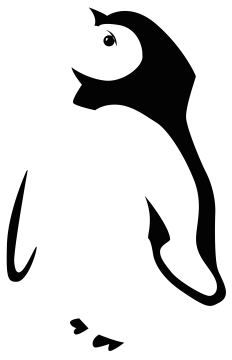
Eric Fritz

The University of Wisconsin-Milwaukee, 2018
Under the Supervision of Professor John Boyland

Program transformations that are able to rely on the presence of *canonical properties* of the program undergoing optimization can be written to be more robust and efficient than an equivalent but generalized transformation that also handles non-canonical programs. If a canonical property is required but broken earlier in an earlier transformation, it must be rebuilt (often from scratch). This additional work can be a dominating factor in compilation time when many transformations are applied over large programs. This dissertation introduces a methodology for constructing program transformations so that the program remains in an *always-canonical* form as the program is mutated, making only local changes to restore broken properties.

TABLE OF CONTENTS

LIST OF FIGURES



1 Introduction

This dissertation introduces a methodology for the construction of an optimizing compiler. Such an optimizer incrementally maintains *canonical properties* of control flow graph and ensures that auxiliary data structures remain in a valid state as the control flow graph undergoes mutations. Such mutations include the insertion and deletion of edges, the addition and removal of blocks to and from the graph, and the movement of instructions.

The primary goal of such a methodology is to reduce or eliminate the need to reconstruct (from scratch) the canonical properties that are expected to be present in later transformations. This extra work can lead to a significant increase in runtime if done frequently. Often, a mutating transformation will break a property in a local way, allowing the property to be repaired with little complexity.

This methodology is the core of the Waddle compiler project, named after its slow and awkward steps toward progress. The compiler includes an optimizing backend that is focused on control flow optimizations operating on an intermediate representation that never breaks canonical properties outside of a single transformation.

1.1 Motivation

Many existing compiler infrastructures express the program under transformation by a well-defined internal representation. The LLVM Compiler Infrastructure, the GNU Compiler Collection, and Oracle's HotSpot Java Virtual Machine all use a variation of *static single assignment form* internally. The properties of such a representation enable more precise reasoning of the program during transformation and often makes the transformation itself more efficient.

Unfortunately, core properties of the representation may be invalidated by mutation of the control flow graph. If a property is potentially invalidated during program transformation, no transformation can rely on its presence. Transformations themselves could be restricted to the set of operations that do not invalidate the representation. However, this would drastically decrease the capabilities of the optimizer as the simplest of transformations can easily break a flowgraph in static single assignment form.

Instead, these properties should be *repaired* after being broken by a transformation. It is often too inefficient to rebuild the representation after each transformation from scratch. Many transformation passes in LLVM, for example, make an attempt to repair as much of the internal form and external analysis information as possible *as the transformation occurs*. Incrementally maintaining such representations *in general* can be intractable, but in the context of a specific transformation, there is usually a clear process to fix broken properties in a local manner. Where applicable, such repairs are performed in an ad-hoc manner and by hand.

There are two major problems with this approach: *fragility* and *obscurity*. Because the responsibility of repairing broken properties is not enforced by the compiler, there may be transformations that completely destroy a property, then neglects to repair it. Any future transformation that depends on that analysis must recompute it from scratch or risk unsoundess, resulting in a fragile transformation implementation. For transformations that do attempt to maintain the internal representation, the logic of the transformation is now intermixed with logic to repair the internal representation. This logic may be non-trivial, depending on the complexity of the transformation. This negatively affects both readability and maintainability of a single transformation, resulting in obscure transformation implementations.

Waddle attempts to attenuate these problems by representing programs in a strict, never-broken *canonical form*. The structure of the optimizer forces transformation to leave the program in a semantically-equivalent canonical form. It is an error on the part of the programmer of the transformation to break this invariant, and such an error causes a hard fault during execution of the optimizer. This ensures that canonical properties are *robust* rather than *fragile*, and their presence can be guaranteed at the beginning of each transformation.

Each transformation is written, so far as possible, as a composition of smaller and more general operations over the program where each operation maintains canonical invariants. This ensures that transformations are *intelligible* rather than *obscure*, leaving the heavy repairs to be performed by a common set of operations shared among transformations.

1.2 Research Contributions

This dissertation makes four major contributions to this line of research, as follows.

(1) A simple kernel IR and deterministic evaluation semantics that is suitable for analysis of control flow optimizations.

(2) A family of algorithms that incrementally maintains canonical properties of a control flow graph as it is mutated (e.g. the insertion, deletion, or relocation of a flow edge, the movement of an instruction, or the insertion or deletion of a block).

(3) An illustration of how the application and composition of these algorithms can together form non-trivial transformations (e.g. jump threading, loop unswitching, conditional simplification). Each transformation maintains canonical properties of the program under transformation, allowing transformations to be reordered and repeated arbitrarily without requiring intermediate phases to rebuild such properties.

(4) An implementation that follows this methodology, and a short experimental evaluation of the implementation.

1.3 Organization

Chapter 2 provides a foundation for prerequisite concepts related to control flow graphs, domination, and loop structures, as well as a common notation for sequences and multisets. Chapter 3 formally defines a syntax, an operational semantics, and a sound type system for Waddle's internal representation. Chapter 4 defines the properties of Waddle's *canonical form*. Chapter 5 presents related algorithms for constructing and reconstructing canonical properties, some of which are used directly by Waddle. Chapter 6 outlines what constitutes a *transformation* and gives a brief proof structure that will be used in the appendices for the following chapters. Chapter 7 presents a family of algorithms to canonicalize a program and later maintain canonical form. Chapter 8 presents a family of algorithms that are used as common subprocedures by other transformations. Chapter 9 presents a bag of classic compiler optimizations written using Waddle's always-canonical methodology, using the subprocedures defined in the previous chapter. To conclude, Chapter 10 provides a preliminary evaluation of the optimizing compiler and Chapter 11 discusses thoughts on future directions for this line of research.

2 Preliminaries

This chapter briefly defines preliminary concepts required to engage with the remainder of the work (namely domination and the definition of reducible loops). This chapter also defines notation for concepts where existing notation is insufficiently standardized.

2.1 Sequences

A **sequence** S is an ordered set of items denoted $\langle s_1, s_2, \ldots, s_n \rangle$. The empty sequence is denoted $\langle\rangle$. We denote the **extension** of one sequence with the elements of another as $S_1 \cup S_2$ such that the resulting set has all elements of S_1 followed by all elements of S_2 in their original order.

2.2 Multisets

A **multiset** M over a ground set S is defined as the mapping $M\colon S \to \mathbb{N}$ where for $s \in S$ the number $M(s)$ denotes the **multiplicity** of s in M. The **root** of a multiset, denoted $root(M)$, is defined as the set of elements with a non-zero multiplicity, $\{s \in S \mid M(s) > 0\}$. The **cardinality** of a multiset, denoted $|M|$, is defined as the sum of its multiplicities, $\sum M(s)$. Multiset membership is denoted by $s \in^k M$ when $M(s) = k$ or $s \in M$ when $M(s) \geq 1$. A classic set S' may be interpreted as a multiset M over the ground set $S \subseteq S'$ such that $M(s) = 1$ if $s \in S'$ and $M(s) = 0$ otherwise for all $s \in S$.

If M_1 and M_2 are multisets over the same ground set S, then $M_1 \subseteq M_2$ if $M_1(s) \leq M_2(s)$ for all $s \in S$. Their union, intersection, difference, and sum are defined as follows where the symbol $\dot{-}$ denotes truncated subtraction (bounded below by zero).

$$(M_1 \cup M_2)(s) = \max(M_1(s),\ M_2(s)) \qquad (M_1 - M_2)(s) = M_1(s) \dot{-} M_2(s)$$

$$(M_1 \cap M_2)(s) = \min(M_1(s),\ M_2(s)) \qquad (M_1 \uplus M_2)(s) = M_1(s) + M_2(s)$$

If M is a multiset of the ground set S, then all of the elements from the set $S' \subseteq S$ can be removed from M by the following operation.

$$(M \setminus S')(s) = \begin{cases} 0 & s \in S' \\ M(s) & \text{otherwise} \end{cases}$$

The empty multiset in which all multiplicities are zero is denoted by $\emptyset$ when the ground set over which the multiset is defined is unambiguous. $\{s_{\langle k \rangle} \mid P(s,k)\}$ defines the multiset M where $s \in^k M$ if the predicate $P(s,k)$ is satisfied and $s \notin M$ otherwise. If the $\langle k \rangle$ subscript is omitted, then it is assumed that the multiplicities are transferred to the resulting multiset whenever the $\in$ or $\in^k$ relationships are used in the predicate. For example, if $M = \{1,2,3,3,4,4,4\}$ then $\{2x \mid x \in M, x > 2\} = \{6,6,8,8,8\}$.

For sets and multisets representing binary relations (such as edges in a graph), we use $dom(M)$ to mean the set of elements in the domain of the mapping and $codom(M)$ to mean the set of elements in the codomain of the mapping.

2.3 Labels

Let $\mathcal{L}^b$ be an infinite set of abstract block labels and let $\mathcal{L}^f$ be an infinite set of abstract function labels. A single label is denoted ℓ. *Basic blocks* and *functions* are further defined in Chapter 3. For now, it is sufficient to know that $lab(b) \in \mathcal{L}^b$ and $lab(f) \in \mathcal{L}^f$ denote respectively the label associated with block b and function f, and $target(b)$ denotes the multiset of block labels that are referenced as the target of a branch originating from the block b.

2.4 Control Flow Graph

A **control flow graph** $G = (B, \ell_0)$ is a directed multigraph that encodes the branching structure between a set of blocks B. Let $\mathit{block}_G(\ell)$ denote the block $b \in B$ such that $\mathit{lab}(b) = \ell$ and let $\mathcal{L}_B^b = \{\mathit{lab}(b) \mid b \in B\}$ denote the set of block labels in B. $E(G)$ is a multiset of edges over the ground set $B \times B$ such that $(b_1, b_2) \in^k E(G)$ if and only if block b_1 contains exactly k references to the label ℓ_{b_2}. It is legal for a block to be adjacent to itself (self-loops), or to be adjacent to another block multiple times (parallel edges). The block $\mathit{entry}(G) = \mathit{block}_G(\ell_0)$ is the designated root block from which all other blocks $b \in B$ are reachable. For a control flow graph to be well-formed, it must be the case that the labels of all blocks in B are distinct, $\mathit{target}(b) \subseteq \mathcal{L}_B^b$ for each block $b \in B$, and $\ell_0 \in \mathcal{L}_B^b$. The visual conventions for control flow graphs are illustrated in Figure 2.1.

Let $\mathit{succ}_G(b) = \{b' \mid (b, b') \in E(G))\}$ denote the set of blocks to which b refers in G and let $\mathit{pred}_G(b) = \{b' \mid (b', b) \in E(G)\}$ denote the set of blocks that refer to block b in G. Let $b_1 \rightsquigarrow_G b_2$ denote the existence of a path from b_1 to b_2 in G and let $b_1 \not\rightsquigarrow_G b_2$ denote the non-existence of such a path. Subscript qualifiers may be dropped when the context resolves any ambiguity. If $\mathit{succ}_G(b_1) = \{b_2\}$ and $\mathit{pred}_G(b_2) = \{b_1\}$, then (b_1, b_2) is a **non-critical edge**. All other edges are **critical edges**.

For complexity analysis, the size of the block and edge sets are abbreviated as $n = |B|$ and $m = |E(G)|$. It is worth noting that m is $\mathcal{O}(n)$ for typical control flow graphs. The *extended sizes* of a set of blocks $B' \subseteq B$ are defined as follows, which accounts for the number

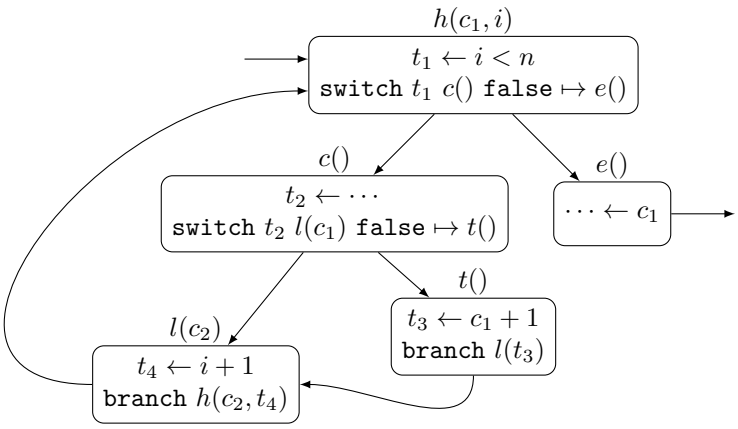


Figure 2.1: An example control flow graph containing a loop and a one-armed conditional. Block content is presented in Section 3.1.

of edges connecting the set of blocks to the rest of the graph.

$$\|B'\|^{\rightarrow} \equiv |B'| + \sum_{(b_1,b_2)\in E} E((b_1, b_2))[b_1 \in B']$$

$$\|B'\|^{\leftarrow} \equiv |B'| + \sum_{(b_1,b_2)\in E} E((b_1, b_2))[b_2 \in B']$$

$$\|B'\|^{\leftrightarrow} \equiv |B'| + \sum_{(b_1,b_2)\in E} E((b_1, b_2))[b_1 \in B' \vee b_2 \in B']$$

Let $G\langle B'\rangle$ denote the subgraph of the control flow graph $G = (B, \ell_0)$ **induced** by the set of blocks $B' \subseteq B$ such that $G\langle B'\rangle = (B', \{(b_1, b_2) \in E \mid b_1, b_2 \in B'\})$. Such induced subgraphs are not necessarily control flow graphs as two blocks may be disconnected, targets may refer outside of the subgraph, or there may be no designated root node.

2.4.1 Reducibility

A control flow graph is **reducible** if it does not contain a strongly connected component with two or more predecessors. Equivalently, if the *limit graph* obtained by the repeated application of the following transformations contains a single block, the original graph is reducible [4]. Every graph produces a unique limit graph regardless of the order in which the transformations are applied [35]. These operations are illustrated in Figure 2.2.

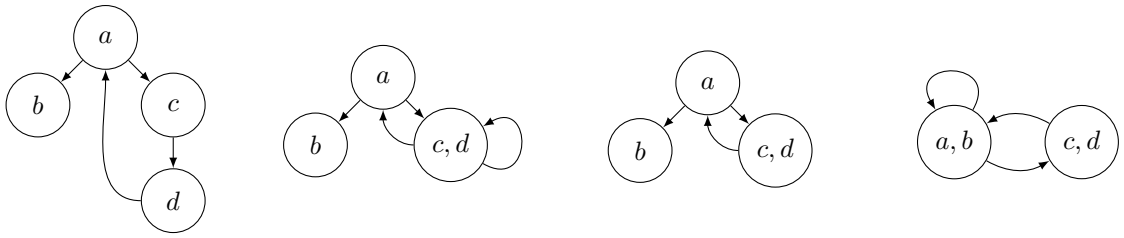


Figure 2.2: The process of collapsing a reducible graph into its limit graph (incomplete).

(T1) Remove a self-loop edge $(b, b) \in E$.

(T2) Find two blocks a and b such that $pred_G(b) = \{a\}$. Replace both blocks with a representative block c and replace all edges incident to either a or b with an edge incident to c.

If a graph is irreducible, then the limit graph obtained by exhaustively applying the transformations above contains more than one block. Such a representative block encodes an irreducible region of the original graph and must have more than one predecessor. An irreducible region can be transformed into an *equivalent* reducible region by applying the following *node splitting* transformation.

(T3) Duplicate a block b with $n > 1$ predecessors and its outgoing edges so that each predecessor references its own copy of b.

Once this transformation is applied, the second transformation is immediately applicable as all of the duplicated nodes have a single predecessor. The repeated application of all three transformations will always result in a trivial limit graph. If the reduction is reversed such that the copies of blocks are left in place, the resulting graph is an equivalent but reducible control flow graph. This operation is illustrated in Figure 2.3.

Transforming an irreducible graph into an equivalent reducible graph comes at the cost of maintaining additional blocks (and, ultimately, code size) that can be exponential in theory. The choice and order of blocks on which the node splitting transformation is performed is

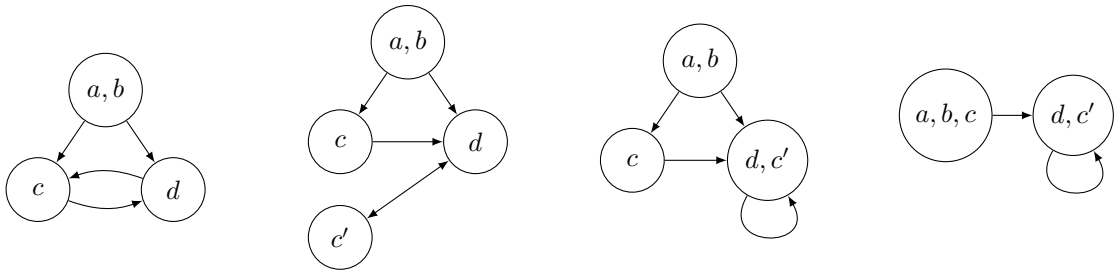


Figure 2.3: The process of collapsing an irreducible graph into its limit graph (incomplete). Transformation (T3) is applied to block b so that transformations (T1) and (T2) can further reduce the graph (incomplete).

significant and directly determines the number of duplicated blocks in the resulting graph. *Controlled node splitting* helps mitigate the problem by choosing an optimal sequence of block splits (exhaustively or with heuristics) [36, 65]. Carter et al. [18] later showed that this complexity is unavoidable in practice for some classes of graphs. Specifically, any reducible graph equivalent to the complete graph on n nodes must have at least 2^{n-1} blocks. Fortunately, the majority of programs written by humans are reducible, and the irreducible regions that do occur are rarely pathological. Common structured programming constructs (e.g. sequence, `if`, `for`, `while`, `do-while`, `switch`, labeled and unlabeled `break` and `continue`) create trivially reducible control flow graphs, and the use of arbitrary branching (e.g. `goto`) creates an irreducible control flow graph only if the target of the branch is in a disjoint control flow cycle (i.e. targeting the body of a loop).

2.4.2 Induced Trees

Let $T_G = (V, E')$ denote a rooted, connected tree induced by the labels of the control flow graph $G = (B, \ell_0)$ where $V = \mathcal{L}_B^b$ and ℓ_0 is the root. The edge sets $E(G)$ and E' have no fixed relation; that is, it is not necessarily the case that $(lab(b_1), lab(b_2)) \in E'$ if $(b_1, b_2) \in E(G)$, or vice versa. Examples of such trees and selections of E' are given in Section 2.4.3 and Section 2.5.1.

Let $b_1 \rightarrow_{T_G} b_2$ denote the existence of a path from $lab(b_1)$ to $lab(b_2)$ in T_G and, more specifically, let $b_1 \overset{+}{\rightarrow}_{T_G} b_2$ denote the existence of a non-empty path (where $b1 \neq b_2$). Let $b_1 \not\rightarrow_{T_G} b_2$ and $b_1 \overset{+}{\not\rightarrow}_{T} b_2$ denote the non-existence of such paths. Let $T_G\langle b\rangle$ denote the subtree of T_G rooted at label $lab(b)$, $p_{T_G}(b)$ denote the block in G whose label is the parent of the label $lab(b)$ in T_G, $child_{T_G}(b)$ denote the set of children of b in T_G, $ancestor_{T_G}(b)$ denote the set of blocks whose labels are the ancestors of the label $lab(b)$ in T_G, and $nca_{T_G}(\{b_1, \ldots, b_k\})$ denote the block in G whose label is the nearest common ancestor of the set of labels $\{lab(b_i) \mid b_1, \ldots, b_k\}$ in T_G.

2.4.3 Depth-First Spanning Tree

A **depth-first spanning tree** DFS_G is an induced tree of G generated by a depth-first traversal starting from its entry block. Let $Ord(DFS_G)$ denote a total ordering of blocks encoding DFS_G such that b_1 occurs before b_2 if and only if $lab(b_1)$ occurs before $lab(b_2)$ in a chosen pre-order traversal of DFS_G. If multiple traversals are possible one can be selected arbitrarily. We sometimes refer to a block in G by its pre-order rank; in particular, $b1 <_{Ord(T_G)} b_2$ means that b_1 occurs before b_2 in $Ord(T_G)$.

2.5 Domination

A block b_1 **dominates** block b_2 in the control flow graph G, denoted $b_1 \preceq_G b_2$, if and only if b_1 occurs on all paths from $entry(G)$ to b_2 in G. Let $dom_G(b) \supseteq \{entry(G), b\}$ denote the set of dominators of block b in G. A block b_1 *strictly* dominates b_2 in G, denoted $b_1 \prec_G b_2$, if $b_1 \preceq_G b_2$ and $b_1 \neq b_2$. The *immediate* dominator of a block $b \neq entry(G)$ in G, denoted $idom_G(b)$, is the unique block $idom_G(b) \in dom_G(b)$ that is dominated by each block $dom_G(b) \setminus \{b\}$ in G.

The **dominance frontier** of a block b in G is the set of blocks $DF_G(b)$ in which b dominates a predecessor of b in G but not b itself, defined formally as follows.

$$DF_G(b) = \{b' \mid b \preceq_G p \wedge b \not\preceq_G b',\ p \in pred_G(b')\}$$

Informally, the dominance frontier of a block b in G consists of all the blocks that are *just barely* not dominated by b. For notational convenience, we extend this definition to sets of blocks as follows,

$$DF_G(B') = \bigcup_{b \in B'} DF_G(b)$$

which denotes the set of blocks that are *just barely* not dominated by all of the blocks in B'. The *iterated* dominance frontier of a set of blocks in G, denoted $DF_G(B')$, is the fixed point of the following relation.

$$\begin{aligned} DF_G^1(B') &= DF_G(B') \\ DF_G^i(B') &= DF_G(B' \cup DF_G^{i-1}(B')) \\ DF_G(B') &= \bigcup_i DF_G^i(B') \end{aligned}$$

2.5.1 Dominator Tree

The dominance relation of a control flow graph G is transitive and can be represented compactly by a *dominator tree* D_G. A **dominator tree** is an induced tree of a control flow graph $G = (B, \ell_0)$ in which the parent of each label $\ell \in \mathcal{L}_B^b$, except when $\ell \neq \ell_0$, is the label of $block_G(\ell)$'s immediate dominator. An example dominator tree is illustrated in Figure 2.4.

2.6 Loops

A **natural loop** is a single-entry, maximal strongly connected component of a control flow graph containing one block that dominates all other blocks. Loops are defined abstractly

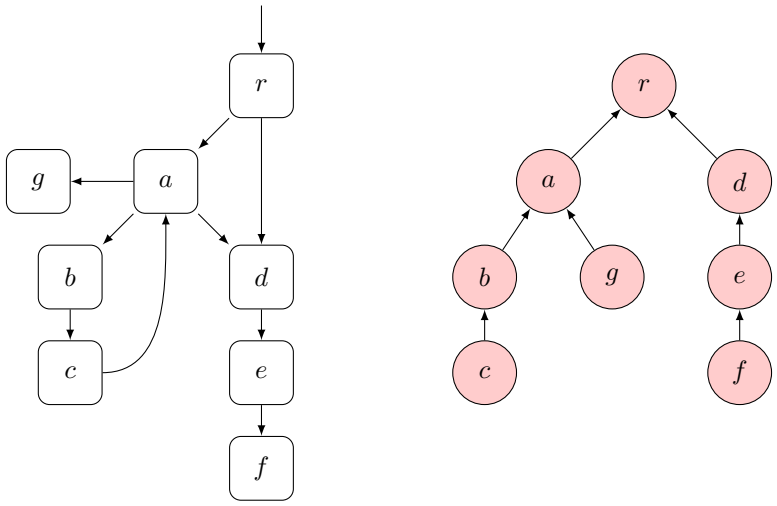


Figure 2.4: A control flow graph with its dominator tree. An edge (b_1, b_2) in the dominator tree indicates that $idom(b_2) = b_1$.

as the tuple $l = (\ell, L, X)$, where $\ell \in L$, $L \subset \mathcal{L}^b$, $X \subset \mathcal{L}^b$, and $L \cap X = \emptyset$. The following properties of the loop l are defined with respect to a control flow graph $G = (B, \ell_0)$ such that $L \subseteq \mathcal{L}^b_B$ and $X \subset \mathcal{L}^b_B$.

The **body** of l, denoted $body_G(l)$, is the set of blocks $\{block_G(\ell) \mid \ell \in L\}$. The **header** of l, denoted $header_G(l)$, is the block $block_G(\ell) \in body_G(l)$ that dominates all other blocks in $body_G(l)$. We use $lab(l) = \ell$ to refer to the label of the loop's header block. An **exit** of l is a block $b \notin body_G(l)$ such that $pred_G(b) \cap body_G(l) \neq \emptyset$. The exits of l are denoted by the set $exit_G(l) = \{block_G(\ell) \mid \ell \in X\}$.

A **backedge** of G is an edge between blocks b_1 and b_2 such that $b_2 \prec_G b_1$; every other edge is a **forward edge**. The set of backedges of G are denoted by $\overleftarrow{E}(G)$ and the set of forward edges by $\overrightarrow{E}(G)$. A **latch** of l is a block $b \in body_G(l)$ identified by the backedge $(b, header_G(l))$. Let $latch_G(l)$ denote the set of latches of l and the unique latch of l when canonical form (discussed in Section 4.3) is assumed. A **preheader** of l is a block $b \notin body_G(l)$ such that $succ_G(b) = \{header_G(l)\}$ and $pred_G(header_G(l)) = \{b\}$. Each loop has at most one preheader and is denoted by $preheader_G(l)$, if it exists.

2.6.1 Loop Nesting

Let l_c and l_p be distinct loops of the same control flow graph G. If $body_G(l_c) \subset body_G(l_p)$, then loop l_c is **nested** within loop l_p. l_c is referred to as the **inner loop** and l_p is referred to as the **outer loop**. If there does not exist a loop l_i such that $body_G(l_c) \subset body_G(l_i) \subset body(l_p)$, then l_c is **immediately nested** in l_p.

Lemma 2.6.1. If l_c and l_p are distinct loops in the control flow graph G, then the loops are either disjoint ($body(l_c) \cap body(l_p) = \emptyset$) or nested ($body(l_c) \subset body(l_p)$), without loss of generality.

Proof. Let $X = body(l_c) \cap body(l_p)$. Suppose $X \neq \emptyset$, $X \neq body(l_c)$, and $X \neq body(l_p)$. Let p be the unique path $entry(G) \rightarrow_{D_G} x$ for some $x \in X$. As the headers of both loops dominate all blocks in X, $header(l_c), header(l_p) \in p$. If $header(l_p)$ occurs first in p, then all blocks in l_c are dominated by $header(l_p)$ and $X = body(l_c)$. Symmetrically, if $header(l_c)$ occurs first in p, then $X = body(l_p)$. □

Lemma 2.6.2. If loop l_c is nested within the loop l_p in the control flow graph G, then $exit(l_c) \cap body(l_p) \neq \emptyset$.

Proof. If $exit(l_c) \cap body(l_p) = \emptyset$, then $b \not\rightsquigarrow_{G\langle body(l_p)\rangle} header(l_p)$ for all $b \in body(l_c)$ and $body(l_p)$ does not represent a strongly connected component of G. □

2.6.2 Loop Nesting Forest

The loop nesting structure of a control flow graph G is transitive and can be represented compactly by a *loop nesting forest* F_G. A **loop nesting forest** is a forest of loops in which the children of a loop l in G are the set of loops that are immediately nested in l. The roots of the forest, denoted by $root(F)$, are the outermost loops of G. An example loop nesting forest is illustrated in Figure 2.5.

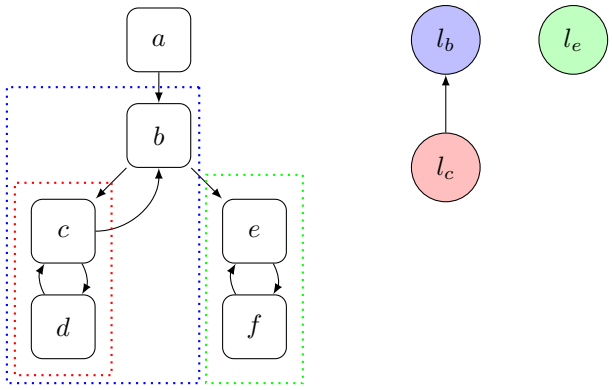


Figure 2.5: A control flow graph containing three loops (one nested) shown with its loop nesting forest. In this example, the set of abstract loops are $l_b = (b, \{b, c, d\}, \{e\})$, $l_c = (c, \{c, d\}, \{b\})$, and $l_e = (e, \{e, f\}, \emptyset)$.

Let $F_G\langle l\rangle$ denote the connected subtree of the loop nesting forest F_G rooted at loop l, $p_{F_G}(l)$ denote the parent of a loop l in F_G, $child_{F_G}(l)$ denote the set of immediate children of l in F_G, $ancestor_{F_G}(l)$ denote the set of ancestors of l in F_G, and $nca_{F_G}(\{l_1, \ldots, l_k\})$ denote the nearest common ancestor of a set of $k > 0$ loops in F_G. Let $loop_{F_G}(b)$ denote the *smallest* loop $l \in F_G$ such that $b \in body(l)$. We adopt the convention that the *phony* loop $l_\varnothing$ is the result of nca_{F_G} on a set of loops in disjoint trees of F_G and the result of $loop_{F_G}(b)$ when b does not occur in the body of any loop of G.

The **depth** of a loop l in a loop nesting forest F_G, denoted $depth_{F_G}(l)$, is the length of the unique path from a root in F_G to l. By convention, the depth of the phony loop $l_\varnothing$ is -1. We extend this definition to blocks, defined as $depth_{F_G}(b) = depth_{F_G}(loop_{F_G}(b))$.

2.6.3 Loop Deconstruction

The loops and the loop nesting forest F of a control flow graph G can be deconstructed into the triple (H_F, L_F, X_F) defined as follows.

$$H_F = \{(lab(l_c), lab(l_p)) \mid l_c, l_p \in F \wedge p_F(l_c) = l_p\}$$

$$L_F = \{(lab(b), lab(h)) \mid l \in F \wedge header_G(l) = h \wedge loop_G(b) = l\}$$

$$X_F = \{(lab(b), lab(h)) \mid l \in F \wedge header_G(l) = h \wedge b \in exit_G(l)\}$$

The set H_F represents the nesting relation of loops in F as pairs of block labels such that $(\ell_c, \ell_p) \in H_F$ when the loop with the header label ℓ_c is immediately nested in the loop with the header label ℓ_p. The set L_F maps blocks to the inner-most loop in F that contains it. Together, H_F and L_F represent the bodies of all loops in F. The set X_F represents the exit sets of loops in F such that $(\ell_b, \ell_l) \in X_F$ when the block with label ℓ_b exits the loop with the header label ℓ_l.

The set of header labels are nested under the loop in H with header label ℓ, denoted $N_H(\ell)$, is the fixed point of the following relation.

$$N_H^1(\ell) = \{\ell\}$$

$$N_H^i(\ell) = N_H^{i-1}(\ell) \cup \{\ell_c \mid (\ell_c, \ell_p) \in H \wedge \ell_p \in N_H^{i-1}(\ell)\}$$

$$N_H(\ell) = \bigcup_i N_H^i(\ell)$$

A loop l with the header label ℓ can be reconstructed from a triple (H, L, X), denoted $l_{(H,L,X)}(\ell)$, as follows.

$$(\ell, \{\ell_b \mid (\ell_b, \ell_l) \in L \wedge \ell_l \in N_H(\ell)\}, \{\ell_b \mid (\ell_b, \ell) \in X\})$$

Given the triple (H, L, X) and the set of loops L reconstructed from it, a loop nesting relation F can be reconstructed by making $l_{(H,L,X)}(\ell_c)$ a child of $l_{(H,L,X)}(\ell_p)$ in F when $(\ell_c, \ell_p) \in H$.

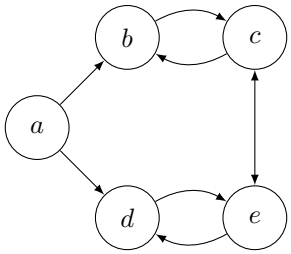


Figure 2.6: An irreducible graph.

2.6.4 Identification of Reducible Loops

The classic algorithm for finding loops is Tarjan's interval finding algorithm [62] that runs in linear time. The identification algorithm takes advantage of the fact that the header necessarily dominates all blocks in single-entry loops and is therefore restricted to reducible graphs. A backedge (l, h) identifies the header (and a latch) of a loop. The loop body can be determined by traversing the control flow graph in reverse starting from l before reaching h. The loop nesting structure can be determined by *collapsing* an identified loop into a single representative node until no backedges remain. Inner loops are collapsed first by following a post-order traversal of the dominator tree.

2.6.5 Identification of Irreducible Loops

In an irreducible control flow graph, it may not be the case that backedges identify a *single entry* connected component. Reducible loops can be generalized to irreducible loops that allow multiple entries. Unlike natural loops, there is no guaranteed domination property with respect to the set of loop entries. However, irreducible loops are generally defined in such a way that they, like reducible loops, are either properly nested or completely disjoint. Thus a loop nesting forest is still a valid encoding of both reducible and irreducible loops.

Steensgaard, Sreedhar et al., and Havlak each provide an algorithm for identifying loop bodies and loop nesting structure for arbitrary graphs [34,59,60]. Each algorithm identifies a

different set of loops. For the irreducible graph shown in Figure 2.6, Steensgaard's algorithm identifies the loops $\{c, e\}$ and $\{b, c, d, e\}$; Sreedhar et al.'s algorithm identifies only the loop $\{b, c, d, e\}$; and Havlak's algorithm identifies the loops $\{d, e\}$, $\{c, d, e\}$, and $\{b, c, d, e\}$. All three algorithms take $\mathcal{O}(n^2)$ time, but modifications by Ramalingam reduce the bound for Sreedhar et al. and Havlak's algorithm to be nearly-linear [54]. Sreedhar et al.'s algorithm requires a dominator tree being built in advance.

3 Internal Representation

In this chapter we formally define the Waddle internal representation. Section 3.1 discusses the syntax of programs, functions, blocks, instructions, values, and types. Section 3.2 presents an operational semantics for the internal representation. Section 3.3 presents a typesystem to ensure that a well-typed program can be evaluated without 'getting stuck'.

There have been efforts to formalize the LLVM internal representation, most notably the Vellvm project [67, 68]. Unfortunately, the LLVM internal representation has non-deterministic semantics, concerns itself with the shape of its data, and tracks allocation regions (differentiating stack and heap allocations). These features are unnecessary for transformations that focus their concern on control flow.

The Waddle internal representation, while similar to the LLVM internal representation, is designed to be a more suitable *kernel language* which can be later extended for a richer set of features for different target concrete internal representations. Languages such as Swift and Rust are powered by LLVM, but define their own intermediate representations known as SIL [31] and MIR [46], respectively. These internal representations contain additional high-level semantic information about the source program. The Swift intermediate language is used for optimizations relating to retain/release optimizations for its reference counting garbage collection as well as dynamic method de-virtualization and closure inlining. The Rust *mid-level* IR is used to encode more information about the lifetime of values. A goal for the design of the Waddle IR is to serve as the design basis for such an internal representation so that the transformations described later in this work can be applied with minimal extension.

3.1 Syntax

This section outlines the basic concepts of the internal representation along with a formal syntax. This syntax is very similar to the textual representation of the internal representation as used by the proof-of-concept implementation.

3.1.1 Blocks, Functions, and Programs

A **block**, defined as $b = (\ell, \langle \overline{r_i \colon t_i} \rangle, \{ \overline{r_j \colon t_j c} \}, \langle I_1, \ldots, I_k, T \rangle)$, represents a maximal fragment of a program absent of control flow (specifically branches). A block tuple contains, in order, its label, a possibly empty ordered sequence of *block parameters*, a set of *implicit block parameters*, and an *instruction stream*. The instruction stream consists of a possibly empty ordered sequence of *instructions* followed by exactly one *terminator*. A terminator may contain multiple *block references* that denote the blocks in which evaluation may continue. Block *components* are defined in Section 3.1.3. Given a block b, its label, block parameters, implicit parameters, instruction stream, instructions, and terminator may be referenced as $\mathit{lab}(b)$, $\mathit{param}(b)$, $\mathit{implicit}(b)$, $\mathit{stream}(b)$, $\mathit{inst}(b)$, and $\mathit{term}(b)$, respectively.

A **function**, defined as $f = (\ell, B, \ell_0, t)$, represents an invocable fragment of a program that includes control flow. The function extends the control flow graph $G = (B, \ell_0)$ with a function label $\ell \in \mathcal{L}^f$ and a return type annotation t. Let $\mathit{lab}(f)$ denote the label of function f, and let $\mathit{ret}(f)$ denote the return type annotation of f. When convenient, we may use f in place of G as a subscript qualifier for notation defined in Chapter 2 and let $E(f)$ denote the mutliset of edges for the control flow graph of the function f. A function's block set and entry block may be referenced as $\mathit{body}(f)$ and $\mathit{entry}(f)$, respectively. Let $\mathit{block}_f(\ell)$ denote the block $b \in \mathit{body}(f)$ such that $\mathit{lab}(b) = \ell$. Let $\mathit{exit}(f)$ denote the set of blocks in $\mathit{body}(f)$ that have no successors (blocks containing a `return` terminator) and let $\mathit{loop}(f)$ denote the roots

of the function's loop nesting forest. The function's annotated return type t, together with the types of the entry block's parameters, form the type of the function, denoted $f\colon (\overline{t_i}) \to t$. Given a block $b \in \mathit{body}(f)$, b can be referenced by the path $\langle f, b\rangle$ in order to distinguish it from an identical block occurring in another function.

An **intrinsic** function, defined as $\hat{f} = (\ell, (\overline{t_i}) \to t)$, represents an opaque invocable object that is not explicitly defined by the program. Intrinsics provide an escape hatch from the internal representation, allowing interaction with external systems (e.g. reading from or writing to a device) and some degree of non-determinism (e.g. time and random number generator) that is not otherwise defined by these semantics. Let $\mathit{lab}(\hat{f})$ denote the label ℓ of the intrinsic $\hat{f}$ and let $\hat{f}\colon (\overline{t_i}) \to t$ denote the type annotation of $\hat{f}$.

Let $\hat{F}$ denote the finite set of all intrinsincs and let $\mathcal{L}^{\hat{f}} = \{\mathit{lab}(\hat{f}) \mid \hat{f} \in \hat{F}\}$ denote the set of all intrinsic labels. The set of intrinsics are considered to be built-in to the language and are available to all defined *programs* with no difference in labels, types, or semantics.

A **program** p is a set of functions. To prevent ambiguity at function callsites, it is assumed that no two functions in p share the same label and no label of a function in p occurs in $\mathcal{L}^{\hat{f}}$. Let $\mathit{func}_p(\ell)$ denote the function $f \in p$ such that $\mathit{lab}(f) = \ell$ and let $\mathcal{L}_p^f = \{\mathit{lab}(f) \mid f \in p\}$ denote the set of function labels in p. It is also assumed that every use of a function label in p occurs in $\mathcal{L}_p^f$, otherwise control may be transferred to an unknown function external to the program. Given a function $f \in p$, f can be referenced by the path $\langle p, f\rangle$ in order to distinguish it from an identical function occurring in another program.

3.1.2 Values and Expressions

A **value** is an item from the infinite set V composed of a set of integers bounded in the range $[-(2^{k-1}), 2^{k-1})$ for some choice of k, the set of references to functions defined within the program, references to intrinsics, and the constants `true`, `false`, and `unit`. A **register**

cv	$::=$		*constant values:*	v	$::=$		*values:*
	$\mid$	`true` $\mid$ `false`	booleans		$\mid$	cv	constant value
	$\mid$	`unit`	unit		$\mid$	`ref` f $\mid$ `ref` $\hat{f}$	function reference
	$\mid$	n	integers				
e	$::=$		*expressions:*	t	$::=$		*types:*
	$\mid$	cv	constant value		$\mid$	*bool*	booleans
	$\mid$	r	register		$\mid$	*unit*	unit
					$\mid$	*int*	integers
					$\mid$	$(\bar{t}) \to t$	function type

Figure 3.1: Syntax of values, expressions, and types.

is a virtual, symbolic storage location for a value chosen from an infinite set R. A register is associated with a value that can change over time. A register cannot be referenced until it has been associated with a value. A special *void* register, denoted by the symbol '$\cdot$', can be used for assignment but cannot be referenced. This register is used to absorb meaningless values so that an *instruction* can be executed for effect rather than for value. Together, the set of values and registers form the set of **expressions**. The syntax of expressions and the expression types is illustrated in Figure 3.1.

3.1.3 Block Components

The **components** of a block consist of block parameters, instructions, terminators, and block references. A component c in block b can be referenced by the path $\langle b, c\rangle$ so that equivalent components in different blocks are distinguishable. A component may be referenced without the path if the block in which it occurs is unambiguous. Let $c_1 <_b c_2$ denote the fact that component c_1 *occurs* before component c_2 in block b. A block's parameters occur first, followed by instructions, then followed by the terminator. All block parameters occur simultaneously, as does the terminator and its references.

The **definition** of a register r with respect to a function f is a component in f that

associates a (possibly unchanged) value with r. The **use** of a register r with respect to a function f is a component in f that reads a value with r. Explicit block parameters and instructions of the form $r \leftarrow v$ compose the set of definitions; implicit block parameters, instructions, terminators, and block references compose the set of uses. Let $reg(u)$ denote the set of registers used by a use u (e.g. $reg(r_1 \leftarrow \texttt{add}(r_2, r_3)) = \{r_2, r_3\}$). The uses and definitions of a register r in function f are denoted respectively by the sets $use_f(r)$ and $def_f(r)$. A *unique* definition of r is also denoted by $def_f(r)$ when static single assignment form (discussed in Section 4.1) is assumed. Let $use_b(r)$ and $def_b(r)$ denote, respectively, the set of uses and definitions of register r in block b. Let $use(b)$ and $def(b)$ denote, respectively, all uses and definitions occurring in block b. Subscript qualifiers may be dropped when the context resolves any ambiguity. Let $typeof_f(d)$ denote the *type* of the definition d. This definition will either resolve immediately by an operation that has a fixed type (e.g. arithmetic and function call operations), or will require determining the type of another defined register (e.g. a register-to-register copy).

3.1.3.1 Block Parameters and Implicit Parameters

A **block parameter** of a block b is a *typed* register of the form $(r : t)$, where t is the expected type of the value associated with r. The value of r is resolved once control enters the block during evaluation. Graphically, we will omit the type annotation of parameters for clarity when they are not necessary. Given a set of typed registers T and a set of registers R, we denote $\{(r : t) \in T \mid r \notin R\}$ as $T \ominus R$.

An **implicit parameter** of a block b is also a typed register of the form $(r : t)$. The set of implicit parameters for a block b are the set of registers that must be defined before b can be evaluated. Intuitively, this set is composed by the registers for which a use occurs without a preceding definition in the same block.

Implicit parameters are defined solely to aid type checking, discussed in Section 3.3. While the domain of the implicit parameters can be inferred syntactically, the proper type annotations for these registers cannot be as easily inferred without type parameters. The types must be supplied externally as part of the program's syntax. Graphically, we will omit implicit parameters for clarity when they are not necessary.

Definition 3.1.1 (Well-Formed Implicit Parameters)**.** A block b's implicit parameters are *well-formed* when $r \in \mathit{dom}(\mathit{implicit}(b))$ if and only if there exists a use $u \in \mathit{use}_b(r)$ that occurs in b such that $u <_b d$ for every $d \in \mathit{def}_b(r)$.

A **nonlocal implicit parameter** of a block b is also a typed register of the form $(r\colon t)$. The set of nonlocal implicit parameters of a block b, denoted $\mathit{nonlocal}_f(b)$, is the set of registers that must be defined before b *and its successors (recursively)*, with respect to a function f extending the control flow graph G, can be evaluated. The set of nonlocal implicit parameters can be calculated by the following without placing an additional annotation burden on the user.

$$\mathit{nonlocal}'_f(b) = \mathit{implicit}(b) \bigcup_{s \in \mathit{succ}_f(b)} \{(r\colon t) \in \mathit{nonlocal}'_f(s) \mid r \notin \mathit{def}(b)\}$$

$$\mathit{nonlocal}_f(b) = \{(r\colon t) \mid t \text{ is chosen arbitrarily from } \{t' \mid (r\colon t') \in \mathit{nonlocal}'_f(b)\}\}$$

The definition of $\mathit{nonlocal}_f(b)$ chooses pairs from $\mathit{nonlocal}'_f(b)$ such that every register is associated with a unique type. Annotation inconsistencies of this kind will fail during type checking as the same register will be used in multiple contexts requiring different types, all but one of which must fail.

This formulation is intuitive: the set nonlocal implicit parameters for a block b are b's implicit parameters and the nonlocal implicit parameters of b's successors that are not defined by b. If $(r\colon t) \in \mathit{nonlocal}_f(b)$, then there must be a block b' containing a use of r and some path $b \rightsquigarrow_G b'$ on which no definition of r occurs.

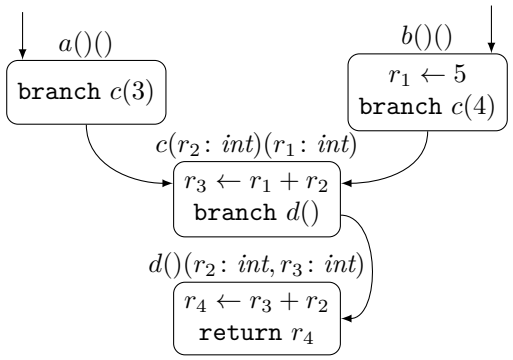


Figure 3.2: A graph annotated with implicit parameters (the second set of parameters).

	a	b	c	d
Local	$\emptyset$	$\emptyset$	$\{r_1 : int\}$	$\{r_2 : int, r_3 : int\}$
Nonlocal	$\{r_1 : int\}$	$\emptyset$	$\{r_1 : int\}$	$\{r_2 : int, r_3 : int\}$

Figure 3.3: Implicit parameters for the graph in Figure 3.2.

Figure 3.2 illustrates (annotated) implicit parameters on a relevant portion of a graph. In particular, block c has an implicit parameter for register r_1 and block d has an implicit parameters for registers r_2 and r_3. Figure 3.3 gives the calculated nonlocal implicit parameters. Of note, the requirement of register r_3 in block d is fulfilled by the assignment in block c, so the requirement does not flow any further along that path. Similarly, block r_1 is assigned by block b, that fulfills the requirement for the path to block c (but not the path from block a).

3.1.3.2 Instructions

An **instruction** contains an operation over $k > 0$ expressions. An operation yields a value that may then be assigned to a target register. If an operation does not yield a meaningful value or if the value resulting from an operation is unused by the program, the void register may be used as a target register. The syntax of instructions is illustrated in Figure 3.4.

Instruction operations include assignment (`move`), memory operations (`load` and `store`), and function reference and invocation (`addr` and `call`), as well as a standard set of logical,

I	$::=$	$r \leftarrow O$		*instruction*
O	$::=$			*operations:*
	$\mid$	$\texttt{move}(e)$		assignment
	$\mid$	$\texttt{load}\langle\!\langle t \rangle\!\rangle(e)$		read memory
	$\mid$	$\texttt{store}(e, e)$		write memory
	$\mid$	$\texttt{addr}(\ell)$	$\ell \in \mathcal{L}^f \cup \mathcal{L}^{\hat{f}}$	function address
	$\mid$	$\texttt{call}(e, \overline{e})$		function call
	$\mid$	$\oplus(e)$	$\oplus \in \{\texttt{pos}, \texttt{neg}, \texttt{not}\}$	negation
	$\mid$	$\oplus(e, e)$	$\oplus \in \{\texttt{add}, \texttt{sub}, \texttt{mul}, \texttt{div}\}$	arithmetic
	$\mid$	$\oplus(e, e)$	$\oplus \in \{\texttt{eq}, \texttt{ne}\}$	equality
	$\mid$	$\oplus(e, e)$	$\oplus \in \{\texttt{le}, \texttt{lt}, \texttt{ge}, \texttt{gt}\}$	comparison
	$\mid$	$\oplus(e, e)$	$\oplus \in \{\texttt{and}, \texttt{or}\}$	logical

Figure 3.4: The syntax of instructions.

arithmetic, and comparison operations. The `move` operation simply yields its evaluated argument without modification. The `load` operation evaluates its argument as an address in memory and yields the content of the memory cell at that address as a value of type t or fails with a memory error. Symmetrically, the `store` operation evaluates its first argument as an address in memory and evaluates its second argument as a value to place in the memory cell at that address. The `addr` operation yields a function reference from the label of a function that is defined in the same program. The `call` operation evaluates its first argument as a function reference, evaluates all remaining arguments as function arguments, then yields the return value of the function. The remaining operation are abstracted by the operator $\oplus$. Each operator has a rigid type, denoted $\oplus\colon t \to t'$, where t is the type of its arguments and t' is the type of its result. Graphically, we will represent the negation, arithmetic, equality, comparison, and logical operation using infix notation (e.g. $r_1 \leftarrow r_2 \times r_3$) when it increases readability.

$$
\begin{array}{rcll}
T & ::= & & \textit{terminators:} \\
 & | & \texttt{switch}\ e\ \overline{cv \mapsto ref}\ ref & \text{switch} \\
 & | & \texttt{return}\ e & \text{return} \\
 & & & \\
ref & ::= & \ell(\overline{e}) \qquad \ell \in \mathcal{L}^b & \text{block reference}
\end{array}
$$

Figure 3.5: The syntax of terminators.

3.1.3.3 Terminators

A **terminator** determines the flow of control between basic blocks. A terminator may contain zero or more **block references** of the form $\ell(\overline{e_i})$ where $\ell \in \mathcal{L}^b$ refers to the label of the target block $b = \mathit{block}_G(\ell)$, and the value of the expression e_i becomes the value of the i-th parameter of b when control is transferred to b. The block references of a block b are denoted by the set $\mathit{ref}(b)$. The multiset of labels in all block references of b are denoted by the set $\mathit{target}(b)$. The syntax of terminators is illustrated in Figure 3.5.

The `switch` terminator transfers control conditionally to one of several blocks, depending on the runtime value of e. If the terminator contains a switch case $v_i \mapsto \mathit{ref}_i$ such that evaluated value of e equals v_i, then control is transferred to the block referred to by ref_i. To ensure deterministic evaluation, the first switch case to occur lexically is chosen when multiple switch cases match. Switch cases with a value of `unit` are considered dead and are never matched during evaluation. Otherwise, control is transferred to the block referred to by the *default* reference. Control can be transferred to a block unconditionally by specifying only a default branch target. Graphically, we abbreviate unconditional branches by the following shorthand.

$$\texttt{branch}\ \mathit{ref} \equiv \texttt{switch}\ \texttt{unit}\ \mathit{ref}$$

The `return` terminator exits the current function, yielding the evaluated value of e to the caller.

3.2 Semantics

An internal representation is useless if it has no defined operational meaning. Section 3.2.1 details function cloning (which will later be shown necessary for evaluation). Section 3.2.2 and Section 3.2.3 defines a set of environments and a small-step operational semantics that are used to evaluate a function.

3.2.1 Function Cloning

First, we detail the method of creating an *isomorphic* clone of a function. As we discuss in Section 3.2.3, this will be necessary to distinguish the same register between recursive invocations of the same function. A **clone** f' of function f is a duplication of f's control flow graph where every occurrence of each register r is replaced by some fresh register r' (for all r). The label of the function, the entry block, and block within the function are stable, as is the function's return type annotation.

Let $\mathcal{C}(f)$ be the (infinite) set of functions that are isomorphic clones of f. By convention, $f \in \mathcal{C}(f)$ and we assume for every $f', f'' \in \mathcal{C}(f)$, f' and f'' have strictly disjoint registers. Given a block $b \in body(f)$, let $\mathcal{C}(b) = \{block_{f'}(lab(b)) \mid f' \in \mathcal{C}(f)\}$ be the (infinite) set of blocks symetric to b in all clones of f. For notational convenience, we extend this definition to sets of blocks as follows.

$$\mathcal{C}(B) = \bigcup_{b \in B} \mathcal{C}(b)$$

3.2.2 Environments

In this section, we define operational semantic rules that evaluate an instruction stream of a function within a program. The terms and environments over which evaluation occurs are illustrated in Figure 3.6.

s	::=		*stream:*	$\hat{s}$	::=		*return context:*
	$\mid$	ϵ	empty		$\mid$	ϵ	top-level
	$\mid$	I, s	instructions		$\mid$	$\langle f, b, r, s \rangle$	caller stream
	$\mid$	$T, \hat{s}$	terminator				
γ	::=		*register environment:*	μ	::=		*memory environment:*
	$\mid$	$\varnothing$	empty		$\mid$	$\varnothing$	empty
	$\mid$	$\gamma[r \mapsto v]$	non-empty		$\mid$	$\mu[n \mapsto v]$	non-empty
ψ	::=		*effect:*	Ψ	::=		*effects list:*
	$\mid$	$\texttt{halt}(v)$	termination		$\mid$	ϵ	empty
	$\mid$	$\texttt{halt}(\textbf{ex}(\text{err}))$	abnormal termination		$\mid$	Ψ, ψ	non-empty
	$\mid$	$\hat{f}(\overline{e})$	intrinsic call				

Figure 3.6: Syntax of terms, environments, nondeterminsm state, and effects.

A **stream** s is a recursively constructed sequence of instructions that ends in a terminator and a *return context*. A **return context** quadruple $\langle f, b, r, s \rangle$ represents a frame in the call stack such that f is the calling function, b is the block that originated the call, r is the target register for the result of the called function, and s is the remaining unevaluated stream of f. The empty return context ϵ denotes execution of a top-level function. Streams can be nested arbitrarily deeply.

The concatenation of a stream and a return context denoted $s, \hat{s}$ replaces the return context of s with $\hat{s}$. The stream of block b can be represented as an evaluation term with the empty return context, as follows.

$$\langle I_1, \ldots, I_n, T \rangle \;\equiv\; I_1, \ldots, I_n, T, \epsilon$$

The **register environment**, denoted γ, is a store that partially maps registers to values: $\gamma \colon R \rightharpoonup V$. We define evaluation of expressions with respect to a register environment as follows:

$$\gamma(e) = \begin{cases} v & \text{if } e = v \in V \\ v & \text{if } e = r \in R \setminus \{\,\cdot\,\} \wedge (r \mapsto v) \in \gamma. \end{cases}$$

Assignment to a register with respect to a register environment γ is denoted $\gamma[r \mapsto v]$, which associates the register r with the value v. This destroys any previous type association for the register r.

The **memory environment**, denoted μ, is a store that maps an infinite address space to bits: $\mu \colon \mathbb{N} \to \{0, 1\}$. The empty memory environment $\varnothing$ maps all addresses to zero. The memory environment does not contain holes that are unaddressable. While the memory environment is infinite, only the first $2^{k-1}-1$ bits are addressable when the program's integer values are encoded with k bits.

Every value $v \in V$ of type t can be described by a bitstring of length w that is determined by the **width** of the values in t. This value, denoted by $\mathit{width}(t)$, is defined with respect to a particular program, but can always be artificially increased in order to match the widths of another program. Notably, if $|\mathit{func}(p)| = k$, then the width of function types with respect to program p must be at least $\lceil \log_2(k) \rceil$.

A bitstring of length w occupying addresses $[n, n + w)$ in the memory environment μ is denoted $\mu(n, w)$. It is possible but not guaranteed that a bitstring of length w is interpretable as a value of type t where $\mathit{width}(t) = w$. Mutation of the memory environment is denoted $\mu[n \mapsto v]$, which overwrites contents of the memory environment at addresses $[n, n+\mathit{width}(t))$ with the bitstring interpretation of value v with type t.

The **nondeterminism state**, denoted ν, is an unbounded, opaque stream of bits that can be consumed by the invocation of an intrinsic. This extra evaluation parameter allows total determinism over the set of functions and intrinsics while still allowing a way to model non-deterministic evaluation.

An **effects list**, denoted Ψ, is an ordered sequence of observable operations made during evaluation. An effect is created for the invocation of an intrinsic, denoted $\hat{f}(\overline{v_i})$ where v_i is a value passed to the intrinsic, and for the final result of a program, denoted as either

`halt`(v) or `halt`(**ex**(err)). A value v is produced when a program terminates successfully. An abnormal exit value **ex**(ex) is produced in a well-defined program (such that evaluation rules do not "get stuck") when a necessary runtime condition does not hold (e.g. a zero denominator or a failed bitstring interpretation). A program that terminates will produce exactly one `halt` effect at the tail of the evaluation's effect list. A non-terminating program will produce no such effect.

3.2.3 Evaluation

The operational semantics is described in terms of a small-step evaluation and is illustrated in Figure 3.7, Figure 3.8, Figure 3.9, Figure 3.10, and Figure 3.11. Term evaluation is defined by a relation of the form $(\langle p, f, b\rangle \mid \gamma \mid \mu \mid \nu \mid \Psi; s) \to (\langle p, f', b'\rangle \mid \nu' \mid \gamma' \mid \mu' \mid \Psi'; s')$, where $\langle p, f, b\rangle$ is the path to the block currently being evaluated, γ and μ are the current register and memory environments, ν is the current nondeterminism state, and Ψ is the sequence of effects that have already been produced. In the resulting tuple, $\langle p, f', b'\rangle$ is the path to the block to which control has transferred, γ' and μ' are the register and memory environments that were (perhaps) mutated by the last evaluated operation, ν' is the resulting nondeterminism state, Ψ' is a list of effects (perhaps) with a fresh tail element, and s' is the sequence of instructions remaining in the evaluation. A reference to the block in which an instruction is being evaluated is not necessary for operational semantics. The rules presented in this chapter do not read the values of b or b' in a meaningful way. However, its presence greatly aids in the proofs of semantic preservation and are thus included.

Figure 3.7 illustrates evaluation of streams beginning with an instruction (excluding `call`) when evaluation does not produce an abnormal exit. Evaluating an instruction stream of the form $(r \leftarrow O, s)$ first evaluates the operation in the head of the list that may modify the memory environment, associates the result value with the register r, then recursively

E-Inst

$$\frac{(p \mid \gamma \mid \mu;\ O) \to (\mu';\ v)}{(\langle p, f, b\rangle \mid \gamma \mid \mu \mid \nu \mid \Psi;\ r \leftarrow O, s) \to (\langle p, f, b\rangle \mid \gamma[r \mapsto v] \mid \nu \mid \mu' \mid \Psi;\ s)}$$

E-Operator

$$\frac{\oplus \neq \texttt{div} \vee \gamma(e_2) \neq 0 \qquad [\![\oplus(\overline{\gamma(e_i)})]\!] = v}{(p \mid \gamma \mid \mu;\ \ \oplus\,(\overline{e_i})) \to (\mu;\ v)}$$

E-Move

$$\frac{\gamma(e) = v}{(p \mid \gamma \mid \mu;\ \texttt{move}(e)) \to (\mu;\ v)}$$

E-Addr

$$\frac{\mathit{func}_p(\ell) = f}{(p \mid \gamma \mid \mu;\ \texttt{addr}(\ell)) \to (\mu;\ \texttt{ref}\ f)}$$

E-Addr-Intrinsic

$$\frac{\mathit{lab}(\hat{f}) = \ell}{(p \mid \gamma \mid \mu;\ \texttt{addr}(\ell)) \to (\mu;\ \texttt{ref}\ \hat{f})}$$

E-Load

$$\frac{\gamma(e) = n \qquad \mathit{width}(t) = w \qquad \mu(n, w) = v \text{ as } t}{(p \mid \gamma \mid \mu;\ \texttt{load«}t\texttt{»}(e)) \to (\mu;\ v)}$$

E-Store

$$\frac{\gamma(e_1) = n \qquad \gamma(e_2) = v \qquad n \geq 0}{(p \mid \gamma \mid \mu;\ \texttt{store}(e_1, e_2)) \to (\mu[n \mapsto v];\ \texttt{unit})}$$

Figure 3.7: Evaluation rules for instructions excluding `call`.

evaluates the tail of the stream. Assignment and sequencing is implicit in rule E-Inst, which evaluates the operation as an independent step.

The remaining rules in Figure 3.7 use a relation of the form $(p \mid \gamma \mid \mu; O) \to (\mu'; v)$, where μ' is the state of memory after evaluating O, and v is the resulting value. Operations themselves may read the register environment but do not alter it. These rules should be unsurprising. Rule E-Addr and rule E-Addr-Intrinsic yields a function reference from a function label and an intrinsic label, respectively. Rule E-Load reads w bits from the memory environment starting at address n and yields the loaded bitstring interpreted as type t. Rule E-Store ensures that the address n is a valid address before modifying the

E-Div-Abnormal

$$\frac{\gamma(e_2) = 0}{(\langle p, f, b\rangle \mid \gamma \mid \mu \mid \nu \mid \Psi;\ r \leftarrow \texttt{div}(e_1, e_2), s) \rightarrow (\langle p, f, b\rangle \mid \gamma \mid \mu \mid \nu \mid \Psi,\ \texttt{halt}(\textbf{ex}(\text{math err}));\ \epsilon)}$$

E-Load-Abnormal

$$\frac{\gamma(e) = n \qquad \mathit{width}(t) = w \qquad n < 0 \vee \mu(n, w) \text{ is not interpretable as } t}{(\langle p, f, b\rangle \mid \gamma \mid \nu \mid \mu \mid \Psi;\ r \leftarrow \texttt{load}\text{«}t\text{»}(e), s) \rightarrow (\langle p, f, b\rangle \mid \gamma \mid \mu \mid \nu \mid \Psi,\ \texttt{halt}(\textbf{ex}(\text{mem err}));\ \epsilon)}$$

E-Store-Abnormal

$$\frac{\gamma(e_1) = n \qquad n < 0}{(\langle p, f, b\rangle \mid \gamma \mid \mu \mid \nu \mid \Psi;\ r \leftarrow \texttt{store}(e_1, e_2), s) \rightarrow (\langle p, f, b\rangle \mid \gamma \mid \mu \mid \nu \mid \Psi,\ \texttt{halt}(\textbf{ex}(\text{mem err}));\ \epsilon)}$$

Figure 3.8: Evaluation rules for instructions evaluating abnormally.

memory environment and yields the value `unit` as the `store` instruction is intended for side-effect. The operations abstracted by $\oplus$ are evaluated by rule E-Operator. This rule uses safe evaluation notation $[\![\oplus(\overline{e})]\!]$ that is expected to return a value when the correct number and types of arguments are supplied. To ensure this is the case, rule E-Operator ensures that the denominator value is non-zero when the operator is division (the only operator that can fail). Other arithmetic operations are assumed to overflow deterministically if the result cannot be encoded in the program's integer range.

Figure 3.8 illustrates operator rules resulting in abnormal termination. Rule E-Div-Abnormal is applied when the denominator of a division operation evaluates to zero. Rule E-Store-Abnormal is applied when the target address n is not a valid address. Similarly, E-Load-Abnormal is applied when either the base address n is not a valid address or the bitstring loaded from memory cannot be interpreted as a value of type t. All three rules terminate evaluation immediately by producing an abnormal exit value.

Figure 3.9 illustrates rules for invoking intrinsics. The invocation of an intrinsic $\hat{f}$ takes as input a memory environment μ, a nondeterminism state ν, and a sequence of values.

E-Intrinsic

$$\frac{\gamma(e_0) = \texttt{ref}\ \hat{f} \qquad \forall i \cdot \gamma(e_i) = v_i \qquad \hat{f}(\mu, \nu, \overline{v_i}) = (\mu', \nu', v)}{(\langle p, f, b\rangle \mid \gamma \mid \mu \mid \nu \mid \Psi;\ r \leftarrow \texttt{call}(e_0, \overline{e_i}), s) \rightarrow (\langle p, f, b\rangle \mid \gamma[r \mapsto v] \mid \mu' \mid \nu' \mid (\Psi,\ \hat{f}(\overline{v_i}));\ s)}$$

E-Intrinsic-Abnormal

$$\frac{\gamma(e_0) = \texttt{ref}\ \hat{f} \qquad \forall i \cdot \gamma(e_i) = v_i \qquad \hat{f}(\mu, \nu, \overline{v_i}) = \mathbf{ex}(\text{err})}{(\langle p, f, b\rangle \mid \gamma \mid \mu \mid \nu \mid \Psi;\ r \leftarrow \texttt{call}(e_0, \overline{e_i}), s) \rightarrow (\langle p, f, b\rangle \mid \gamma \mid \mu \mid \nu \mid (\Psi,\ \hat{f}(\overline{v_i}),\ \texttt{halt}(\mathbf{ex}(\text{err})));\ \epsilon)}$$

Figure 3.9: Evaluation rules for the invocation of an intrinsic.

On success, the invocation produces a triple (μ', ν', v) consisting of a modified nondeterminism state, a modified memory environment, and a single value. On failure, the invocation produces an abnormal exit value on invocation failure. Any attempt to invoke an intrinsic function adds an effect to the tail of the resulting effect list. For ease of proving equivalent semantics after transformation, we restrict the behavior of intrinsics when operating on function type values. If values `ref` f and `ref` f' reduce to the same bit pattern (with respect to any program), then this value may not meaningfully influence the intrinsic's output.

Figure 3.10 illustrates rules that dictate control flow between blocks of the same function. The auxiliary rule E-Ref partially evaluates a block reference using a relation of the form $(f \mid \gamma; \mathit{ref}) \rightarrow (b \mid \gamma'; s)$, where f is a function containing possible target blocks, s is the stream of the block referenced by *ref*, b is the target block, and γ' is the register environment γ modified such that each target block parameter is associated with the corresponding argument in the block reference.

Rule E-Switch replaces the head of the stream with the stream of a target block chosen dynamically via rule E-Case or rule E-Case-Default. This leaves the return context unchanged as evaluation has not moved out of the current function, but does change the block currently being evaluated. If the evaluated switch expression matches some v_i, then

E-Switch

$$\frac{(\gamma(e);\ \overline{cv_i \mapsto ref_i}\ ref_d) \to ref \qquad (f \mid \gamma; ref) \to (b' \mid \gamma';\ s')}{(\langle p, f, b\rangle \mid \gamma \mid \mu \mid \nu \mid \Psi;\ \texttt{switch}\ e\ \overline{cv_i \mapsto ref_i}\ ref_d, \hat{s}) \to (\langle p, f, b'\rangle \mid \gamma' \mid \mu \mid \nu \mid \Psi;\ s', \hat{s})}$$

E-Case

$$\frac{cv_i = v \wedge \forall j < i \cdot cv_j \neq v \vee cv_j = \texttt{unit}}{(v;\ \overline{cv_i \mapsto ref_i}\ ref_d) \to ref_i}$$

E-Case-Default

$$\frac{\forall i \cdot cv_i \neq v \vee v = \texttt{unit}}{(v;\ \overline{cv_i \mapsto ref_i}\ ref_d) \to ref_d}$$

E-Ref

$$\frac{block_f(\ell) = b \qquad stream(b) = s \qquad \forall i \cdot \gamma(e_i) = v_i \qquad dom(param(b)) = \langle \overline{r_i} \rangle}{(f \mid \gamma;\ \ell(\overline{e_i})) \to (b \mid \gamma[\overline{r_i \mapsto v_i}];\ s)}$$

Figure 3.10: Evaluation rules for the `switch` terminator.

the associated block reference is used. Otherwise, the default block reference is used.

Figure 3.11 illustrates rules that dictate inter-procedural control flow. Rule E-Call determines the target function by address and replaces the head of the stream with the stream of the target function's entry block. The remainder of the current stream is stored in a new return context object along with the target register r, both of which are needed by the evaluation of the call function's return terminator.

This rule contains the additional obligation of ensuring that b is the entry block of the fresh clone of the target function. Specifically, the clone must be minimally fresh with respect to this evaluation. To illustrate why this is necessary, consider the function *double* with parameters ($r_1 \colon$ *int*) and an entry block stream equivalent to ($r_2 \leftarrow r_1 + r_1,\ \texttt{return}\ r_2$). Now, suppose we evaluate the following stream with the register environment $[r_2 \mapsto 12]$ and the empty memory environment.

$$r_1 \leftarrow \texttt{addr}(\mathit{double}),\quad r_3 \leftarrow \texttt{call}(r_1, r_2),\quad r_3 \leftarrow r_3 - r_2,\quad \texttt{return}\ r_3$$

E-Call

$$\frac{\gamma(e_0) = \texttt{ref } f' \qquad f'' \in \mathcal{C}(f') \qquad \mathit{entry}(f'') = b' \qquad \mathit{lab}(b') = \ell \qquad (f \mid \gamma;\ \ell(\overline{e_i})) \to (b' \mid \gamma';\ s')}{(\langle p, f, b\rangle \mid \gamma \mid \mu \mid \nu;\ r \leftarrow \texttt{call}(e_0, \overline{e_i}), s) \to (\langle p, f'', b'\rangle \mid \gamma' \mid \mu \mid \nu;\ s', \langle f, b, r, s\rangle)}$$

E-Return

$$\frac{\gamma(e) = v}{(\langle p, f, b\rangle \mid \gamma \mid \mu \mid \nu \mid \Psi;\ \texttt{return } e, \langle f', b', r, s\rangle) \to (\langle p, f', b'\rangle \mid \gamma[r \mapsto v] \mid \mu \mid \nu \mid \Psi;\ s)}$$

E-Return-Last

$$\frac{\gamma(e) = v}{(\langle p, f, b\rangle \mid \gamma \mid \mu \mid \nu \mid \Psi;\ \texttt{return } e, \epsilon) \to (\langle p, f, b\rangle \mid \gamma \mid \mu \mid \nu \mid \Psi,\ \texttt{halt}(v);\ \epsilon)}$$

Figure 3.11: Evaluation rules for the `call` instruction and `return` terminator.

If we do not rename the registers in *double*, then evaluation reduces to the value 0 when we expect the value 12. This is due to the evaluation of *double* overwriting the result of r_2, which is still referenced by the remaining stream of the calling function. To resolve this issue, we ensure that each function *invocation* operates in an independent register space. This must be done at evaluation time in order to resolve the issue when recursive function invocations may be present.

Rule E-Return is the dual of the `call` instruction and uses the return context $\langle f', b', r, s\rangle$ constructed by the immediate caller. This sequence reduces to the caller's remaining instruction stream and modifies the register environment to associate the register r with the return value. Rule E-Return-Last is applied when no caller exists and is the only rule that produces a terminal value.

Finally, we provide a means of entry into evaluation of a program given a chosen start function $f \in p$. This rule is illustrated in Figure 3.12. Rule E-Entry simply converts a

E-Entry

$$\frac{f \in p \qquad \mathit{entry}(f) = b \qquad \mathit{lab}(b) = \ell \qquad (f \mid \gamma;\ \ell(\overline{v_i})) \to (b \mid \gamma';\ s')}{(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_i})) \to (\langle p, f, b\rangle \mid \gamma' \mid \mu \mid \nu \mid \Psi;\ s')}$$

Figure 3.12: Evaluation rule for invoking a program function externally.

$$\Gamma \quad ::= \quad \varnothing \mid \Gamma, (r : t) \quad \textit{register environment typing}$$

Figure 3.13: Typing environment.

function object and a list of positional arguments into the correct register context and stream necessary to begin evaluation. Such a rule allows the invocation of a function $f(e_1, e_2, e_3)$ and invocation of the same function after a transformation, $f'(e_1, e_2, e_3)$, to begin evaluation in a syntactically similar way. This allows us to elide the mapping of the entry register context in the case when the entry block parameters of f and f' have differing names.

3.3 Type System

In this section, we detail a type system over the syntax described earlier in the chapter. The **register environment typing** Γ, given in Figure 3.13, tracks the types of values associated with registers as an ordered sequence of pairs $(r : t)$, signifying that register r contains a value of type t. Re-binding the type of a register makes the previous type inaccessible. Typing rules for expressions are illustrated in Figure 3.14. The content of memory is not tracked statically, therefore no memory environment typing exists.

Typing is attacked from two fronts. First, we type instruction streams until we reach a terminator. These streams may be partial. That is, a result from performing a sequence of reductions starting from a block's initial instruction stream. At this point, we only ensure

$$\textsc{T-Bool}\ \frac{v \in \{\texttt{true}, \texttt{false}\}}{\Gamma \vdash v \colon \mathit{bool}} \qquad \textsc{T-Unit}\ \frac{}{\Gamma \vdash \texttt{unit} \colon \mathit{unit}} \qquad \textsc{T-Int}\ \frac{}{\Gamma \vdash n \colon \mathit{int}} \qquad \textsc{T-Ref}\ \frac{f \colon (\overline{t_i}) \to t}{\Gamma \vdash \texttt{ref}\ f \colon (\overline{t_i}) \to t}$$

$$\textsc{T-Ref-Intrinsic}\ \frac{\hat{f} \colon (\overline{t_i}) \to t}{\Gamma \vdash \texttt{ref}\ \hat{f} \colon (\overline{t_i}) \to t} \qquad \textsc{T-Reg}\ \frac{r \neq \cdot \qquad (r \colon t) \in \Gamma}{\Gamma \vdash r \colon t}$$

Figure 3.14: Typing judgments for expressions.

that the terminator is well-typed with respect to the *declared* types of the target block. The typing judgments for a stream take the form $\Gamma \mid t \vdash s$, where Γ is the register environment typing and t is the expected return type of the stream up to a terminator. It is important to note that type checking the stream does not follow block references, as doing so would essentially evaluate the program. Second, we type each block independently to ensure that the declared types match the block's stream.

Typing Streams Figure 3.15 illustrates rules for typing a stream that begins with an instruction. Rule T-Inst types a stream of the form $(r \leftarrow O, s)$ by determining the result type t of the operator O, then typing the remaining stream with t bound to r in the register environment typing. The remaining rules in Figure 3.15 type operations. Operations may be side-effecting but do not modify the register environment, therefore they cannot modify the register environment typing.

Figure 3.16 illustrates rules for typing a stream that consists of only a terminator. Rule T-Term types a stream of the form $(T, \hat{s})$ by ensuring that both the terminator T and the return context $\hat{s}$ are well-typed. Terminators are typed with respect to a register environment typing and the declared return type t of the function. Rule T-Switch ensures that the switch expression is well-typed and all block references are well-typed. Rule T-Return

$$\textsc{T-Inst}\ \frac{(p\ f \mid \Gamma) \vdash O : t' \qquad (p\ f \mid \Gamma, (r : t')) \vdash s}{(p\ f \mid \Gamma) \vdash r \leftarrow O, s} \qquad \textsc{T-Move}\ \frac{\Gamma \vdash e : t}{(p\ f \mid \Gamma) \vdash \texttt{move}(e) : t}$$

$$\textsc{T-Addr}\ \frac{\mathit{func}_p(\ell) : t \vee (\mathit{lab}(\hat{f}) = \ell \wedge \hat{f} : t)}{(p\ f \mid \Gamma) \vdash \texttt{addr}(\ell) : t} \qquad \textsc{T-Load}\ \frac{\Gamma \vdash e : \mathit{int}}{(p\ f \mid \Gamma) \vdash \texttt{load«}t\texttt{»}(e) : t}$$

$$\textsc{T-Store}\ \frac{\Gamma \vdash e_1 : \mathit{int} \qquad \Gamma \vdash e_2 : t}{(p\ f \mid \Gamma) \vdash \texttt{store}(e_1, e_2) : \mathit{unit}} \qquad \textsc{T-Call}\ \frac{\Gamma \vdash e_0 : (\overline{t_i}) \to t \qquad \forall i \cdot \Gamma \vdash e_i : t_i}{(p\ f \mid \Gamma) \vdash \texttt{call}(e_0, \overline{e_i}) : t}$$

$$\textsc{T-Operator}\ \frac{\oplus : t \to t' \qquad \forall i \cdot \Gamma \vdash e_i : t}{(p\ f \mid \Gamma) \vdash \oplus(\overline{e_i}) : t'}$$

Figure 3.15: Typing judgments for instruction streams.

ensures the value being returned matches the declared type of the containing function.

A block reference is typed by rule T-Ref, which ensures that the number and type of arguments to the block match the supplied argument and that the set of nonlocal implicit block parameters already exist in the register environment typing. This ensures that any register reference within the target stream is defined before evaluation begins on that stream. For now, we make the assumption the target block's declared parameters ensure a well-typed stream.

Rule T-Context and rule T-Empty ensure a return context $\hat{s}$ is well-typed with respect to a register environment typing and the type of the called function's return value t. A non-empty return context $\langle f, b, r, s \rangle$ is well-typed if s is well-typed when r has type t in the register environment typing. An empty return context is trivially well-typed.

$$\textsc{T-Term}\ \frac{(p\ f \mid \Gamma) \vdash T \qquad (p\ f \mid \Gamma) \vdash \hat{s}}{(p\ f \mid \Gamma) \vdash T, \hat{s}} \qquad \textsc{T-Switch}\ \frac{\Gamma \vdash e\colon t' \qquad \forall i \cdot (p\ f \mid \Gamma) \vdash \mathit{ref}_i}{(p\ f \mid \Gamma) \vdash \texttt{switch}\ e\ \overline{\mathit{cv}_i \mapsto \mathit{ref}_i}\ \mathit{ref}_d}$$

$$\textsc{T-Return}\ \frac{\Gamma \vdash e\colon t}{(p\ f \mid \Gamma) \vdash \texttt{return}\ e} \qquad \textsc{T-Context}\ \frac{f\colon (\overline{t_i}) \to t \qquad (p\ f' \mid \Gamma, (r\colon t)) \vdash s}{(p\ f \mid \Gamma) \vdash \langle f', b, r, s\rangle} \qquad \textsc{T-Empty}\ \frac{}{(p\ f \mid \Gamma) \vdash \epsilon}$$

$$\textsc{T-Ref}\ \frac{\mathit{block}_f(\ell) = b \qquad \mathit{nonlocal}_f(b) \subseteq \Gamma \qquad \forall (r_i\colon t_i) \in \mathit{param}(b) \cdot \Gamma \vdash e_i\colon t_i}{(p\ f \mid \Gamma) \vdash \ell(\overline{e_i})}$$

Figure 3.16: Typing judgments for terminators.

Rule T-Empty also provides a means to type a point of termination after a top-level function has returned and no instructions remain to be evaluated.

Typing Blocks On the second front, we ensure a block's stream is well-typed when block and nonlocal implicit parameters are assumed to exist with their declared types. Figure 3.17 shows the rules for well-typed blocks and functions. A block is well-typed, according to rule T-Block, if its implicit parameters are well-formed and its stream is well-typed provided a register environment typing that includes both the block parameters and nonlocal implicit parameters. A function is well-typed, according to rule T-Func, if all its blocks are well-typed and the entry block requires no registers to be defined prior to entry to the function. If this were not the case, then the function would not be able to be evaluated properly with only its arguments. It would require some magical global state to be initialized before evaluation began, which is unsupported (and unwanted) in this internal representation.

Additionally, rule T-Entry ensures that a set of arguments can be applied to a well-typed function if the types of the argument values match the number and types of the

T-Block

$$\frac{\mathit{implicit}(b) \text{ is well-formed} \qquad \mathit{stream}(b) = s \qquad (p\ f \mid (\mathit{param}(b), \mathit{nonlocal}_f(b))) \vdash s}{p\ f \mid \varnothing \vdash b}$$

T-Func

$$\frac{\mathit{nonlocal}_f(\mathit{entry}(f)) = \emptyset \qquad \forall b \in B \cdot (p\ f \mid \emptyset \vdash b)}{p \mid f \text{ is well-typed}}$$

T-Entry

$$\frac{p \mid f \text{ is well-typed} \qquad f\colon (\overline{t_i}) \to t \qquad \vdash v_i\colon t_i}{p \mid \varnothing \vdash f(\overline{v_i})}$$

Figure 3.17: Well-typed properties for blocks and functions.

function's arguments.

Appendix

3.A Soundness

In this section, we prove the soundness of the type system via a pair of progress and preservation theorems [66]. We show that a well-typed non-empty stream can make a step of evaluation (Theorem 3.A.9) and a step of evaluation yields a well-typed stream (Theorem 3.A.10). Interleaving these theorems concludes that a well-typed stream can be evaluated until the termination of an instruction stream, or forever on non-termination. First, we give the following definitions, which are necessary to state the progress and preservation theorems precisely.

Definition 3.A.1. A register environment γ is *well-typed* with respect to a register environment typing Γ, denoted $\Gamma \vdash \gamma$, if $dom(\gamma) = dom(\Gamma)$ and $\forall r \in dom(\gamma) \cdot \Gamma \vdash \gamma(r) : \Gamma(r)$.
NB: $dom(\gamma)$ includes only registers, not constant values.

Definition 3.A.2. A stream s and a return context $\hat{s}$ are *independent* if no register occurring in s occurs in $\hat{s}$ and the stream of every block reachable from the terminator of s is also independent from $\hat{s}$. A stream s and a return context $\hat{s}$ are *recursively independent* if s and $\hat{s}$ are independent and the stream of $\hat{s}$ (when not empty) is recursively independent from its own return context.

Definition 3.A.2 leads immediately to the following two corollaries, which state that a

stream and its return context will remain independent as evaluation progresses.

Corollary 3.A.3. If $s = I, s'$ and $\hat{s}$ are (recursively) independent, then s' and $\hat{s}$ are also (recursively) independent.

Corollary 3.A.4. Let $\ell(\overline{e_i})$ be a reference to block b occurring in T. If T and $\hat{s}$ are (recursively) independent, then $\mathit{stream}(b)$ and $\hat{s}$ are also (recursively) independent.

Next, we define the following four lemmas, which respectively state that additional (unreferenced) register-type mappings can be added to a register environment typing without changing the result of type checking, that a register environment and its typing can grow symmetrically, that a well-typed expression can be evaluated, and that a well-typed stream can be embedded in an independent stack context.

Lemma 3.A.5. If $p\ f \mid \Gamma \vdash s$, then $p\ f \mid \Gamma', \Gamma \vdash s$ for any Γ'.

Proof. Trivial as $\Gamma \subseteq \Gamma', \Gamma$. □

Lemma 3.A.6. If $\Gamma \vdash \gamma$ and $v : t$, then $\Gamma, (r : t) \vdash \gamma[r \mapsto v]$.

Proof. Let $\Gamma' = \Gamma, (r : t)$ and let $\mathrm{s}\gamma' = (\gamma[r \mapsto v])$. Then, as $\mathit{dom}(\Gamma) = \mathit{dom}(\gamma)$, it must be the case that $\mathit{dom}(\Gamma') = \mathit{dom}(\gamma')$. Any previous value and type associated with r is overwritten. Therefore $\Gamma' \vdash \gamma'(r) : t$ and all registers $r' \neq r \in \gamma$ remain well-typed with respect to Γ. □

Lemma 3.A.7. If $\Gamma \vdash e : t$ and $\Gamma \vdash \gamma$, then $\gamma(e)$ is well-defined.

Proof. The value $\gamma(e)$ is well-defined if $e \neq \cdot$ and $e \in \mathit{dom}(\gamma)$ by inversion of rule T-Reg. Otherwise, if e is a constant value, then $\gamma(e)$ is well-defined by inversion of rules T-Int, T-Bool, or T-Unit. □

Lemma 3.A.8. If s and $\langle f', b, r, s' \rangle$ are recursively independent, $(p\ f \mid \Gamma) \vdash s$, $\mathit{ret}(f) = t$, and $(p\ f' \mid \Gamma', (r : t)) \vdash s'$ for some $\Gamma' \subseteq \Gamma$, then $(p\ f \mid \Gamma) \vdash s, \langle f', b, r, s' \rangle$.

Proof. As $(p\ f' \mid \Gamma', (r\colon t)) \vdash s'$ and $\Gamma' \subseteq \Gamma$, then $(p\ f' \mid \Gamma, (r\colon t)) \vdash s'$ by application of Lemma 3.A.5. Then, rule T-Context applies during the application of T-Term at the tail of s, which is the only step that differs from the typing of $(p\ f \mid \Gamma) \vdash s, \epsilon$. □

Now, progress and preservation can be stated (and proved).

Theorem 3.A.9 (Progress)**.** If $(p\ f \mid \Gamma) \vdash s$ and all functions of p are well-typed, then for any register environment γ such that $\Gamma \vdash \gamma$, the stream s can evaluate one step.

More formally, for any memory environment μ, nondeterminism state ν, and effects list Ψ, the step of evaluation $(\langle p, f, b\rangle \mid \gamma \mid \mu \mid \nu \mid \Psi;\ s) \to (\langle p, f', b'\rangle \mid \gamma' \mid \mu' \mid \nu' \mid \Psi';\ s')$ is well-defined by the rules of Section 3.2 for some block b, block b', function f', register context γ', memory context μ', nondeterminism state ν', effects list Ψ', and stream s'.

Proof. We prove by case analysis on s. For brevity, it is safe to assume that $f' = f$, $b' = b$, $\mu' = \mu$, $\nu' = \nu$, and $\Psi' = \Psi$ unless otherwise stated.

Case ($r \leftarrow \texttt{move}(e), s'$). Evaluation of $\gamma(e)$ is well-defined by inversion of rules T-Inst and T-Move and application of Lemma 3.A.7. Then $\gamma' = \gamma[r \mapsto \gamma(e)]$ by application of rules E-Inst and E-Move.

Case ($r \leftarrow \oplus(\overline{e_i}), s''$). Syntax ensures the correct arity of arguments for each operator. Inversion of rules T-Inst and T-Operator and application of Lemma 3.A.7 ensure the evaluation of each $\gamma(e_i)$ is well-defined and the type of each argument matches the declared parameter type of the operator. Then, the result v of the operator is defined for all cases *except* division by zero by application of rules E-Inst and E-Operator where $s' = s''$ and $\gamma' = \gamma[r \mapsto v]$. For the division by zero case, $s'' = \epsilon$ and $\Psi' = \Psi, \texttt{halt}(\textbf{ex}(\text{math err}))$ by application of rule E-Div-Abnormal.

Case ($r \leftarrow \texttt{addr}(\ell), s'$). By inversion of rules T-Inst and T-Addr, ℓ must either be the label of a function $f'' \in p$ or an intrinsic $\hat{f}$ with the function type t. In the former case, $\gamma' = \gamma[r \mapsto \texttt{ref}\ f'']$ by application of rules E-Inst and E-Addr. In the latter case, $\gamma' = \gamma[r \mapsto \texttt{ref}\ \hat{f}]$ in the latter case by application of rules E-Inst and E-Addr-Intrinsic.

Case ($r \leftarrow \texttt{load«}t\texttt{»}(e), s''$). Evaluation of $\gamma(e)$ is well-defined by inversion of rules T-Inst and T-Load and application of Lemma 3.A.7. Let $w = \mathit{width}(t)$. If $v = \mu(\gamma(e), w)$ is interpetable as a value of type t, then $s' = s''$ and $\gamma' = \gamma[r \mapsto v]$ by application of rules E-Inst and E-Load. Otherwise, $s' = \epsilon$ and $\Psi' = \Psi, \texttt{halt}(\textbf{ex}(\text{mem error}))$ by application of rule E-Load-Abnormal.

Case ($r \leftarrow \texttt{store}(e_1, e_2), s''$). Evaluation of $\gamma(e_1)$ and $\gamma(e_2)$ are well-defined by inversion of rules T-Inst and T-Store and application of Lemma 3.A.7. Let $w = \mathit{width}(t)$. If $\gamma(e_1) > 0$, then $s' = s''$, $\gamma' = \gamma[r \mapsto \texttt{unit}]$, and $\mu' = \mu[\gamma(e_1) \mapsto \gamma(e_2)]$ by application of rules E-Inst and E-Store. Otherwise, $s' = \epsilon$ and $\Psi' = \Psi, \texttt{halt}(\textbf{ex}(\text{mem error}))$ by application of rule E-Store-Abnormal.

Case ($r \leftarrow \texttt{call}(e_0, \overline{e_i}), s''$). Evaluation of $\gamma(e_0)$ and evaluation of each $\gamma(e_i)$ is well-defined by inversion of rules T-Inst and T-Call and application of Lemma 3.A.7. By inversion of rules T-Inst and T-Call, $\gamma(e_0)$ must either have the form $\texttt{ref}\ f''$ such that $f'' \in p$ or the form $\texttt{ref}\ \hat{f}$. In either case, the arity and type of expressions are guaranteed match the parameters of the function by inversion of rule T-Call. First we consider the former case. Let f' be a clone of function f'', let b' be the entry block of function f', and let $\mathit{param}(b') = \langle \overline{r_i \colon t_i} \rangle$. Then, $s' = \mathit{stream}(b'), \langle f, b, r, s'' \rangle$ and $\gamma' = \gamma[\overline{r_i \mapsto \gamma(e_i)}]$ by application of rules E-Call and E-Ref. Now consider the latter case. If $\hat{f}(\mu, \nu, \overline{\gamma(e_i)}) = (\mu', \nu', v)$, then $s' = s''$ and $\Psi' = \Psi, \hat{f}(\overline{\gamma(e_i)})$ by application of rule E-Intrinsic. Otherwise, $s' = \epsilon$ and

$\Psi' = \Psi, \hat{f}(\overline{\gamma(e_i)}), \texttt{halt}(\textbf{ex}(\text{err}))$ by application of rule E-Intrinsic-Abnormal.

Case (`switch` e $\overline{cv_i \mapsto ref_i}$ $ref_d, \hat{s}$). Evaluation of $\gamma(e)$ is well-defined by inversion of rule T-Switch and application of Lemma 3.A.7. Let $\ell(\overline{e_i})$ be the target block reference chosen by the selection rules E-Case and E-Case-Default. Evaluation of each $\gamma(e_i)$ is well-defined by inversion of rules T-Switch and T-Ref and application of Lemma 3.A.7. Let $b \in f$ be a block such that $lab(b) = \ell$ and let $param(b) = \langle \overline{r_i : t_i} \rangle$. Then, $s' = stream(b)$ and $\gamma' = \gamma[\overline{r_i \mapsto \gamma(e_i)}]$ by application of rules E-Switch and E-Ref.

Case (`return` $e, \hat{s}$). Evaluation of $\gamma(e)$ is well-defined inversion of rule T-Return and application of Lemma 3.A.7. If $\hat{s} = \epsilon$, then $s' = \epsilon$ and $\Psi' = \Psi, \texttt{halt}(\gamma(e))$ by application of rule E-Return-Last. Otherwise, $\hat{s} = \langle f', b', r, s'' \rangle$ and $s' = s''$ and $\gamma' = \gamma[r \mapsto \gamma(e)]$ by application of rule E-Return. □

Theorem 3.A.10 (Preservation)**.** For every step of evaluation

$$(\langle p, f, b \rangle \mid \gamma \mid \mu \mid \nu \mid \Psi;\ s) \to (\langle p, f', b' \rangle \mid \gamma' \mid \mu' \mid \nu' \mid \Psi';\ s')$$

where all functions of program p are well-typed, the stream s is recursively independent from its return context, $(p\ f \mid \Gamma) \vdash s$, and $\Gamma \vdash \gamma$, then for some register environment typing Γ', s' is recursively independent from its return context, $(p\ f' \mid \Gamma') \vdash s'$, and $\Gamma' \vdash \gamma'$.

Proof. We prove by case analysis on the evaluation rule. We cover similar rules in bulk first. For brevity, it is safe to assume that $f' = f$ unless otherwise stated.

Case (E-Inst). In this case, s has the form $r \leftarrow O, s'$. Let v be the value yielded by the evaluation of the operator O by inversion of one of the following rules: E-Operator, E-Move, E-Addr, E-Addr-Intrinsic, E-Load, or E-Store. Let $\Gamma \vdash v : t$ by respective inversion of rule T-Operator, T-Move, T-Addr, T-Addr, T-Load, and T-Store.

Then, $\gamma' = \gamma[r \mapsto v]$ and, by inversion of rule T-Inst, $(p\ f \mid \Gamma, (r\colon t)) \vdash s'$. Then, $\Gamma' = \Gamma, (r\colon t)$ and $\Gamma' \vdash \gamma'$ by application of Lemma 3.A.6. Lastly, s' is recursively independent from its return context by application of Corollary 3.A.3.

Case (E-Call). In this case, s has the form $r \leftarrow \texttt{call}(e_0, \overline{e_i}), s''$. Let $\gamma(e_0) = \texttt{ref}\ f''$, f' be a clone of the function f'', b' be the entry block to f', and $\mathit{param}(\mathit{entry}(f')) = \langle \overline{r_i\colon t_i} \rangle$. Then, $s' = \mathit{stream}(b'), \langle f, b, r, s'' \rangle$ and $\gamma' = \gamma[\overline{r_i \mapsto \gamma(e_i)}]$. By inversion of rule T-Call, $\Gamma \vdash e_0\colon (\overline{t_i}) \to t$ and $\Gamma \vdash e_i\colon t_i$ for each e_i. Then, $\Gamma' = \Gamma, \overline{(r_i\colon t_i)}$. By inversion of rule T-Func, $\mathit{nonlocal}_f(b) = \emptyset$. By inversion of rule T-Block, $(p\ f' \mid \emptyset) \vdash \mathit{stream}(b')$ and, by application of Lemma 3.A.5, $(p\ f' \mid \Gamma') \vdash \mathit{stream}(b')$. Then, $(p\ f' \mid \Gamma') \vdash s'$ by application of Lemma 3.A.8 and $\Gamma' \vdash \gamma'$ by repeated application of Lemma 3.A.6. Lastly, $\mathit{stream}(b')$ is recursively independent from $\langle f, b, r, s'' \rangle$ as b' is a block of a cloned function referencing only fresh registers and s'' is recursively independent from its own return context.

Case (E-Intrinisc). In this case, s has the form $r \leftarrow \texttt{call}(e_0, \overline{e_i}), s'$ where $\gamma(e_0) = \texttt{ref}\ \hat{f}$. Let $\hat{f}(\mu, \nu, \overline{\gamma(e_i)}) = (\mu', \nu', v)$ and let $\gamma' = \gamma[r \mapsto v]$. By inversion of rule T-Call, we have $\Gamma \vdash e_0\colon (\overline{t_i}) \to t$ which implies $v\colon t$. By inversion of rule T-Inst, $(p\ f \mid \Gamma, (r\colon t)) \vdash s'$. Then, $\Gamma' = \Gamma, (r\colon t)$ and $\Gamma' \vdash \gamma'$ by application of Lemma 3.A.6. Lastly, s' is recursively independent from its return context by application of Corollary 3.A.3.

Case (E-Switch). In this case, s has the form $\texttt{switch}\ e\ \overline{cv_i \mapsto \mathit{ref}_i}\ \mathit{ref}_d, \hat{s}$. Then, if $\ell(\overline{e_i})$ is the target reference chosen by the application of rules E-Case and E-Case-Default, $\mathit{block}_f(\ell) = b'$, $\mathit{stream}(b') = s''$, $\mathit{param}(b') = \langle \overline{r_i\colon t_i} \rangle$, $s' = (s'', \hat{s})$, and $\gamma' = \gamma[\overline{r_i} \mapsto \gamma(e_i)]$. By inversion of rule T-Ref, $\mathit{nonlocal}_f(b) \subseteq \Gamma$ and $\Gamma \vdash e_i\colon t_i$ for each e_i. Then, we have $\Gamma' = \Gamma, \overline{(r_i\colon \gamma(e_i))}$ and $\Gamma' \vdash \gamma'$ by application of Lemma 3.A.6. Next, by inversion of

rule T-Block $(p\ f \mid \mathit{param}(b), \mathit{nonlocal}_f(b)) \vdash s''$. It follows that $(p\ f \mid \Gamma') \vdash s''$ by application of Lemma 3.A.5. If $\hat{s} = \epsilon$, then $(p\ f \mid \Gamma') \vdash s', \hat{s}$ is immediate. Otherwise, $\hat{s}$ has the form $\langle f', b', r, s''' \rangle$ such that $(p\ f \mid \Gamma) \vdash \hat{s}$ by inversion of rule T-Term and $(p\ f \mid \Gamma') \vdash (s', \hat{s})$ by application Lemma 3.A.8. Lastly, s' is recursively independent from its return context by application of Corollary 3.A.4.

Case (E-Return). In this case, s has the form `return` $e, \hat{s}$ where $\hat{s} = \langle f', b', r, s' \rangle$ and $\gamma' = \gamma[r \mapsto \gamma(e)]$. By inversion of rules T-Term and T-Context, $(p\ f' \mid \Gamma, (r\colon t)) \vdash s'$. By inversion of rule T-Return, $\gamma(e)\colon t$. Then, $\Gamma' = \Gamma, (r\colon t)$ and $\Gamma' \vdash \gamma'$ by application of Lemma 3.A.6. Lastly, s' is recursively independent from its own return context as s is recursively independent from $\hat{s}$.

Case (Termination). For the application of rules E-Return-Last, E-Div-Abnormal, E-Load-Abnormal, E-Store-Abnormal, and E-Intrinsic-Abnormal, $s' = \epsilon$ and $\gamma' = \gamma$. The empty stream ϵ types by application of rule T-Empty where $\Gamma' = \Gamma$ and is trivially independent from its own empty return context. □

Lemma 3.A.11 (Well-Typed SSA)**.** If $p \mid f$ is well-typed, $\langle f, b_1, d \rangle$ is the unique definition of register r, and $\langle f, b_2, u \rangle$ is a use of register r, then $b_1 \prec_f b_2$.

Proof. Suppose this is not the case. Then, $(r\colon t) \in \mathit{implicit}(b_2)$ as $b_1 \not\prec b_2$ and there must exist some path $\mathit{entry}(f) \rightsquigarrow_f b_2$ which does not define r. Then, $(r\colon t) \in \mathit{nonlocal}_f(\mathit{entry}(f))$ by definition of the function $\mathit{nonlocal}'_f$ and rule T-Block cannot be applied. □

4 Properties

In this chapter, properties of the Waddle internal representation that must be maintained by graph transformations are defined. Section 4.1 defines *static single assignment form* that ensures each use of a register has a unique definition. Section 4.2 defines *loop-closed static single assignment form* that adds the restriction that no definition can escape the loop in which it is defined. Section 4.3 defines the *canonical form* that adds several restrictions to aid reasoning about loop structures.

4.1 Static Single Assignment Form

A control flow graph in **static single assignment form** (SSA form) requires that every register is defined at a *unique* location in the control flow graph and that every use of a register is dominated by its definition. A definition *dominates* a use if either the definition comes before the use in the same block or the block containing the definition dominates the block containing the use.

SSA form was first introduced by Rosen et al. [56] as a means to recognize and exploit equivalences of expressions in a program. Prior to SSA form, redundancy elimination techniques were either restricted to comparing computations lexically or restricted to value numbering within a single basic block. Early global value numbering algorithms required computing equivalence classes for variables that are assigned multiple times [7]. With SSA form, global value numbering can be performed without the need to consider multiple assignments.

With single assignment, use-definition relationships are explicit as there is exactly one *reaching definition* for every use of a register. Dead code elimination becomes an iterative

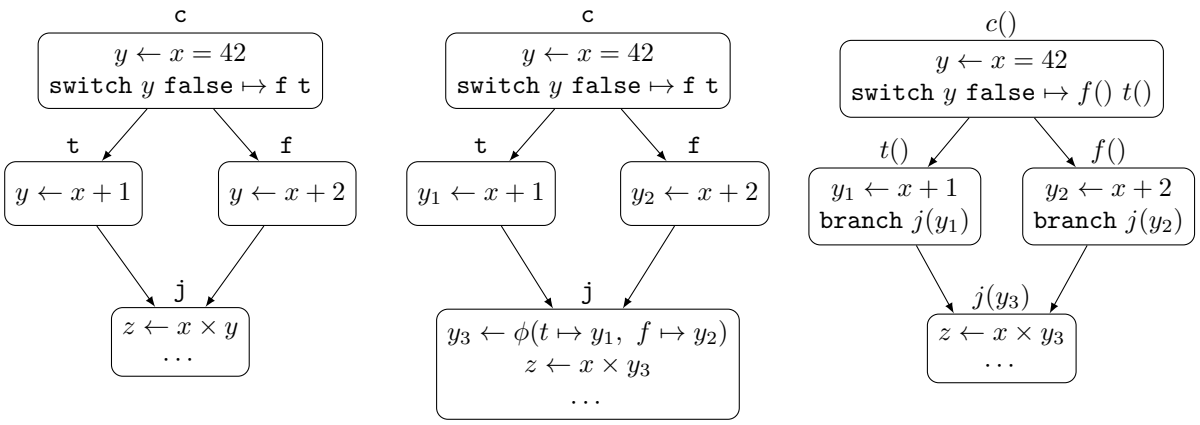


Figure 4.11: Insertion of a ϕ-node at a *join point*. The graph on the right uses block parameters instead of explicit ϕ-nodes.

search for registers with no uses. Copy propagation can be implemented trivially by replacing all uses of a register r defined as $r \leftarrow v$ with v. SSA form also greatly simplifies register allocation, as the interference graphs of such programs are chordal and can be colored optimally in linear time [32, 50]. Unfortunately, performing register assignment *before* SSA elimination poses a significant difficulty. If a copy must occur between two registers that have been spilled, there will be no additional registers to use for temporary space on architectures that do not support memory-to-memory copies.

A control flow graph can be converted into SSA form by versioning registers at each assignment point so that all assignments are unique. Then, each use of a register r is rewritten to use the version of the variable most recently assigned on the control flow path to the use. If there are multiple control flow paths to the use with different live version of r, then a special *phony* instructions denoted by ϕ-nodes are inserted where these paths join. These nodes represent a choice of values determined by the specific control flow path that was taken during program evaluation. It is important to note that ϕ-nodes are a temporary and notational artifact of the internal representation and are replaced by copy instructions when converted out of SSA form. This process is illustrated in Figure 4.11. SSA construction algorithms are discussed further in Section 5.3.

The use of ϕ-nodes are subsumed by Waddle's use of block parameters and terminator arguments. A similar technique is also used by the Swift Intermediate Language [31] as early as 2015. Block parameters provide a major conceptual advantage. The node $z \leftarrow \phi(x, y)$ defines z and uses x and y. However, these uses actually occur on the *edge* into this block as y is not guaranteed to be well-defined on the control flow path from the first predecessor, nor is x guaranteed to be well-defined on the control flow path from the second predecessor. When using block parameters, the ϕ-nodes are split across block boundaries such that the uses occur at the point of the arguments and the definitions occur at the point of the block parameters. This makes the location of register definitions and uses explicit.

Several interesting extensions to SSA form exist to encode more information in the internal representation. These extensions are discussed briefly below, but are not considered in the remainder of this work.

Static Single Information form [10] adds σ-nodes, the dual of ϕ-nodes, at *branch points* such that uses of variables in disjoint control flow paths are distinguishable. For example, a branch predicate may test a value against zero before a division. On each control flow path it is possible to make different equality statements against the same value. Such statements are not possible to make in SSA form without additional dataflow analysis.

Psi-SSA form [61] supports encoding predication on architectures that support conditional execution. For example, the expression $r \leftarrow \psi(v_1, p \ ? \ v_2)$ states that r has value v_2 if predicate p is true and has value v_1 otherwise. Figure 4.12 shows an example of conditional execution in the ARM instruction set. The `CMP` instruction sets a series of flags that are read by the following instructions (using the conditional execution suffixes `GT`, `LT`, and `NE`). If the corresponding flag is not set, then the operation does nothing for that cycle.

Hashed SSA [20] models certain memory operations. The node $r' \leftarrow \chi(r)$ is inserted at program locations where r may be modified through indirect stores or overlapping memory

```
loop:  CMP     R_i, R_j
       SUBGT   R_i, R_i, R_j
       SUBLT   R_j, R_j, R_i
       BNE     loop
```

Figure 4.12: Euclid's Algorithm in the ARM instruction set.

regions and the node $\mu(r)$ is inserted wherever r is read. This additional information allows a store to be declared dead once no μ-node references some location.

Array SSA form [38] models the partial update of array values by tracking the most recent time a definition to an array element was evaluated. The statement $@A_i[j] = i$ denotes that the definition A_i updated index j at time i. This form also requires that ϕ-nodes be inserted at certain definition points in order to merge partial updates of arrays. This is unnecessary in SSA as assignments completely kill the previous scalar value. Array SSA form has applications in parallelization of loop iterations.

4.2 Loop-Closed Static Single Assignment Form

A control flow graph in **loop-closed static single assignment form** (LCSSA form) requires, in addition to the properties of SSA, that definitions do not escape loops. Specifically, all uses of a register defined in a loop must occur in that loop. This additional property tends to make control flow transformations over loops more *local*. Transformations such as loop unrolling may affect blocks arbitrarily far from the loop being transformed. Figure 4.21 demonstrates such non-locality. After the block b is duplicated, a second reaching definition of the logical register s is made available to the block distantly reachable via block e. To repair this graph, block parameters must be inserted at join points and the use must be rewritten to the new unique reaching definition. To enforce locality of loop transformations,

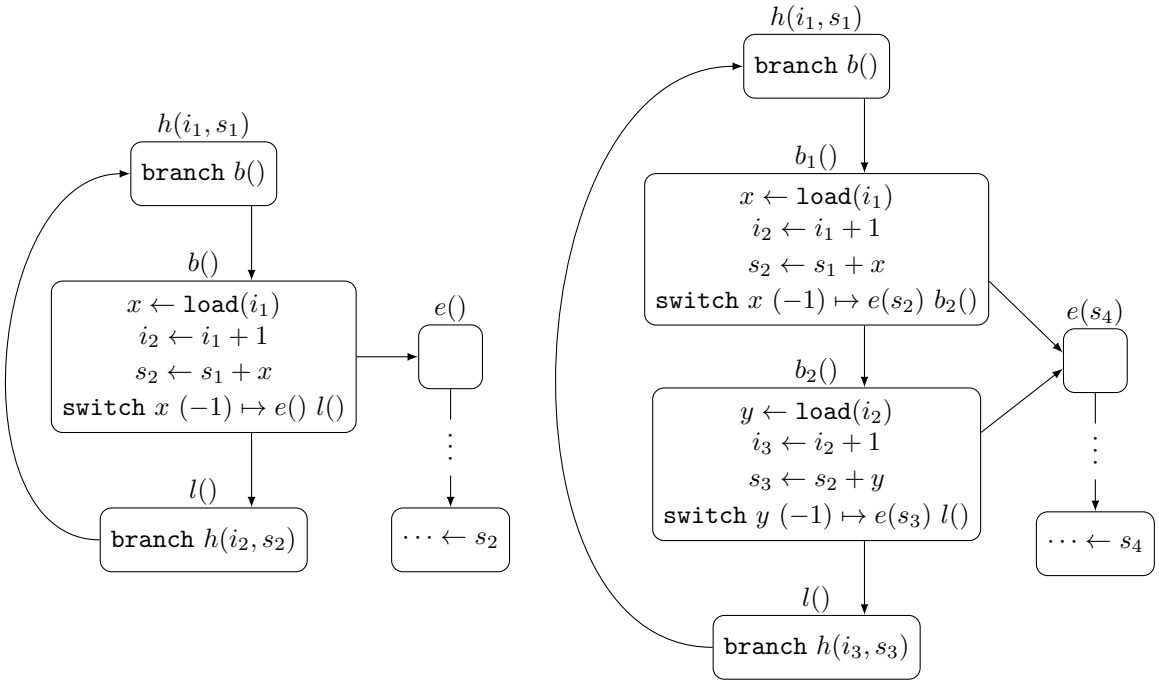


Figure 4.21: Unrolling a loop can affect blocks arbitrarily in the graph.

LCSSA form ensures that all values escape their defining loop through explicit block parameters. LCSSA construction and reconstruction transformations are discussed further in Section 7.2.

4.3 Canonical Form

A control flow graph in **canonical form** is a control flow graph with the following restrictions applied to each loop l.

Property 4.3.1. Loop l has exactly one latch.

Property 4.3.2. Loop l has exactly one preheader.

Property 4.3.3. Every exit of loop l has predecessors only in l.

The latter two properties deal with *dedication* of a block to a particular loop. A preheader and a dedicated exit clearly define the boundaries of a loop, and are ineffectual to the program

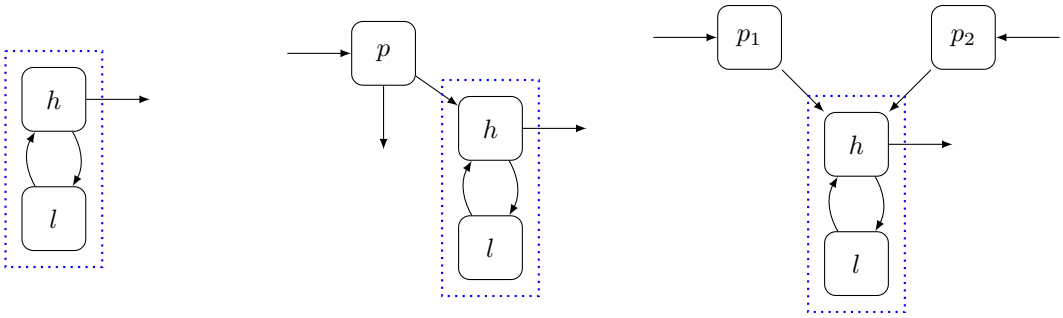


Figure 4.31: Violations of canonical form that prevent the efficient hoisting of a loop invariant instruction.

when the loop is not otherwise entered or exited.

(Property 1) ensures that each loop has exactly one backedge. As it turns out, this property greatly simplifies loop trip count calculation and reconstruction of loops and loop nesting structures after a change in the control flow graph.

(Property 2) helps to simplify *hoisting* of instructions from within loop bodies. A dedicated preheader ensures a safe target for such instructions. An instruction hoisted from loop l must be moved to each block $P \equiv pred_G(header_G(l)) \setminus body_G(l)$, otherwise a path is created that no longer executes this instruction. When $|P| > 1$, the hoisted instruction is duplicated and there now exist duplicate definitions of the same register. This breaks SSA form and requires reconstruction. If a block $b \in P$ has multiple successors, then there may be a path containing b that never enters the loop, and hoisting the instruction to b will add an instruction to this path. Control flow graphs violating this property are illustrated in Figure 4.31.

(Property 3) similarly helps to simplify *sinking* of instructions from within loop bodies. Dedicated exits ensure a set of safe targets for such instructions. An instruction sunk from loop l must be moved to the each block $e \in exit_G(L)$. If $header_G(l) \not\prec_G e$, then sinking the instruction to e may add an instruction on a path that never enters the loop. If an exit block $e \in exit_G(l)$ has a predecessor outside of l (but is still dominated by $header_G(l)$),

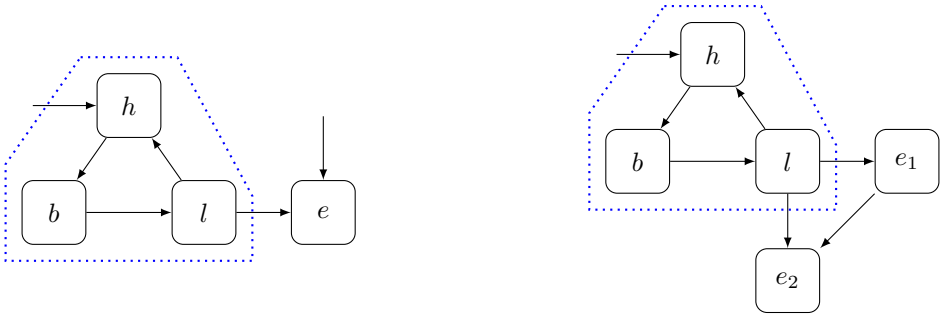


Figure 4.32: Violations of canonical form that prevent the efficient sinking of a instruction calculated but unused/unobserved in the loop body.

then the sunk instruction may be executed twice on a path leaving the loop. Control flow graphs violating this property are illustrated in Figure 4.32. It is important to note that an instruction sunk to multiple exits, even when in canonical form, may need to be rewritten to preserve SSA form.

A flowgraph can be converted into canonical form by inserting additional blocks on edges that violate one of the three restricted properties of a loop. The details of canonicalization are further discussed in Chapter 7.

5 Related Work

In this section, algorithms that form the foundation of Waddle's reconstruction techniques are introduced. Some of the algorithms presented in this chapter are used directly by the Waddle implementation. Others are presented for completeness. Section 5.1 and Section 5.2 illustrate construction and reconstruction algorithms for the dominator tree. Section 5.3 and Section 5.4 illustrate construction and reconstruction algorithms for static single assignment form.

5.1 Dominator Tree Construction

Dominator tree construction is an extremely well-researched topic. This section is broken into four sections. First, we discuss a class of *iterative* algorithms that use solutions to dataflow equations to build a dominator tree. This also serves as a good foundation for how these algorithms were later improved. Second, we discuss the Lengauer-Tarjan algorithm which is very widely used in practice. Third, we discuss an improvement on the Lengauger-Tarjan algorithm which first approximates a solution, then refines it. Fourth, we discuss a few examples of linear time algorithms, some of which sacrifice clarity for speed.

5.1.1 Iterative Algorithms

Aho and Ullman [2] and later Purdom and Moore [51] present straightforward techniques to determine the dominators of a block b by removing it from the graph and determining which nodes are no longer reachable from b_0. If b' is no longer reachable after removing b, then all paths from b_0 to b' include b. Conversely, if b' is still reachable after removing b, then there

must be another path from b_0 to b' that avoids b. This algorithm has a worst-case complexity of $\mathcal{O}(mn)$. A similar algorithm by Aho et al. [1] reduces this complexity to $\mathcal{O}(m \log(m))$ for reducible graphs.

Allen and Cocke [2, 3] showed that the dominance relation is equivalent to the maximal solution of the following forward data-flow equation. A solution to $dom(b)$ for all blocks in a control flow graph can be computed by finding the maximum fix point solution to the following set of equations.

$$dom(b_0) = \{b_0\} \qquad\qquad dom(b) = \left(\bigcap_{p \in pred(b)} dom(p) \right) \cup \{b\}$$

These equations form a *rapid framework*, and such an iterative algorithm that processes the graph in a reverse post-order with respect to T will halt in no more than $d(G,\ T) + 3$ iterations, where $d(G,\ T)$ is the *loop connectedness* number with respect to T (the maximum number of backedges in a cycle-free path) of G [37]. For reducible graphs, the value of $d(G,\ T)$ is independent of the traversal order encoded by T. Typically, control flow graphs are reducible [3] and $d(G,\ T) \leq 3$ [39]. A naïve fixed-point solution of the above dataflow equations can be found in $\mathcal{O}(mn^2)$ time.

Cooper et al. [22] present an engineered version of the iterative algorithm that performs well in practice. The algorithm uses the key insight that the *dom* sets resulting from the above data-flow equations can be ordered

$$dom(b) = \langle b_0,\ \ldots,\ idom(idom(b)),\ idom(b),\ b \rangle$$

such that the set encodes the unique path $b_0 \rightarrow_D b$. The engineered algorithm uses an ordered intersection operator that creates $dom(b)$ precisely in this order and encodes the elements of the set *directly* in the dominator tree under construction. By ordering dominators in this manner, if $dom(b_1) \cap dom(b_2) \neq \emptyset$, then the intersection is a prefix of both $dom(b_1)$ and $dom(b_2)$.

The time required for each iteration is dominated by performing intersections on paths $b_0 \rightarrow_D b_1$ and $b_0 \rightarrow_D b_2$. These intersections are equivalent to finding the ancestors of $nca_D(b_1, b_2)$ in D. If these queries are performed by tracing paths up the tree, a single iteration of Cooper's algorithm takes $\mathcal{O}(nm)$ time. In practice, many queries will require $o(n)$ time.

Cooper et al. claim their algorithm to be competitive with the Simple Lengauer-Tarjan algorithm (discussed below in Section 5.1.2) for small and moderately-sized graphs. Later, testing against a different implementation of the simple Lengauer-Tarjan algorithm led to contradictory experimental results [27, 30].

5.1.2 Lengauer-Tarjan Algorithm

Lengauer and Tarjan propose an algorithm that uses properties of *semidominator paths* to make an initial guess of a block's immediate dominator, then refines this initial guess in a second pass [45, 47]. The algorithm works in three sequential steps, briefly described below. We will use a concrete example to illustrate the latter two steps.

1. **Order blocks.** Generate a depth-first spanning tree T and a total ordering of vertices denoted O_T.

2. **Calculate semidominators.** The *semidominator* of a node $b \neq b_0$, denoted $sdom(b)$, is the node s with the minimal rank with respect to O_T such that there is a *semidominator path* $P = s \rightsquigarrow b$ where $b \leq_{O_T} b_i$ for all $b_i \in P \setminus \{s\}$. Notice for each block $b \neq b_0$, $sdom(b) \neq b$.

3. **Calculate immediate dominators.** For each block $w \neq b_0$, a block $u \neq sdom(w)$ is chosen such that $u \in sdom(w) \rightarrow_T w$ and $sdom(u)$ has the smallest rank with respect to O_T. Then, the immediate dominator of w can be found directly by the following.

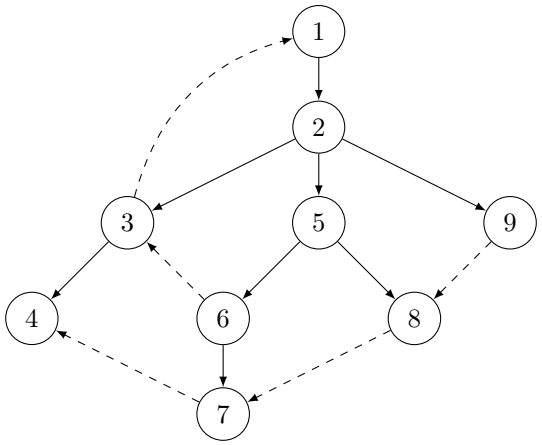


Figure 5.11: A depth-first spanning tree with flow graph edges shown as dashed lines.

$$idom(w) = \begin{cases} sdom(w) & \text{if } sdom(w) = sdom(u), \\ idom(u) & \text{otherwise} \end{cases}$$

Example In Figure 5.11, the block with rank 7 has *semidominator paths* $\langle 6,\ 7\rangle$, $\langle 5,\ 8,\ 7\rangle$, and $\langle 2,\ 9,\ 8,\ 7\rangle$, making its semidominator the block with rank 2. The tree path from 2 to 7 is $\langle 2,\ 5,\ 6,\ 7\rangle$, making 5, 6, and 7 candidates to be the *relative dominator* of 7. The semidominator of these blocks are 2, 5, and 2 respectively, making u either the block with rank 2 or the block with rank 5. In either case, the semidominators are the same and the immediate dominator of the block with rank 7 is the node with rank 2.

The blocks of the graphs are processed in reverse (right to left) pre-order to ensure that the necessary information is available when needed. In order to find the block u with the minimal semidominator without exhaustive search, a *link-eval* data structure is used. This data structure maintains a forest $\mathcal{F}$ so that by the time a block u with a minimal semidominator needs to be chosen, there is a tree in $\mathcal{F}$ rooted at $sdom(w)$ containing the blocks in P. Initially, each block in T is a single-vertex tree in $\mathcal{F}$. The data structure defines the following two operations.

- link(w) Insert the edge (p, w) into $\mathcal{F}$, where $p = p_T(w)$.

- eval(w) Let r be the root of the subtree in $\mathcal{F}$ containing w. If $w = r$, return r. Otherwise, return the block $u \in P \setminus \{r\}$ with the minimum semidominator.

The algorithm runs in $\mathcal{O}(m \log_{m/n+1}(n))$ using a simple implementation of the link-eval data structure using only path compression as an optimization. This bound can be reduced to $\mathcal{O}(m\alpha(m, n))$ using a more sophisticated implementation of the data structure that keeps $\mathcal{F}$ relatively balanced. We refer to these two versions as the *Simple* Lengauer-Tarjan algorithm and the *Sophisticated* Lengauer-Tarjan algorithm, respectively. Here, $\alpha(m, n)$ refers to the extremely slow-growing functional inverse of the fast-growing Ackermann function, defined as

$$\alpha(m,\ n) = \min\{i \geq 1 \mid A(i,\ \lfloor \frac{m}{n} \rfloor) > \log(n)\}$$

where one of many equivalent definitions of the Ackermann function $A(i, j)$ is defined for parameters $i, j \geq 1$ as follows.

$$\begin{aligned} A(1,\ j) &= 2^j \\ A(i,\ 1) &= A(i-1,\ 2) \\ A(i,\ j) &= A(i-1,\ A(i,\ j-1)) \qquad\qquad \text{for } i,\ j \geq 2 \end{aligned}$$

It is important to note that $\alpha(m, n) \leq 4$ for all conceivable values of m and n [23]. Although the algorithm is asymptotically superlinear, it is effectively linear in practice.

5.1.3 Semi-NCA

Georgiadis [27] presents an alternative version of the Lengauer-Tarjan algorithm, referred to as *SNCA*, which uses a simpler method to calculate immediate dominators from semidominators. The algorithm relies on the observation that for any block $b \neq b_0$, the following

holds.

$$idom(b) = nca_D(\{sdom(b),\ p_T(b)\})$$

The algorithm performs the first two steps of the Lengauer-Tarjan algorithm to calculate semidominators. Then, the dominator tree D is built incrementally by inserting blocks in the order that they occur within O_T. The parent of b in D is the block $x \in b_0 \rightarrow_D p_T(b)$ with the largest rank such that $x \leq_{O_T} sdom(b)$. If this step is performed naïvely, then the modified step takes $\mathcal{O}(n^2)$ in the worst-case. However, empirical results indicate that semidominators are generally a good approximation of immediate dominators. Gabow, Cole, and Hariharan's [26] results imply this step can be accomplished with linear asymptotic complexity, but the application of this result to the algorithm does not seem to have been attempted.

This algorithm is simpler for several reasons. First, there is no need to maintain an auxiliary structure that maintains a set of blocks sharing a semidominator, and another auxiliary array no longer needs to be maintained as the *eval* operation can be simplified. Third, the algorithm can be performed using one less pass over the blocks of the graph as the immediate dominator is found directly, not implicitly through relative dominators.

5.1.4 Linear Time Algorithms

In 1985, Harel [33] introduced a linear-time algorithm for computing dominators, but the description was later found to be incomplete [8, 29]. Alstrup et al. [8] later gave a true linear-time algorithm using Fredman and Willard's Q-heaps [25]. However, the construction of this algorithm is purely theoretical. Fredman and Willard's Q-heaps forego any pretense of practicality and presume $n \geq 2^{12^{20}}$.

Alstrup and Lauridsen [9] describe a simple linear time algorithm for constructing the dominator tree of a reducible control flow graphs. In summary, the immediate dominator

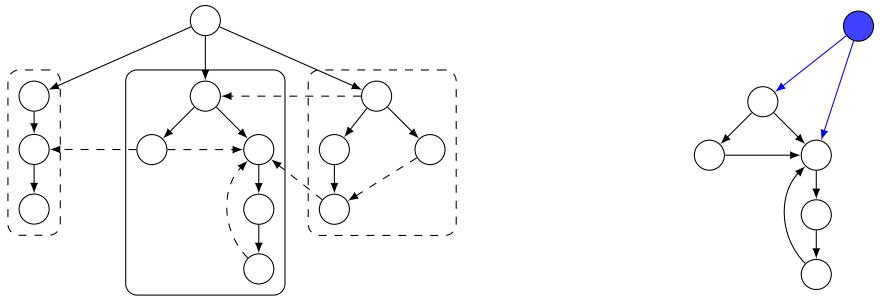


Figure 5.12: A depth-first spanning tree with flow graph edges shown as dashed lines. The figure on the right is the augmented graph of the microtree with a solid outline, with the root and blue edges highlighted.

of each block $b \neq b_0$ is $nca_D(pred(b))$, and processing the blocks in topological order (with backedges removed) guarantees that the predecessors of b have already been inserted into D by the time b is processed. Gabow [26], and later Cole and Hariharan [21], present a useful algorithm that computes nearest common ancestor queries for trees in which vertices are added and removed only at the leaves in linear time.

Buchsbaum et al. [15] present a claimed linear-time algorithm based on the Lengauer-Tarjan algorithm that first decomposes T into a set of vertex-disjoint regions called *microtrees*. Each microtree contains at most g blocks for some fixed parameter g. Every block belongs to a single microtree, and the microtree containing block b is denoted by $micro(b)$.

Each non-trivial microtree M is associated with an *augmented-graph*, $aug(M)$, which is the control flow graph induced by the blocks of M plus an additional *phony* block $root(M)$, representing the blocks of the graph external to the microtree, and the following set of *blue* edges.

$$\{(root(M),\ b_j) \mid (b_i,\ b_j) \in E, b_i \notin M, b_j \in M\}$$

Augmented graphs are illustrated in Figure 5.12. The Simple Lenguager-Tarjan algorithm is applied to each augmented graph and the result is memoized so that the result is not recalculated for isomorphic microtrees. Notice that each graph has $O(g)$ blocks and $O(g^2)$ edges, and this step can be completed in linear time when g is $\mathcal{O}(\log^{1/3} n)$ [29].

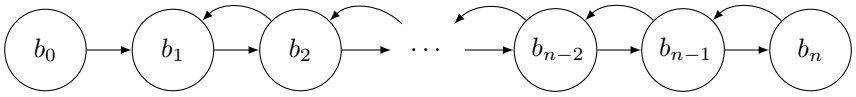


Figure 5.21: Inserting the sequences of edges $(b_{n-2}, b_n), (b_{n-3}, b_n), \ \ldots, \ (b_1, b_n)$ changes the immediate dominators of $\Theta(n)$ blocks per operation, and $\Theta(n^2)$ blocks overall.

For each block b, it is determined whether $idom(b) \in micro(b)$. If so, the immediate dominator is already known. Otherwise, the immediate dominator is calculated by approximating a *pushed external dominator* and refining the result.

The inverse Ackermann function $\alpha(m, \ n)$ reduces to a constant when m/n is $\Omega(\log^{\mathcal{O}(1)} n)$ (when m is slightly superliner on n). The runtime analysis claims to achieve this reduction by applying new bottom-up disjoint set union results to the original link-eval data structure [14]. The analysis was found to be incorrect and the runtime was later repaired by changes to the implementation of the link-eval data structure to support the original claims [16, 27].

5.2 Dominator Tree Reconstruction

Carroll and Ryder [17] point out the *non-locality* property of domination, as follows.

> Given two nodes x and y in the control flow graph, determining whether x dominates y depends on the presence or absence of paths through nodes arbitrarily far from either x or y. Adding or removing a single flow edge – an act that can add or remove a large number of paths – can thus affect domination between nodes arbitrarily far from the altered edge.

A sequence of $\Theta(n)$ edge insertions or deletions can be constructed such that **each** operation changes the immediate dominator of $\Theta(n)$ blocks [48]. Such a sequence is illustrated in Figure 5.21. In practice, it may be unlikely for such a pessimistic sequence to occur naturally.

5.2.1 Ramalingam-Reps Algorithm

Ramalingam and Reps [53] describe an approach to update the dominator tree of a reducible control flow graph when a single flow edge is inserted or removed. The basic idea of the algorithm is to conservatively approximate a set of *affected* blocks for which the immediate dominator changes, then re-calculate the new immediate dominator using the old dominator tree.

If the affected vertices are processed in topological order, the algorithm requires only a single pass. The control flow graph is *prioritized* such that each block $b \in B$ is assigned a priority, denoted $\mathit{priority}(b)$, where $\mathit{priority}(b_1) < \mathit{priority}(b_2)$ when $b_1 \rightsquigarrow b_2$ in the acyclic graph induced by removing the backedges. Block priorities imply a topological ordering. The insertion of an edge must also update the priority ordering of the graph. The deletion of an edge, however, does not affect the priority ordering of the graph.

The worst-case complexity of updating the dominator tree (excluding the step of updating the priority ordering when necessary) is $\mathcal{O}(\|A\|^{\leftrightarrow} \log(n))$, where A is the approximate set of affected blocks. Alpern et al. [6] present an incremental algorithm to update the priority orderings of a graph after the insertion of an edge that takes $\mathcal{O}(\|\kappa\|^{\leftrightarrow} \log(\|\kappa\|^{\leftrightarrow}))$ time in the worst case, where κ is the set of blocks whose priorities will change. In all, updating the dominator tree after inserting or removing a single edge takes $O(m \log(n))$ time.

Edge Insertion The insertion of backedges does not affect domination and requires no update to the dominator tree[1]. The insertion of a forward edge (u, w) where w was previously reachable may create an additional path to a block y that avoids $\mathit{idom}(y)$. The affected blocks are conservatively approximated by the follow set, which contains the blocks whose

[1]Of course, such a change can still deeply affect other structures and properties of the graph that may need to be updated independently.

immediate dominator lies on the ancestor path from w to $nca_D(\{u, w\})$ in D.

$$\{v \mid idom(v) \text{ dominates } idom(w) \text{ and is strictly dominated by } nca(\{u, w\})\}$$

The immediate dominator of each block in this set must be recalculated. This transformation is illustrated in Figure 5.22. If a block's dominator changes after the insertion of an edge, its new immediate dominator must be $nca_D(\{u, w\})$.

Let R be a graph rooted at w disjoint from G. The insertion of the forward edge (u, w) expands G to include the blocks in R. To update the dominator tree of G, the dominator tree of R is first calculated statically (using Lengauer-Tarjan or a similar algorithm), and the dominator tree rooted at w is made a child of u in D. Then, each forward edge from R to a previously reachable block is inserted individually as described above.

Edge Deletion The deletion of backedges does not affect domination and requires no update to the dominator tree. The deletion of a forward edge (u, w) where w is still reachable may reduce the set of paths to a block y, changing its immediate dominator. The affected blocks are conservatively approximated by the following set, which contains the blocks that are siblings of w in D.

$$\{v \mid idom(v) = idom(w)\},$$

The immediate dominator of each block in this set is recalculated. If the deletion of edge (u, w) causes the subgraph R (consisting of the blocks dominated by w) to become disconnected, then each block is removed (or marked as unreachable), and any forward edges from R to a reachable block must be deleted individually as described above.

5.2.2 Dynamic SNCA Algorithm

Patakakis et al. [49] present *DSNCA*, a dynamic version of the static *SNCA* algorithm described in Section 5.1.3. The dynamic version of the algorithm maintains semidominators

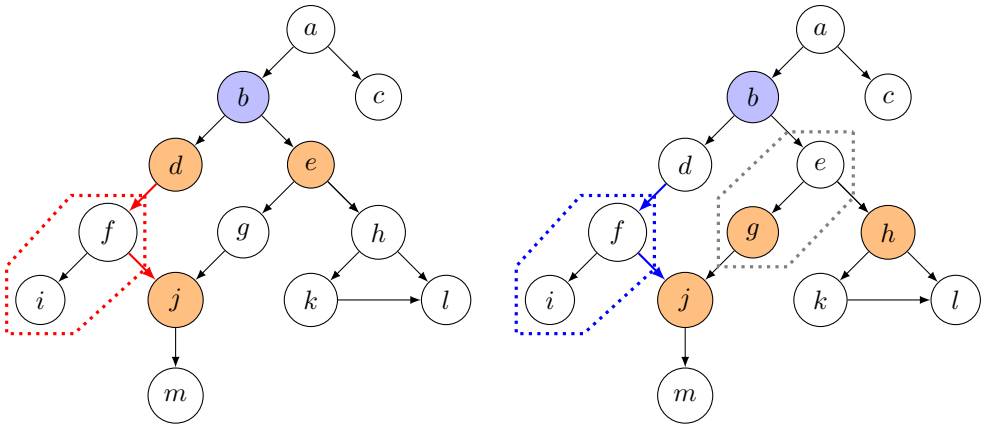


Figure 5.22: The deletion (left) or insertion (right) of edge (d, f) requires consequent processing of edge (f, j). The set of possibly affected nodes are highlighted in orange and $nca(f, j)$ is highlighted in blue.

of each block. Properties of the depth-first spanning tree T and a total ordering of vertices O_T is used as a heuristic to determine when the dominator tree can be updated locally. In this version, each block $b \in T$ is annotated with pre-order and post-order traversal ranks with respect to O_T, denoted $pre(b)$ and $post(b)$ respectively.

If the insertion or deletion of a flow edge (u, w) invalidates the ordering O_T, then the algorithm runs from scratch (and chooses a new ordering O_T in the process). The ordering is invalidated after the insertion of an edge when $pre(u) < pre(w)$ and $post(u) < post(w)$ or $w \notin T$ and is also invalidated after the deletion of an edge when $u = p_T(w)$ or $u \in sdom(w) \rightsquigarrow w$.

Otherwise, T remains a valid depth-first spanning tree and dominators can be updated incrementally. The semidominators are recomputed only for blocks $b \in T$ where $pre(b) \leq pre(w)$. The immediate dominators are recomputed from scratch from the updated semidominators for all blocks. Notice that when O_T remains a valid ordering, individual pre-order and post-order annotations do not need to be modified. On insertion a flow edge is inserted but a tree edge is not. On deletion, the relative ordering of pre-order and post-order ranks is unchanged.

Each operation take $\Omega(n)$ time, but experimental results show this algorithm to be more

efficient than static recomputation of the dominator tree using the Simple Lengauer-Tarjan algorithm [49] as well as a dynamic algorithm presented by Sreedhar et al. [57, 58], which performs better for edge insertions.

5.2.3 Depth-Based Heuristic

Georgiadis et al. [28] improve the approximation of affected vertices by restricting the traversal of D to blocks that exceed a given depth. The *exact* set of affected blocks is given by the following on the insertion of an edge (u, w), which is computed using a max-priority queue of blocks ordered by depth in D.

$$\{b \mid depth_D(nca_D(\{u, w\})) < depth_D(idom(b)) < depth_D(b')\ \forall b' \in (w \rightsquigarrow b)\}$$

A single insertion is bound by the number of blocks and their adjacent edges when finding the set of affected vertices, which is $\mathcal{O}(m)$ in the worst case. However, a sequence of k insertions can be done in $\mathcal{O}(m \min\{k, n\} + kn)$ time (and the kn factor can be reduced to k [26]). This is useful for the *bulk* insertion cases, where a previously unreachable component is reachable after the insertion of an edge and additional edges leaving the component must be processed. Experimental results show this algorithm to be more efficient than *DSNCA* (discussed in Section 5.2.2) when dealing primarily with the insertion of edges.

5.3 SSA Construction

In this section we present algorithms to convert a flow graph into SSA form. One such algorithm is incremental in nature, allowing it also to be used for SSA reconstruction as the flow graph is transformed, discussed further in the next section. Here, we also discuss the concept of *minimality*. Minimality is a desired property so that unnecessary definitions are not placed into the program. This can balloon the size of the program under optimization,

make optimization opportunities more difficult to discover, and make the optimization less efficient, as it has to manage a larger number of blocks.

Cytron et al. present an efficient algorithm to construct SSA form from an arbitrary control flow graph [24]. The algorithm consists of two sequential phases. First, ϕ-nodes of the form $r \leftarrow \phi(b_1 \rightarrow r, \ \ldots, \ b_n \rightarrow r)$ are inserted at the blocks composing the dominance frontier for each definition of the register r. Such ϕ-nodes are *trivial*, as they are defined in terms of themselves only. The insertion of such nodes causes the live ranges of each assignment to the register r to become disjoint. Then, all assignment targets are renamed with unique subscripts (r becomes r_1, r_2, ...) and each use of a register is replaced by a unique reaching definition. This step can be accomplished by maintaining a stack of reaching definitions for each register. Blocks are traversed recursively in dominance order, where an assignment to a register is a push to that register's stack and exiting a block is a symmetric pop. Each use of a register can be replaced by the top of that register's stack at the point of use. Because this algorithm requires the calculation of dominance frontiers, the algorithm is quadratic in the number of blocks.

Cytron's construction algorithm guarantees a translation into *minimal* SSA form, where the number of inserted ϕ-nodes are as small as possible subject to the following condition: if two non-null paths $b_1 \rightsquigarrow b_j$ and $b_2 \rightsquigarrow b_j$ first converge at block b_j, and both b_1 and b_2 contain an assignment to r, then a trivial ϕ-node for r has been inserted at b_j.

In minimal SSA construction, ϕ-nodes are placed at all the blocks where the reaching definitions for two registers first merge. The reaching definitions at this merge point are not necessarily *live*, and the inserted ϕ-node may never be used. Choi et al. [19] present a modification to Cytron's construction algorithm utilizing global liveness analysis that instead produces a *pruned* SSA form, which prevents the insertion of *dead* ϕ-nodes in addition to being minimal.

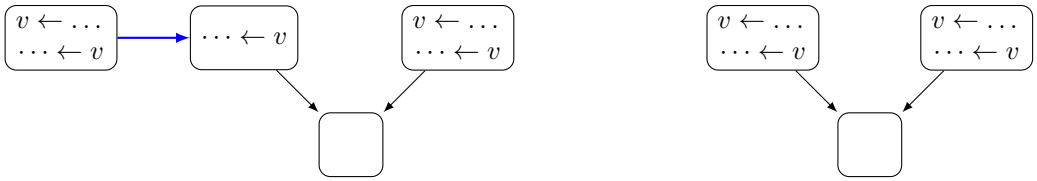


Figure 5.31: Briggs-minimal places a ϕ-node at the merge point in the graph on the left as v is live across the blue edge, but pruned SSA does not. Neither Briggs-minimal nor pruned algorithms places a ϕ-node at the merge point in the graph on the right.

A third flavor of SSA, called *semi-pruned* or *Briggs-minimal*, uses a simple local heuristic to place fewer dead ϕ-nodes than minimal SSA construction without requiring data-flow analysis [13]. Briggs et al. made the assumption that the live range of registers can often be very small and contained within a single basic block. Semi-pruned construction may place dead ϕ-nodes, but will never place a ϕ-node for a variable that is not live across **any** edge in the control flow graph. The *non-local* registers that are live across a control flow edge can be found in a linear pass over the control flow graph. The difference between pruned and semi-pruned form is illustrated in Figure 5.31.

Braun et al. [12] present an alternative construction algorithm for SSA form that notably does not depend on the calculation or maintenance of dominance frontiers. Instead of calculating the merge points from the set of definitions and pushing the definitions down to the uses of a register, the algorithm traces the control flow graph backwards and searches for a reaching definition at each use, inserting ϕ-nodes to disambiguate multiple reaching definitions as they are discovered. A version of this algorithm is presented in greater detail in Section 5.4.

This algorithm has several practical advantages. The algorithm is suited for construction of a control flow graph in SSA-form directly from an abstract syntax tree, while Cytron's algorithm requires a control flow graph to already exist. In addition, transformations such as arithmetic simplification, common subexpression elimination, constant folding, and constant propagation can be applied on the fly during construction, resulting in fewer ϕ-nodes and

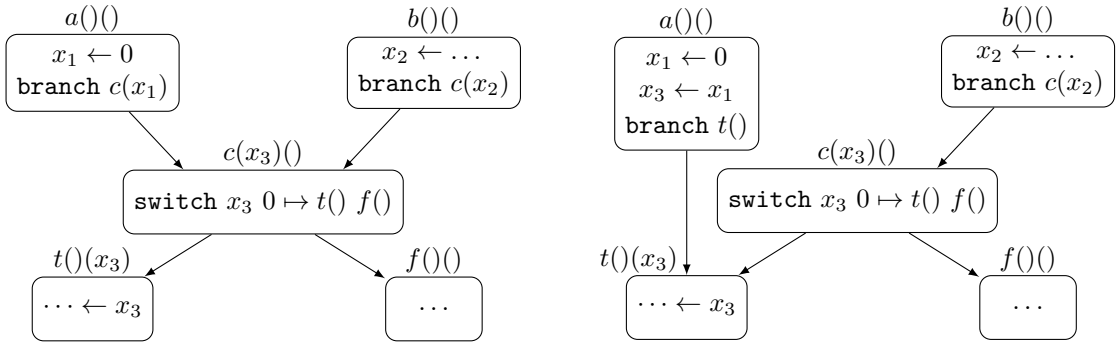


Figure 5.41: A simplified jump-threading transformation shows that the static single assignment property of a control flow graph can be easily broken.

even fewer basic blocks when the value of a conditional jump can be determined as the control flow graph is generated. This algorithm runs competitively with the highly-optimized version of Cytron's algorithm used in LLVM.

5.4 SSA Reconstruction

Some transformations insert additional assignments that can break the static single assignment property of the control flow graph. This can happen in transformations that duplicate blocks or instructions (such as loop unswitching and jump threading). Figure 5.41 demonstrates this problem. The condition ($x_3 = 0$) is known to be true when control flow enters from block a, and as the target of a's successor is statically known, a can instead jump there directly. To maintain semantic equivalence, the block a must assign x_3 so the use of the register later in execution is well-defined. While there is no semantic problem with the resulting control flow graph, it defines x_3 in two distinct instructions and breaks the static single assignment property. To maintain this property, the new assignment of x_3 must be given a unique name, a ϕ-node must be inserted at the new join point, and all successive uses of x_3 must be rewritten.

Braun's algorithm has immediate applicability in SSA reconstruction after modification

of the control flow graph. Transformations that clone instructions will inevitably duplicate an assignment, breaking the static single assignment property. This property can be repaired by rewriting all duplicate assignments to use fresh variables, then rewriting the uses of the original variables to refer to the correct reaching definition. A key procedure in Braun's algorithm is able to rewrite uses exactly in this manner. This algorithm is later formalized in Chapter 7 as part of *loop-closed SSA form* reconstruction, but we explain the intuition of the algorithm here.

Braun's algorithm takes as input register r assigned multiple times in the program, a set of definitions R, and a set of uses U. Then, each occurrence of register r in a use in U is replaced with the *reaching definition* of a register defined by some definition in D. It is expected that the definitions in D are rewritten to assign to fresh registers (with no uses) before application of this procedure. This ensures that register r no longer breaks the single assignment property and that all previous uses of register r instead use the nearest logical definition. It is also assumed that the set of definitions D dominate each use in U such that there is no path from the entry of the program to a use in U that does not pass through some definition in D.

The bulk of this algorithm requires finding the reaching definition of register r with respect to a use u. This definition will either be a definition in the set D, or will be a freshly inserted block parameter used to disambiguate multiple reaching definitions (in the case where the set of paths from the entry of the program to the use passes through multiple definitions). There are two simple cases to determine the reaching definition for a use u. First, if a definition in D occurs before u in the same block, then the last such definition is the reaching definition (as any earlier definition is killed by the later one). Second, if the block containing u has only one predecessor, then the reaching definition is the reaching definition of the predecessor block.

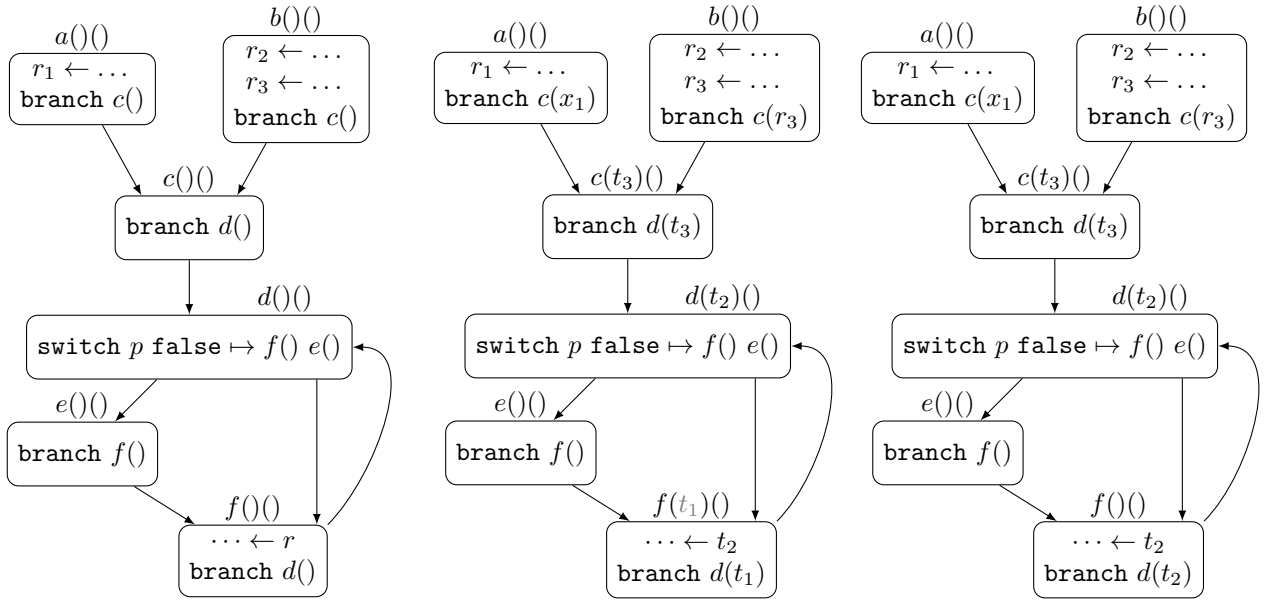


Figure 5.42: A control flow graph in broken SSA form (left) can be rewritten so that the use of r is replaced with the correct reaching definition of the register set $\{r_1, r_2, r_3\}$ (right). The middle graph shows the progress of the algorithm before t_1 is discovered to be unnecessary. The uncommitted (but referenced) block argument is shown in gray.

The general case to determine the reaching definition is a bit more involved, especially in the presence of loops. Let b denote the block containing u. If all of b's predecessors have the same reaching definition, then the reaching definition for b is unambiguous. Otherwise, multiple reaching definitions must be disambiguated based on path, and a block parameter is added to b, and the parameter for each predecessor is its own reaching definition.

Unfortunately, this recursive search does not halt within a loop. To remedy indefinite recursion, the parameter is added to block b before determining the reaching definitions of b's predecessors. During search, this block parameter is used as the reaching definition of b and breaks the recursion on that path. Unfortunately, eagerly adding block parameters will add parameters when it is not necessary. If the set of reaching definitions for a block contains only one unique register apart from its own block parameter, then the block parameter is unnecessary and can be removed (and any references to the block parameter found via an interim search can be rewritten to be the actual reaching definition of the block).

This algorithm is demonstrated in Figure 5.42, which rewrites the use of register r in block f with the correct reaching definition of the register set $\{r_1,\ r_2,\ r_3\}$, where register R_1 is defined in block a, and registers r_2 and r_3 are defined in block b. The algorithm begins in block f, which has multiple predecessors and contains no assignment to a register in $\{r_1,\ r_2,\ r_3\}$. The reaching definition of block f is speculated to be the fresh register t_1 while the reaching definition of the block's predecessors are determined. Similarly, the reaching definitions of blocks d and c are speculated to be t_2 and t_3, respectively. The reaching definitions of block a and block b are r_1 and r_3 (notice that the assignment to r_3 kills the live range of r_2). To disambiguate the multiple definitions reaching block c, the block parameter defining t_3 is committed with arguments r_1 and r_3. Similarly, t_2 is committed to block d with arguments t_3 and t_1. The definitions reaching block f are both t_2 (through d directly and e indirectly). This makes the speculative register t_1 unnecessary. Prior uses of t_1 must be replaced by t_2. One such occurrence is the argument from block f to block c. Finally, the use of r can be replaced by the sole reaching definition t_2. After discarding the definition of t_1, the block parameter t_2 becomes trivial. Such arguments can be pruned by iteratively applying a local criteria: if the arguments to some parameter r contains only r and a single value $o \neq r$, then r can be replaced by o. This may additionally make other arguments trivial.

6 Transformations

A transformation of a function f is a relation of the form

$$(f, D, H_F, L_F, X_F) \xrightarrow[P]{\mathrm{T}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

where T is an identifier and P is a set of parameters that may control the part of the function that is transformed. Along with f, the transformation takes a dominator tree D and a loop nesting forest F deconstructed into the triple (H_F, L_F, X_F) (as described in Section 2.6.3) as input. Unless otherwise stated, we assume that $D = D_f$ is the correct dominator tree and $F = F_f$ is the correct loop nesting forest for the input function. Unfortunately, this cannot always be the case as some transformations are called recursively over sections of the control flow graph (leaving each step in a locally-repaired but globally-broken state). When this convention is broken the relationship between the function and the structure will be described explicitly. A transformation outputs, symmetrically, a modified function, a dominator tree D_{out}, and the elements of a triple $(H_{out}, L_{out}, X_{out})$ from which a loop nesting forest F_{out} may be reconstructed.

6.1 Notation

Let $\xrightarrow[A]{\mathrm{T}}{}^{*}$ denote the repeated application of transformation T using each argument $a_i \in A$, defined as follows. It is assumed that the output of the sequence of transformations are independent of the order of application.

$$\frac{O \text{ is an arbitrary ordering of } A \qquad |O| = n \qquad S_i \xrightarrow[O_i]{\mathrm{T}} S_{i+1}}{S_0 \xrightarrow[A]{\mathrm{T}}{}^{*} S_n}$$

Let $b[c/c']$ denote the block obtained by replacing all occurrences of the component c with c' in block b, let $b[r/r']$ denote the block obtained by replacing all occurrences of the register r with r' in block b, and let $p[f/f']$ denote the program obtained by replacing all occurrences of the function f with f' in p.

We extend this notation to a substitution set $\sigma = \{(c_1, c'_1), \ldots, (c_n, c'_n)\}$ such that $X[\sigma]$ expands to $X[c_1/c'_1][\ldots][c_n/c'_n]$. In a slight abuse of notation, we define lookup on substitutions to behave as follows. Simply, lookup returns the mapped value if the queried value exists in the substitution and returns the queried value otherwise.

$$\sigma(c) = \begin{cases} c' & (c, c') \in \sigma \\ c & \text{otherwise} \end{cases}$$

Let $D[b_1 \mapsto b_2]$ denote the tree in which $lab(b_1)$ is made a child of $lab(b_2)$ in D. Let $D[b_1 \mapsto b_2 \colon P]$ denote $D[b_1 \mapsto b_2]$ when the predicate P holds and denote D otherwise. Let $D[b_1 \mapsto b_2 \mid P]$ denote the application of the set of assignments $\{b_1 \mapsto b_2 \mid P\}$ in arbitrary order to D. This notation is extended to the structures H and L as follows.

$$H[b_1 \mapsto b_2] = H \setminus \{(\ell_1, \ell_2) \in H \mid lab(b_1) = \ell_1\} \cup \{(lab(b_1), lab(b_2))\}$$

$$L[b \mapsto l] = L \setminus \{(\ell_1, \ell_2) \in L \mid lab(b) = \ell_1\} \cup \{(lab(b), lab(l))\}$$

6.2 Theorems

We define the following six theorems over transformations in order to show that they maintain properties of the control flow graph incrementally. Names of components bound in these proof prototypes (and the corresponding templates) will use the subscript t to reduce ambiguity between use of names in the proof and use of names in the specific transformation for which they are proving a property.

6.2.1 Symmetric Evaluation

First, we state that no transformation alters the observable semantics of a function when given the same input contexts and parameters.

Theorem Prototype 6.2.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be evaluated for n' steps with the same initial context and arguments (replacing references of f with f_{out}) and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa). More formally, for any register context γ, memory context μ, nondeterminism state ν, effects list Ψ, and function arguments $\langle\overline{v_{t_i}}\rangle$ for which there exists the following n-step evaluation sequence

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \rightarrow_{\rho}^{n} (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ s_{t_1})$$

there exists the following n'-step evaluation sequence

$$(p[f/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}}) \rightarrow^{n'} (\langle p[f/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2 \mid \mu' \mid \nu' \mid \Psi';\ s_{t_2})$$

for some integer n', some function f_{t_2}, some block b_{t_2}, some register context γ_2 and some instruction stream s_{t_2}. In the other direction, for any register context γ, memory context μ, nondeterminism state ν, effects list Ψ, and function arguments $\langle\overline{v_{t_i}}\rangle$ for which there exists the following n'-step evaluation sequence

$$(p[f/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \rightarrow_{\rho'}^{n'} (\langle p[f/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2 \mid \mu' \mid \nu' \mid \Psi';\ s_{t_2})$$

there exists the following n-step evaluation sequence

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}}) \rightarrow^{n} (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ s_{t_1})$$

for some integer n, some function f_{t_1}, some block b_{t_1}, some register context γ_1, and some instruction stream s_{t_1}.

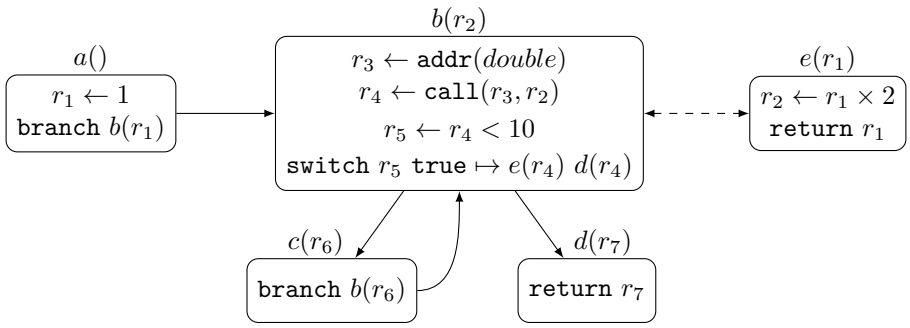


Figure 6.21: A simple two-function graph. Block a is the entry to an unnamed function and block e is the entry to a function called *double*.

Evaluation Path We use the subscript ρ, as written above, to represent the *path* of evaluation of function f up to step n. This ordered sequence contains an element for each time an instruction is evaluated where a control branches to another block. Each element contains the component that caused the transfer of control (terminators and non-intrinsic function calls) and the target block. For example, we refer to Figure 6.21. In this example, the eleventh step of evaluation starting at block a (in which control is transferred to block e for the second time) has the following six-element evaluation path.

$$\langle(\langle f_1, a, \texttt{branch}\ b(r_1)\rangle, \langle f_1, b\rangle),\ (\langle f_1, b, r_4 \leftarrow \texttt{call}(r_3, r_2)\rangle, \langle f_2, e\rangle),$$
$$(\langle f_2, e, \texttt{return}\ r_1\rangle, \langle f_1, b\rangle),\ (\langle f_1, b, \texttt{switch}\ r_5\ \texttt{true} \mapsto e(r_4)\ d(r_4)\rangle, \langle f_1, c\rangle),$$
$$(\langle f_1, c, \texttt{branch}\ b(r_6)\rangle, \langle f_1, b\rangle),\ (\langle f_1, b, r_4 \leftarrow \texttt{call}(r_3, r_2)\rangle, \langle f_3, e\rangle)\rangle$$

When a value is pushed onto a return context, it may be annotated with the current evaluation path (e.g. $\langle f, b, r, s\rangle_\rho$). This freezes the evaluation path with respect to the function f, the block b, and the stream that leads to the instruction sequence s. We use the subscript ρ' similar for evaluation of function f_{out} up to step n'. Construction of these paths is trivial during evaluation.

These sequences will play a crucial part in deterministically defining equivalent structures for pre- and post-transformation programs. For example, the equivalent block after

jump simplification can be either the original block or the clone, depending on whether or not control flow involves a particular predecessor. Using evaluation paths breaks such ambiguities.

Function References We make the assumption that the bit patterns of function reference values within programs p and $p[^{f}/_{f_{out}}]$ are consistent such that the bit pattern for value `ref` f in p is shared with the bit pattern for the value `ref` f_{out} in $p[^{f}/_{f_{out}}]$, and any function that exists in both programs shares the same bit pattern in both programs. This allows us to omit the substitution of `ref` f with `ref` f_{out} in register contexts and effects list, as both structures would be identical before and after such a substitution is applied. This also allows us to read and write function reference values to and from a memory context without requiring an additional consistent translation.

Function Cloning For each clone of a function in p there is a symmetric clone in $p[^{f}/_{f_{out}}]$. We describe this specifically in two cases (simple case first). Consider the function $f' \in p$ where $f' \neq f$. First, as $f' \in p[^{f}/_{f_{out}}]$ and as clones only need to be fresh within the context of one evaluation, using the same clone in a parallel evaluation has no effect and the same clone can be used in both an evaluation of a function in program p and a function in program $p[^{f}/_{f_{out}}]$. Now, consider a clone of function f called f'. A similar clone of f_{out} can be constructed by first replacing all occurrences of register r with register r' where r was replaced by r' in the construction of f', then replacing all remaining registers of f_{out} with fresh registers (as normal). Functions with such relationships will be referenced as *similar*.

We make the assumption that a clone of a function created in one evaluation is chosen so that no fresh register conflicts with the parallel evaluation. For example, a clone of f must not be chosen such that it contains a 'fresh' register that was introduced by the transformation to f_{out}. Intuitively, any such violating clone can be replaced by a non-violating clone without

further affecting these proofs. We denote the mapping from registers of f to registers of f' by the substitution set $\overrightarrow{\sigma_{f'}}$ such that $f' \in \mathcal{C}(f)$ and f is the *uncloned* function. When used in the proof appendices, we extend this idea so that the mapping may include entries for fresh registers introduced by a transformation. As these registers are not otherwise used in the original function, these map to an arbitrarily chosen *stable* register.

We further abuse notation by extending the meaning of $\mathcal{C}$ for registers. Let r be a register defined within function f. Then, we use $\mathcal{C}(r)$ to denote the infinite set of registers which are symmetric to register r in all clones of function f. Because registers are chosen for function clones such that they do not conflict with exiting (or future) register spaces, each such infinite set of registers may be unique from any other relevant register when necessary.

Notes for Proof Structure Proving this theorem for a particular transformation is done by induction over the number of steps of evaluation: n in the forward case and n' in the reverse case. Large chunks of these proofs are very similar, so we define a *templated* high-level proof in the appendix to this chapter, Template 6.A.1, containing *slots* that are filled in with details for a specific transformation. Additionally, the theorem as stated is too weak to prove inductively, but a stronger one cannot be stated that holds for all transformations. Each transformation must define a strengthened form of the theorem such that there exists some relation between f_{t_1} and f_{t_2}, b_{t_1} and b_{t_2}, γ_1 and γ_2, and s_{t_1} and s_{t_2}. This allows $(n+1)$-th step of evaluation to flow directly from the n-th step of evaluation of f and a symmetric $(n'+1)$-th step of evaluation to flow directly from a symmetric n'-th step of evaluation of f_{out}.

Each transformation will also define a binary relation *trans* between functions, blocks, and streams in the evaluation of f and the symmetric element in the evaluation of f_{out}.

This relation, in part, defines the strengthened induction. As the bulk of this translation is consistent across transformations, we provide the following as a template. Portions of these relations may be rewritten when necessary. However, all refined translations must retain the property that if the functions and blocks conform to the relationships $f_{t_2} = \mathit{trans}(f_{t_1})$ and $b_{t_2} = \mathit{trans}(\langle f_{t_1}, b_{t_1} \rangle)$, then the streams conform to the relationship $\mathit{stream}(b_{t_2}) = \mathit{trans}(\langle f_{t_1}, b_{t_1}, b_{t_1} \rangle)$.

$$\mathit{trans}_\rho(f_t) = \begin{cases} f_{out}[\overrightarrow{\sigma_{f_t}}] & f_t \in \mathcal{C}(f) \\ f_t & \mathit{otherwise} \end{cases}$$

$$\mathit{trans}_\rho(\langle f_t, b_t \rangle) = b_t$$

$$\mathit{trans}_\rho(\langle f_t, b_t, (s_t, \hat{s}_t) \rangle) = s_t, \mathit{trans}_\rho(\hat{s}_t)$$

$$\mathit{trans}_\rho(\langle f_t, b_t, r_t, s_t \rangle_{\rho'}) = \langle \mathit{trans}_{\rho'}(f_t), \mathit{trans}_{\rho'}(b_t), r_t, \mathit{trans}_{\rho'}(\langle f_t, b_t, s_t \rangle) \rangle$$

$$\mathit{trans}_\rho(\epsilon) = \epsilon$$

The subscript denoting the relevant evaluation path may be omitted when it is not useful in a particular transformation and, as it does not need to be explicitly maintained during evaluation, does not appear elsewhere in the proof. For induction in the reverse direction, trans^{-1} translates certain elements of function f_{out} into the symmetric element of f and may also be rewritten when necessary. The default relations for the inverse are identical to the cases above.

6.2.2 Structural Theorems

The remaining theorems, defined below, state that transformations maintain canonical form, the well-typed property, the dominator tree, and the loop nesting forest of the function that undergoes the transformation. These properties trivially compose as well. Many transformations may not always require each precondition (in which case they may be omitted in

the proof statement).

Theorem Prototype 6.2.2. If f is in canonical form, then f_{out} is in canonical form.

Theorem Prototype 6.2.3. If f is in LCSSA form, then f_{out} is in LCSSA form.

Theorem Prototype 6.2.4. If $p \mid f$ is well-typed and f is in SSA form, then $p[f/f_{out}] \mid f_{out}$ is well-typed.

Theorem Prototype 6.2.5. The unique dominator tree of $G_{f_{out}}$ is D_{out}.

Theorem Prototype 6.2.6. If f is in canonical form, then F_{out} reconstructed from the triple $(H_{out}, L_{out}, X_{out})$ is the unique loop nesting forest of $G_{f_{out}}$.

We also make the following observation in order to simplify the proofs that types are preserved after a transformation.

Lemma 6.2.7. If $f : t$ and $f_{out} : t$, then $p\ f' \mid \Gamma \vdash s$ and $p[f/f_{out}]\ f' \mid \Gamma \vdash s$ have the same derivation tree.

Proof. The program parameter of a typing derivations is used only to determine the types of function references, which do not change between p and $p[f/f_{out}]$. □

Corollary 6.2.8. By Lemma 6.2.7, it follows that in order to prove Theorem Prototype 6.2.4 it is sufficient to show that if $p \mid f$ is well-typed and f is in SSA form, then $p \mid f_{out}$ is well-typed.

Appendix

6.A Proof Template for Maintenance of Evaluation

In this section, we supply a proof *template* to show that the semantics of a function which has undergone a transformation of the following form does not change.

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \rightarrow (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Theorem Template 6.A.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa). See Theorem Prototype 6.2.1 for the full theorem statement and additional assumptions made about the process of evaluation.

Proof. We prove by induction on n in the forward case and on n' in the reverse case.

$\rightarrow$ **Forward Case:** The following relationships between the resulting functions, blocks, and streams after a step of evaluation of f and a parallel step of evaluation of f_{out}.

$$f_{t_2} = trans_\rho(f_{t_1}) \qquad b_{t_2} = trans_\rho(\langle f_{t_1}, b_{t_1} \rangle) \qquad s_{t_2} = trans_\rho(\langle f_{t_1}, b_{t_1}, s_{t_1} \rangle)$$

[**Slot $\rightarrow$ Refinement**] Each transformation refines the *trans* relation such that the functions, blocks, streams, blocks, and return contexts referenced by the evaluation of f can be

translated into symmetric values referenced by the evaluation of f_{out}. Each transformation also places an additional restriction on the register contexts (e.g. ensuring that registers have the same value if defined or that a point of evaluation of f_{out} defines a superset of registers for any symmetric evaluation point in f).

[Slot → Invokability] For functions f and f_{out} to be semantically equivalent, they must be symmetrically invokable. Let $f_{t_1} \in \mathcal{C}(f)$, $\mathit{entry}(f_{t_1}) = b_{t_1}$, and $\mathit{dom}(\mathit{param}(b_{t_1}) = \langle \overline{r_{t_i}} \rangle$. Then, a clone of function f is invokable with the following form.

$$\rightarrow_\rho \ (\langle p, f_{t_1}, b_{t_2} \rangle \mid \gamma_1[\overline{r_{t_i} \mapsto v_{t_i}}] \mid \mu \mid \nu \mid \Psi;\ \mathit{stream}(b_{t_1})), \hat{s}_{t_1})$$

Let $f_{t_2} = f_o ut[\overrightarrow{\sigma_{f_{t_1}}}]$, $\mathit{entry}(f_{t_2}) = b_{t_2}$, and $\mathit{dom}(\mathit{param}(b_{t_2}) = \langle \overline{r'_{t_i}} \rangle$. Then, each transformation must show that given the following invocation, there exists a sequence of zero or more steps that brings the following evaluation of f_{out} to a state where the function, block, streams, and register contexts are consistent with the strengthened induction.

$$\rightarrow \quad (\langle p[f/_{f_{out}}], f_{t_2}, b_{t_2} \rangle \mid \gamma_2[\overline{r'_{t_i} \mapsto v_{t_i}}] \mid \mu \mid \nu \mid \Psi;\ \mathit{stream}(b_{t_2}), \mathit{trans}(\hat{s}_{t_1}))$$

→ Base Case ($n = n' = 1$): The base case is trivially implied by [Slot *Invokability*] where $\rho = \langle\rangle$ and $\gamma_1 = \gamma_2 = \varnothing$.

→ Inductive Step: We assume a non-terminal n-step evaluation of f and a parallel n'-step evaluation of f_{out} consistent with the strengthened induction. We show that if evaluation of f can make a step of progress, then evaluation of f_{out} can progress zero or more steps to reach a point that is consistent with the strengthened induction. The $(n+1)$-th step of evaluation of f must be possible according to Theorem 3.A.10, as the stream s_{t_1} is both non-terminal and, by assumption, well-typed with respect to its register context.

Asymmetric Evaluation We first consider the cases where the n-th step of evaluation of f and the n'-th step of evaluation of f_{out} diverge as evaluation of f_{out} has entered an area of the function which was modified by the transformation.

[**Slot → Asymmetric Evaluation**] Each transformation must show that for every pair of stream locations which can occur on the n-th step of evaluation of f and a parallel n'-th step of evaluation of f_{out}, the evaluation of f_{out} can progress to a point that is consistent with the strengthened induction defined by [Slot *Refinement*] with respect to the $(n+1)$-th step of evaluation of f.

Symmetric Evaluation Now, we consider the remaining cases where the next steps of progress occur symmetrically such that both evaluations make a step of progress with the same evaluation rule. Assume there exists an n-step evaluation of f with the following form.

$$\begin{aligned}(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \rightarrow_{\rho}^{n}\ & (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ s_{t_1}, \hat{s}_{t_1}) \\ \rightarrow\ & (\langle p, f'_{t_1}, b'_{t_1}\rangle \mid \gamma'_1 \mid \mu'' \mid \nu'' \mid \Psi'';\ s'_{t_1})\end{aligned}$$

By our strengthened inductive hypothesis, the n'-th step of evaluation of f_{out} stops with symmetric instruction stream, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction defined by [Slot *Refinement*]. Here, $f_{t_2} = trans_{\rho}(f_{t_1})$.

$$(p[f/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \rightarrow^{n'} (\langle p[f/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2 \mid \mu' \mid \nu' \mid \Psi';\ s_{t_2}, \hat{s}_{t_2})$$

After the exhaustion of the asymmetric cases in [Slot *Asymmetric Evaluation*], the remaining cases must obey the following relationships.

$$trans_{\rho}(f_t) = f_t \qquad trans_{\rho}(\langle f_t, b_t\rangle) = b_t \qquad trans_{\rho}(\langle f_t, b_t, (s_t, \hat{s}_t)\rangle\rangle = s_t, trans(\hat{s}_t)$$

$$((r, cv) \in \gamma_1 \implies (r, cv) \in \gamma_2) \text{ for every register } r \in defs_f(b_{t_1}) \cup uses_f(b_{t_1})$$

$$\hat{s}_{t_2} = (trans(f'_{t_1}), trans(\langle f'_{t_1}, b'_{t_1}\rangle), r, trans(s'_{t_1})) \text{ if } \hat{s}_{t_1} = (f'_{t_1}, b'_{t_1}, r, s'_{t_1})$$

We can then offload the work of proving symmetric evaluation onto five common lemmas defined in Section 6.B. One such lemma can be applied to the n-th and $(n+1)$-th steps of evaluation of f and the n'-th step of evaluation of f_{out} given that the following conditions hold: stream s_{t_1} and $trans(\langle f_{t_1}, b_{t_1}\rangle, s_{t_1})$ have the same leading instruction or terminator; every register in the leading component of s_{t_1} has the same value in γ_1 and γ_2; blocks $param(block_f(\ell_b)) = param(block_{f_{out}}(\ell_b))$ when control transfers to $block_f(\ell_b)$; and functions f and f_{out} have identical entry block parameters, or f cannot be called from this callsite. The first two conditions are true by the assumed relationships between streams and register contexts. The following conditions are assumed to be true as the transfer of control to a modified block and a call to the transformed function in which the parameters of the entry block are modified are considered asymmetric cases and were covered previously.

Now, we show that the $(n'+1)$-th step of evaluation of f_{out} resulting from application of one of the five symmetric evaluation lemmas is consistent with the strengthened induction (with respect to the $(n+1)$-th step of evaluation of f) defined by [Slot *Refinement*]. Application of any of these lemmas allows us to bound the parallel relationships between step $n/(n+1)$ and step $n'/(n'+1)$'s functions, blocks, register context, and streams as follows.

Symmetric Instructions By application of Lemma 6.B.1, the functions and blocks move in identical fashion: $f'_{t_1} = f_{t_1}$, $f'_{t_2} = f_{t_2}$, $b'_{t_1} = b_{t_1}$, and $b'_{t_2} = b_{t_2}$. The input and output functions are equivalent, as are the input and output blocks and return contexts. As the input functions, blocks, and return contexts were consistent with the strengthened induction, so are the output functions, blocks, and return contexts. The resulting streams are either the input streams with their head elements removed or both ϵ. In the former case, the tails of both streams remain consistent with the strengthened induction as the input streams are equivalent and consistent with the strengthened induction. In the latter case, ϵ streams are

consistent with the strengthened induction as the translation of empty streams and empty return contexts are unrefined for all transformations described in this work.

Symmetric Function Call We discuss the result of the application of Lemma 6.B.2. First, we consider the case where $f'_{t_1} \in \mathcal{C}(f)$. Then, $f'_{t_2} \in \mathcal{C}(f_{out})$ by translation, and slot [Slot *Invokability*] shows how additional steps of progress can be taken to advance the evaluation of f_{out}. In all other cases, the resulting functions and blocks are equivalent (by exhaustion of the asymmetric cases). Then, the resulting blocks and their unmodified streams are consistent with the strengthened induction.

Symmetric Branch By application of Lemma 6.B.3, $f'_{t_1} = f_{t_1}$ and $f'_{t_2} = f_{t_2}$. The input and output functions are equivalent, as are the input and output return contexts. As the input functions and return contexts were consistent with the strengthened induction, so are the output functions and return contexts. The resulting blocks share the same label and are equivalent in both functions (by exhaustion of the asymmetric cases). Then, the resulting blocks and their unmodified streams are consistent with the strengthened induction.

Symmetric Return (Non-Empty) By application of Lemma 6.B.4, $\hat{s}_{t_1}$ has the form $\langle f'_{t_1}, b'_{t_1}, r, s'_{t_1} \rangle$ and $\hat{s}_{t_2}$ has the form $\langle f'_{t_2}, b'_{t_2}, r, s'_{t_2} \rangle$. By assumption, the functions, blocks, streams, and resulting return contexts are consistent with the strengthened induction.

Symmetric Return (Empty) By application of Lemma 6.B.5, $f'_{t_1} = f_{t_1}$, $f'_{t_2} = f_{t_2}$, $b'_{t_1} = b_{t_1}$, and $b'_{t_2} = b_{t_2}$. As stated in the first case, these components continue to be consistent with the strengthened induction. Again, empty streams are consistent with the strengthened induction as the translation of empty streams is unrefined by assumption.

After application of any lemma, register contexts are constrained such that both of the

following conditions hold for some (possibly empty) set of registers $\{\overline{r_i}\}$ and a matching set of values $\{\overline{v_i}\}$.

$$\gamma_1' = \gamma_1[\overline{r_i \mapsto v_i}] \qquad \gamma_2' = \gamma_2[\overline{r_i \mapsto v_i}]$$

Simply, both register contexts gain a symmetric set of register-value pairs. As register contexts γ_1 and γ_2 are consistent with the strengthened induction, the resulting contexts are consistent with the strengthened induction as well (by exhaustion of the asymmetric cases, which include the cases where the register relationship is otherwise constrained).

This concludes the forward case.

$\leftarrow$ **Forward Case:** The following relationships between the resulting functions, blocks, and streams after a step of evaluation of f_{out} and a parallel step of evaluation of f.

$$f_{t_1} = trans_{\rho'}^{-1}(f_{t_2}) \qquad b_{t_1} = trans_{\rho'}^{-1}(\langle f_{t_2}, b_{t_2}\rangle) \qquad s_{t_1} = trans_{\rho'}^{-1}(\langle f_{t_2}, b_{t_2}, s_{t_2}\rangle)$$

[**Slot $\leftarrow$ Refinement**] Each transformation refines the $trans^{-1}$ relation such that the functions, blocks, streams, blocks, and return contexts referenced by the evaluation of f_{out} can be translated into symmetric elements referenced by the evaluation of f. Again, each transformation also places an additional restriction on the register contexts.

[**Slot $\leftarrow$ Invokability**] Let $f_{t_2} \in \mathcal{C}(f_{out})$, $entry(f_{t_2}) = b_{t_2}$, and $dom(param(b_{t_2}) = \langle\overline{r_{t_i}}\rangle$. Then, a clone of function f_{out} is invokable with the following form.

$$\rightarrow_{\rho'} \ (\langle p[f/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r_{t_i} \mapsto v_{t_i}}] \mid \mu \mid \nu \mid \Psi;\ stream(b_{t_2})), \hat{s}_{t_2})$$

Let $f_{t_1} = f[\overrightarrow{\sigma_{f_{t_2}}}]$, $entry(f_{t_1}) = b_{t_1}$, and $dom(param(b_{t_1}) = \langle\overline{r'_{t_i}}\rangle$. Then, each transformation must show that given the following invocation, there exists a sequence of zero or more

steps that brings the following evaluation of f to a state where the function, block, streams, and register contexts are consistent with the strengthened induction.

$$\rightarrow \quad (\langle p, f_{t_1}, b_{t_1} \rangle \mid \gamma_1[\overline{r'_{t_i} \mapsto v_{t_i}}] \mid \mu \mid \nu \mid \Psi;\ \mathit{stream}(b_{t_1}), \mathit{trans}^{-1}(\hat{s}_{t_2}))$$

$\leftarrow$ **Base Case ($n = n' = 1$):** The base case is trivially implied by [Slot *Invokability*] where $\rho' = \langle\rangle$ and $\gamma_1 = \gamma_2 = \varnothing$.

$\leftarrow$ **Inductive Step:** We assume a non-terminal n'-step evaluation of f_{out} and a parallel n-step evaluation of f consistent with the strengthened induction. We show that if evaluation of f_{out} can make a step of progress, then evaluation of f can progress zero or more steps to reach a point that is consistent with the strengthened induction. The $(n'+1)$-th step of evaluation of f_{out} must be possible according to Theorem 3.A.10 as the stream s_{t_2} is both non-terminal and, by assumption, well-typed with respect to its register context.

Asymmetric Evaluation We first consider the cases where the n'-th step of evaluation of f_{out} and the n-th step of evaluation of f are diverge as evaluation of f_{out} has entered an area of the function which was modified by the transformation.

[**Slot $\leftarrow$ Asymmetric Evaluation**] Each transformation must show that for every pair of stream locations which can occur on the n'-th step of evaluation of f_{out} and a parallel n-th step of evaluation of f, the evaluation of f can progress to a point that is consistent with to the strengthened induction defined by [Slot *Refinement*] with respect to the $(n'+1)$-th step of evaluation of f_{out}.

Symmetric Evaluation This section mirrors the forward case (with elements related to functions f and f_{out} in the opposite position).

This concludes the reverse case and the proof template. □

6.B Symmetric Evaluation

The following five lemmas, referenced heavily by Template 6.A.1, perform the bulk of the work in proving that an unaffected segment of the program maintains its semantics after transformation.

Each lemma assumes that two functions, f and f_{out}, are being evaluated in parallel. While these names imply f_{out} is the result of a transformation of f, this is not necessarily the case (it is true when applied in the forward case of Template 6.A.1, but untrue when applied in the reverse case).

6.B.1 Symmetric Instructions

The first lemma shows that symmetric evaluation of streams up to a point where they lead with the same instruction, excluding non-intrinsic function calls, can make steps of evaluation similar to one another. This requires additionally that both register contexts γ_1 and γ_2 cause the instruction to produce the same result.

Lemma 6.B.1. Assume there exists an $(n+1)$-step evaluation of f with the following form.

$$\begin{aligned}(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \rightarrow^n\ & (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ I, s_{t_1}) \\ \rightarrow\ & (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1' \mid \mu'' \mid \nu'' \mid \Psi'';\ s_{t_1}')\end{aligned}$$

If there exists an n'-step evaluation of function f_{out} symmetric to the first n steps of the evaluation of f, $p[f/f_{out}] \mid f_{out}$ is well-typed, and $\gamma_2(e) = \gamma_1(e)$ for every expression e that occurs in I, then an $(n'+1)$-th step of evaluation of function f_{out} exists with the following

form.

$$(p[f/f_{out}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \rightarrow^{n'} (\langle p[f/f_{out}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2 \mid \mu' \mid \nu' \mid \Psi';\ I, s_{t_2})$$
$$\rightarrow (\langle p[f/f_{out}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2' \mid \mu'' \mid \nu'' \mid \Psi'';\ s'_{t_2})$$

Additionally, each statement below holds for some register r defined by I and some value v.

- $s'_{t_1} = s_{t_1}$ and $s'_{t_2} = s_{t_2}$, or $s'_{t_1} = s'_{t_2} = \epsilon$
- $\gamma'_1 = \gamma_1$ and $\gamma'_2 = \gamma_2$, or $\gamma'_1 = \gamma_1[r \mapsto v]$ and $\gamma'_2 = \gamma_2[r \mapsto v]$

Proof. We prove by case analysis on s_{t_1}. Unless otherwise stated, assume that $\gamma'_1 = \gamma_1$, $\gamma'_2 = \gamma_2$, $\mu'' = \mu'$, $\nu'' = \nu'$, and $\Psi'' = \Psi'$.

Case ($r \leftarrow \texttt{move}(e), s'$). The $(n+1)$-th step of evaluation of f follows by application of rules E-Inst and E-Move such that $s'_{t_1} = s_{t_1}$ and $\gamma'_1 = \gamma_1[r \mapsto \gamma_1(e)]$. Symmetrically, the $(n'+1)$-th step of evaluation of f_{out} follows by the same rules (by Theorem 3.A.10) such that $s'_{t_2} = s_{t_2}$ and $\gamma'_2 = \gamma_2[r \mapsto \gamma_1(e)[\texttt{ref}\ /\texttt{ref}\ f_{out}]]$ as $\gamma_2(e) = \gamma_1(e)$ by assumption.

Case ($r \leftarrow \oplus(\overline{e_i}), s'$). We first handle the non-abnormal step of evaluation. The $(n+1)$-th step of evaluation of f follows by application of rules E-Inst and E-Operator such that $s'_{t_1} = s_{t_1}$ and $\gamma'_1 = \gamma_1[r \mapsto v]$. Symmetrically, the $(n'+1)$-th step of evaluation of f_{out} follows by the same rules (by Theorem 3.A.10) such that $s'_{t_2} = s_{t_2}$ and $\gamma'_2 = \gamma_2[r \mapsto v]$. Notice that as the operators are defined over only integer and boolean values, so the value v is yielded with respect to either register context γ_1 or γ_2. We now handle the division by zero case. The $(n+1)$-th step of evaluation of f follows by application of rule E-Div-Abnormal such that $s'_{t_1} = \epsilon$ and $\Psi'' = \Psi', \texttt{halt}(\textbf{ex}(\text{math err}))$. Symmetrically, the $(n'+1)$-th step of evaluation of f_{out} follows by the same rule (by Theorem 3.A.10) such that $s'_{t_2} = \epsilon$ and $\Psi''' = \Psi', \texttt{halt}(\textbf{ex}(\text{math err}))$.

Case ($r \leftarrow \texttt{addr}(\ell), s'$). First, suppose ℓ refers to a function $f_t \in p$. Let $f'_t = f_t$ if $f_t \neq f$ and let $f'_t = f_{out}$ otherwise. Notice that $f'_t \in p[f/f_{out}]$. The $(n+1)$-th step of evaluation of f and the $(n'+1)$-th step of evaluation of f_{out} follow by application of rules E-Inst and E-Addr such that $s'_{t_1} = s_{t_1}$, $s'_{t_2} = s_{t_2}$, $\gamma'_1 = \gamma_1[r \mapsto \texttt{ref}\ f_t]$, and $\gamma'_2 = \gamma_2[r \mapsto \texttt{ref}\ f'_t]$. Now, suppose ℓ refers to an intrinsic $\hat{f}$ which is addressable by both program p and $p[f/f_{out}]$. The $(n+1)$-th step of evaluation of f and the $(n'+1)$-th step of evaluation of f_{out} follow by application of rules E-Inst and E-Addr-Intrinsic such that $s'_{t_1} = s_{t_1}$, $s'_{t_2} = s_{t_2}$, $\gamma'_1 = \gamma_1[r \mapsto \texttt{ref}\ \hat{f}]$, and $\gamma'_2 = \gamma_2[r \mapsto \texttt{ref}\ \hat{f}]$.

Case ($r \leftarrow \texttt{load«}t\texttt{»}(e), s'$). By inversion of rule T-Load, e is an integer value with respect to both register contexts and $\gamma_1(e) = \gamma_2(e) = v$. First, suppose $v' = \mu'(v, \mathit{width}(t))$ suppose is interpretable as a value of type t with respect to program p. The $(n+1)$-th step of evaluation of f follows by application rules E-Inst and E-Load such that $s'_{t_1} = s_{t_1}$ and $\gamma'_1 = \gamma_1[r \mapsto v']$. As the widths of each type can be made the same in program p and $p[f/f_{out}]$, $v' = \mu'(v, \mathit{width}(t))$ is interpretable as a value of type t with respect to program $p[f/f_{out}]$. Then, the $(n'+1)$-th step of evaluation of f_{out} follows symmetrically by the same rules (by Theorem 3.A.10) such that $s'_{t_2} = s_{t_2}$ and $\gamma_2[r \mapsto v']$. Now, suppose $\mu'(v, \mathit{width}(t))$ cannot be interpreted as a value of type t with respect to program p. Then, the $(n+1)$-th step of evaluation of f and the $(n'+1)$-th step of evaluation of f_{out} follow by application of rule E-Load-Abnormal such that $s'_{t_1} = \epsilon$ and $s'_{t_2} = \epsilon$, and $\Psi'' = \Psi', \texttt{halt}(\textbf{ex}(\text{mem err}))$ and $\Psi''' = \Psi', \texttt{halt}(\textbf{ex}(\text{mem err}))$.

Case ($r \leftarrow \texttt{store}(e_1, e_2), s'$). By inversion of rule T-Load, e_1 is an integer value with respect to both register contexts and $\gamma_1(e_1) = \gamma_2(e_1) = v$. Additionally, the values $\gamma_1(e_2)$ and $\gamma_2(e_2)$ are necessarily representable by the same length w bitpattern. First suppose that $v \geq 0$. Then, the $(n+1)$-th step of evaluation of f and the $(n'+1)$-th step of evaluation of

f_{out} follow by application of rules E-Inst and E-Store such that $s'_{t_1} = s_{t_1}$, and $s'_{t_2} = s_{t_2}$, $\gamma'_1 = \gamma_1[r \mapsto \texttt{unit}]$ and $\gamma'_2 = \gamma_2[r \mapsto \texttt{unit}]$, and $\mu' = \mu[v \mapsto \gamma_1(e_1)]$. Now, suppose $v < 0$. Then, the $(n+1)$-th step of evaluation of f and the $(n'+1)$-th step of evaluation of f_{out} follow by application of rule E-Store-Abnormal such that $s'_{t_1} = \epsilon$ and $s'_{t_2} = \epsilon$, and $\Psi'' = \Psi', \texttt{halt}(\textbf{ex}(\text{mem error}))$ and $\Psi''' = \Psi', \texttt{halt}(\textbf{ex}(\text{mem err}))$.

Case ($r \leftarrow \texttt{call}(e_0, \overline{e_i}), s'$). As the $(n+1)$-th step of evaluation of f remains in the same function and block, it must be the case that $\gamma_1(e_0) = \gamma_2(e_0) = \hat{f}$. In the non-halting case, let $\hat{f}(\mu', \nu', \overline{\gamma_1(e_i)}) = (\mu'', \nu'', v)$. Notice that as $\gamma_2(e_i) = \gamma_1(e_i)$. In the abnormal intrinsic case, the $(n+1)$-th step of evaluation of f and the $(n'+1)$-th step of evaluation of f_{out} follow by application of rule E-Intrinsic-Abnormal such that $s'_{t_1} = \epsilon$ and $s'_{t_2} = \epsilon$, and $\Psi'' = \Psi', \hat{f}(\overline{\gamma_1(e_i)}), \texttt{halt}(\textbf{ex}(\text{err}))$ and $\Psi''' = \Psi', \hat{f}(\overline{\gamma_2(e_i)}), \texttt{halt}(\textbf{ex}(\text{err}))$. □

6.B.2 Symmetric Function Calls

This next lemma proves a similar condition for evaluation of calls to non-intrinsics. This requires the same consistency condition for registers contexts γ_1 and γ_2 as well as requiring that the entry block of the target function has the same parameters in the evaluation of both functions f and f_{out}.

Lemma 6.B.2. Assume there exists an $(n+1)$-step evaluation of f with the following form. Let function $f'_t \in p$ refer to the original target of the call instruction such that $f'_{t_1} \in \mathcal{C}(f'_t)$ and $b'_{t_1} = \mathit{entry}(f'_{t_1})$.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \rightarrow^n$$

$$(\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ r \leftarrow \texttt{call}(e_0, \overline{e_i}), s_{t_1}) \rightarrow$$

$$(\langle p, f'_{t_1}, b'_{t_1}\rangle \mid \gamma_1[\overline{r_i \mapsto v_i}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b'_{t_1}), \langle f_{t_1}, b_{t_1}, r, s_{t_1}\rangle)$$

If there exists an n'-step evaluation of function f_{out} symmetric to the first n steps of the evaluation of f, $p[f/f_{out}] \mid f_{out}$ is well-typed, $\gamma_2(e) = \gamma_1(e)$ for every expression e that occurs in the leading `call` instruction, and $param(entry(f)) = param(entry(f_{out}))$ (unless function f cannot be called from this callsite), then an $(n'+1)$-th step of evaluation of function f_{out} exists with the following form such that $f'_{t_2} = f'_{t_1}$ when $f'_{t_1} \notin \mathcal{C}(f)$, $f'_{t_2} = f_{out}[\overrightarrow{\sigma_{f'_{t_1}}}]$ when $f'_{t_1} \in \mathcal{C}(f)$, and $b'_{t_2} = entry(f'_{t_2})$.

$$(p[f/f_{out}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \rightarrow^{n'}$$

$$(\langle p[f/f_{out}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2 \mid \mu' \mid \nu' \mid \Psi';\ r \leftarrow \texttt{call}(e_0, \overline{e_i}), s_{t_2}) \rightarrow$$

$$(\langle p[f/f_{out}], f'_{t_2}, b'_{t_2}\rangle \mid \gamma'_2[\overline{r_i \mapsto v_i}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b'_{t_2}), \langle f_{t_2}, b_{t_2}, r, s_{t_2}\rangle)$$

Proof. As $\gamma_1(e_0) = \texttt{ref}\ f'_t$, $\gamma_2(e_0)$ must be equal to either $\texttt{ref}\ f'_t$ when $f'_t \neq f$ or $\texttt{ref}\ f_{out}$ otherwise. In the former case, the params of the entry block of f'_t are identical in both evaluations. In the latter case, the params of the entry blocks of f and f_{out} are equal by assumption. Then, the $(n'+1)$-th step of evaluation of f_{out} follows from application of rule E-Call. The target function is identical when the call does not target function f (as identical registers can be chosen for a clone of the non-modified function), and is equal to a clone of f_{out} where all common registers are chosen to be identical as described in Chapter 6. □

6.B.3 Symmetric Branch

Now, we show that symmetric evaluation of streams up to a point where they lead with the same terminator can make steps of evaluation similar to one another. We first cover branching, which is conceptually similar to the case of calls to non-intrinsics.

Lemma 6.B.3. Assume there exists an $(n+1)$-step evaluation of f with the following form.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{switch}\ e\ \overline{cv_i \mapsto ref_i}\ ref_d, \hat{s}_{t_1})$$
$$\to (\langle p, f_{t_1}, b'_{t_1}\rangle \mid \gamma_1[\overline{r_i \mapsto v_i}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b'_{t_1}), \hat{s}_{t_1})$$

If exists an n'-step evaluation of function f_{out} symmetric to the first n steps of the evaluation of f (shown below), $p[^f/_{f_{out}}] \mid f_{out}$ is well-typed, $\gamma_2(e) = \gamma_1(e)$ for every expression e that occurs in the `switch` terminator, and there is a block $b_{t_2} \in body(f_{out})$ such that $lab(b'_{t_1}) = lab(b'_{t_2})$ and $param(b'_{t_1}) = param(b'_{t_2})$, then an $(n'+1)$-th step of evaluation of function f_{out} exists with the following form.

$$(p[^f/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \to^{n'}$$
$$(\langle p[^f/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{switch}\ e\ \overline{cv_i \mapsto ref_i}\ ref_d, \hat{s}_{t_2}) \to$$
$$(\langle p[^f/_{f_{out}}], f_{t_2}, b'_{t_2}\rangle \mid \gamma_2[\overline{r_i \mapsto v_i}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b'_{t_2}), \hat{s}_{t_2})$$

Proof. Let $\ell(\overline{e_i})$ be the block reference chosen by rules E-Case and E-Case-Default in the evaluation of f. If $\gamma_1(e)$ is an integer or boolean value, then $\gamma_1(e) = \gamma_2(e)$ and the same block reference will be chosen by rule E-Case in the evaluation of f_{out}. If $\gamma_1(e)$ is a block reference or a unit value and the block reference was chosen by rule E-Case-Default as no value cv_i can match either of these values (syntax forbids function references and E-Case forbids matching unit values). Then, $\gamma_2(e)$ is also one of these two types and the same block reference is chosen by rule E-Case-Default. The $(n'+1)$-th step of evaluation of f_{out} then follows by application of rule E-Switch and Theorem 3.A.10. □

6.B.4 Symmetric Return

Finally, we show the same for streams that return up the call stack. We split this statement into the cases of streams with non-empty return context and streams with empty return

contexts. Purposefully absent is the case where one side is non-empty and the other is empty, which is a very obvious non-symmetry.

Lemma 6.B.4. Assume there exists an $(n+1)$-step evaluation of f with the following form.

$$\begin{aligned}(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n\ & (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{return}\ e, \langle f'_{t_1}, b'_{t_1}, r, s'_{t_1}\rangle) \\ \to\ & (\langle p, f'_{t_1}, b'_{t_1}\rangle \mid \gamma_1[r \mapsto v] \mid \mu' \mid \nu' \mid \Psi';\ s'_{t_1})\end{aligned}$$

If there exists an n'-step evaluation of function f_{out} symmetric to the first n steps of the evaluation of f, $p[^f/_{f_{out}}] \mid f_{out}$ is well-typed and $\gamma_2(e) = \gamma_1(e)$, then an $(n'+1)$-th step of evaluation of function f_{out} exists with the following form.

$$\begin{gathered}(p[^f/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \to^{n'} \\ (\langle p[^f/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{return}\ e, \langle f'_{t_2}, b'_{t_2}, r, s'_{t_2}\rangle) \to \\ (\langle p[^f/_{f_{out}}], f'_{t_2}, b'_{t_2}\rangle \mid \gamma'_2[r \mapsto v \mid \mu' \mid \nu' \mid \Psi';\ s'_{t_2})\end{gathered}$$

Proof. Trivial by application of rule E-Return and Theorem 3.A.10. □

Lemma 6.B.5. Assume there exists an $(n+1)$-step evaluation of f with the following form.

$$\begin{aligned}(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n\ & (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{return}\ e) \\ \to\ & (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi', \texttt{halt}(v);\ \epsilon)\end{aligned}$$

If there exists an n'-step evaluation of function f_{out} symmetric to the first n steps of the evaluation of f, $p[^f/_{f_{out}}] \mid f_{out}$ is well-typed, and $\gamma_2(e) = \gamma_1(e)$, then an $(n'+1)$-th step of evaluation of function f_{out} exists with the following form.

$$\begin{gathered}(p[^f/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \to^{n'} \\ (\langle p[^f/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{return}\ e) \to \\ (\langle p[^f/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2 \mid \mu' \mid \nu' \mid \Psi', \texttt{halt}(v);\ \epsilon)\end{gathered}$$

Proof. Trivial by application of rule E-Return-Last and Theorem 3.A.10. □

6.C Common Lemmas

In this section we present a lemma that is used by several transformation proofs. This lemma reduces the boilerplate, showing that a parallel step of evaluation can be taken in functions f and f_{out} if an instruction is evaluated and the streams differ only by the terminator.

Lemma 6.C.1. Assume there exists an $(n+1)$-step evaluation of f with the following form for which predicate P is true.

$$\begin{aligned}(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n\ & (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ I, s_{t_1}) \\ \to\ & (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1' \mid \mu'' \mid \nu'' \mid \Psi'';\ s_{t_1}')\end{aligned}$$

Let function f and f_{out} relate via a particular transformation. Let this transformation refine the translation of streams with the existence of the following property for some predicate P, and assume the translation of empty streams and empty return contexts are unrefined.

$$trans(\langle f_t, b_t, ((s_t', T_t), \hat{s}_t)\rangle) = \begin{cases} (s_t', T[\overrightarrow{\sigma_{f_t}}]), trans(\hat{s}_t) & P \\ & \vdots \end{cases}$$

Additionally, let this transformation define the relationship between register contexts such that $((r, cv) \in \gamma_1 \implies (r, cv) \in \gamma_2)$ for every register $r \in defs_f(b_{t_1}) \cup uses_f(b_{t_1})$. Then, if there exists an n'-step evaluation of function f_{out} symmetric to the first n steps of the evaluation of f, $p[f/f_{out}] \mid f_{out}$ is well-typed, then an $(n'+1)$-th step of evaluation of function f_{out} exists that is consistent with the strengthened induction.

Proof. By application of Lemma 6.B.1, the resulting blocks are the same as the input blocks, which are consistent with the strengthened induction by assumption, and the resulting streams are either the input streams with their head elements removed or both ϵ. In the former case, the tails of both streams remain consistent with the strengthened induction

by translation. In the latter case, ϵ streams are consistent with the strengthened induction by assumption. Additionally, the resulting register contexts either remain the same, or are extended with the same register-value pair, which does not violate the strengthened induction by assumption. $\square$

7 Canonicalization

A control flow graph can be *canonicalized* by manipulating the edges associated with a canonical form property violation, as demonstrated in Figure 7.01. In this example, the colored edges in the control flow graph violate a particular property of canonical form – the blue edges (l_1, h) and (l_2, h) violate (Property 1); the red edges (p_1, h) and (p_2, h) violate (Property 2); and the green edges (p_2, e_1) and (e_2, e_3) violate (Property 3). Empty blocks l' and h' can be inserted respectively onto the blue and red edges to form a dedicated preheader and a unique latch. The target of each green edge is an exit of the loop that also has a predecessor outside of the loop. Empty blocks e'_1 and e'_2 can be inserted onto the non-highlighted edges for each violating exit to form exits that have predecessors only in the loop they exit.

Section 7.1 presents a transformation used to repair a set of definitions that violate the single assignment property. Section 7.2 presents a transformation used to convert an SSA control flow graph into an LCSSA control flow graph. Section 7.3 presents a transformation

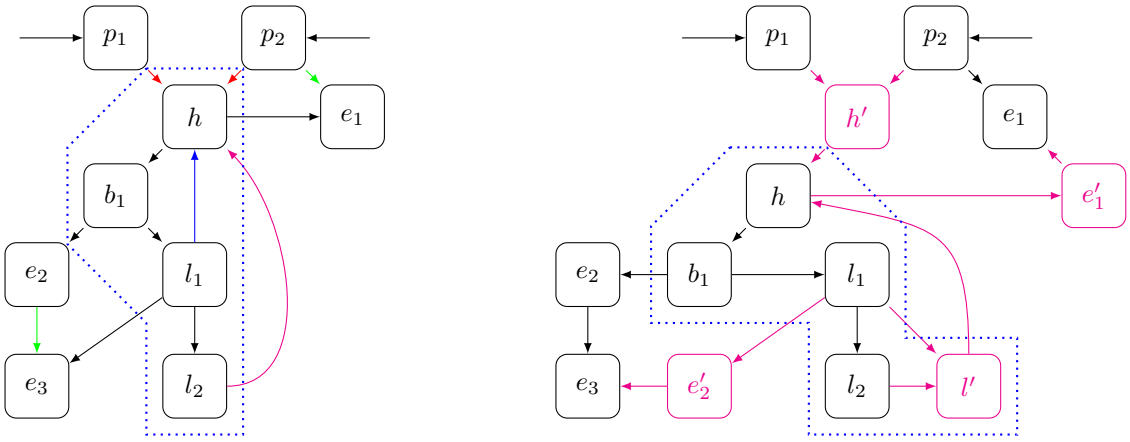


Figure 7.01: A (very) non-canonical control flow graph and its canonical equivalent.

$$\frac{b_{out} = b\left[{}^{param(b)}/_{param(b) \cup \langle (r\colon t) \rangle}\right]}{b \xrightarrow[(r\colon t)]{\text{add param}} b_{out}}$$

$$\frac{def_f(r) \in b \qquad b_{out} = b\left[{}^{\ell_{b'}(\overline{e})}/_{\ell_{b'}(\overline{e},r)} \mid b' \in B \text{ where } \ell_{b'} = lab(b')\right]}{b \xrightarrow[B,(r\colon t)]{\text{add arg}} b_{out}}$$

$$\frac{def_f(r) \notin b \qquad b\left[{}^{\ell_{b'}(\overline{e})}/_{\ell_{b'}(\overline{e},r)} \mid b' \in B \text{ where } \ell_{b'} = lab(b')\right] \xrightarrow[\{(r\colon t)\},\emptyset]{\text{repair implicits}} b_{out}}{b \xrightarrow[B,(r\colon t)]{\text{add arg}} b_{out}}$$

Figure 7.11: Inserting parameters to blocks and arguments to block references.

that places a fresh empty block at the end of the path shared by a set of edges. This transformation modifies the number or source of predecessors of a particular block and is useful in controlling the edges crossing loop boundaries. Section 7.4 presents transformations to repair specific canonical form violations. Unlike the transformations discussed in Chapter 9, canonicalization transformations do not assume that the input graph is already in canonical form.

7.1 SSA Reconstruction

A set of register definitions breaking SSA form can be rewritten so that each definition writes a uniquely named fresh register and all uses of the old register are replaced by the reaching definition at the point of use. This general idea was discussed in Section 5.4.

We first define a set of necessary operations for inserting additional block parameters and block reference arguments. These operations are formally defined in Figure 7.11 and should be unsurprising. Adding a block parameter simply adds an additional typed register to the

$$\frac{b_{out} = (lab(b), param(b), (implicit(b) \setminus P_-) \cup P_+, stream(b))}{b \xrightarrow[P_+,P_-]{\text{repair implicits}} b_{out}}$$

Figure 7.12: Adding and removing values from the set of implicit block parameters.

$$\frac{b' \in B \quad lab(b') = lab(b) \quad b' \xrightarrow[(r\colon t)]{\text{add param}} b_{out}}{B \xrightarrow[b,(r\colon t)]{\text{add param set}} B \setminus \{b\} \cup \{b_{out}\}} \qquad \frac{b' \in B \quad lab(b') = lab(b) \quad b' \xrightarrow[B',(r\colon t)]{\text{add arg}} b_{out}}{B \xrightarrow[b,B',(r\colon t)]{\text{add arg set}} B \setminus \{b\} \cup \{b_{out}\}}$$

Figure 7.13: Inserting parameters to blocks and arguments to block references identified by a label.

existing parameter sequence. Adding an argument to a block reference simply appends an expression to the tail of the reference's arg list – if this expression is a register that is not defined in block b, then the register must also be added to the set of b's implicit parameters. The *add arg* operation takes as input a set of blocks B in order to select the block references to be modified. The *repair implicits* operation, defined in Figure 7.12, is applied when references to new registers in a block or removing the last reference to a register from a block.

Figure 7.13 defines the operations *add param set* and *add argset*. These operations perform the same block transformations as *add param* and *add arg*, respectively, but instead of working directly on a block b, it finds a block b' from the set of blocks B with the same label. The resulting set of blocks, B_{out}, is the set B with block b' replaced by the transformed block. These operations allow us to add a set of block parameters and associated arguments to a program without the need to name intermediate blocks or sets of blocks.

SSA Reconstruction is performed by an operation that mirrors the *rewrite uses* algorithm

$$\frac{\begin{array}{c}\forall d_i \in def_f(r) \cdot typeof_f(d_i) = t \qquad f = (\ell, t, B, \ell_0) \qquad \sigma_R = \{(d_i, r_i) \mid d_i \in def_f(r)\} \\ r_i \text{ is fresh} \qquad \sigma_D = [(d_i, d_i') \mid d_i \in def_f(r) \text{ where } d_i' = d_i[{}^{r}/_{\sigma_R(d_i)}] \\ R_0 = B_{\phi_0} = \emptyset \qquad (R_{i-1}, B_{\phi_{i-1}}) \xrightarrow[\{(u, def_f(r)[\sigma_D]) \mid u \in use_f(r)\}]{\text{find def} \downarrow}{}^{*} (R_i, B_{\phi_i}) \qquad |use_f(r)| = k \\ B[\sigma_D] \xrightarrow[\{(b_i, (r_i : t)) \mid (b_i, r_i) \in B_{\phi_k}\}]{\text{add param set}}{}^{*} \xrightarrow[\{(b_i, \{b_i\}, R_k(p_i)) \mid p_i \in pred_f(b_i) \wedge b_i \in B_{\phi_k}\}]{\text{add arg set}}{}^{*} \xrightarrow[\{u, (r : t), R_k^u(b) \mid \langle b, u\rangle \in use_f(r)\}]{\text{replace register set}}{}^{*} B_{out} \\ f_{out} = (\ell, t, B', \ell_0) \qquad D_{out} = def_f(r)[\sigma_D] \cup \{\langle b_i, (r_i : t)\rangle \mid (b_i, r_i) \in B_{\phi_k}\}\end{array}}{f \xrightarrow[(r : t)]{\text{rewrite uses}} (f_{out}, D_{out})}$$

$$\frac{b \xrightarrow[(r' : t)]{\text{imp def}} P_+ \qquad b \xrightarrow[(r : t)]{\text{imp use}} P_- \qquad b[{}^{c}/_{c[r/r']}] \xrightarrow[P_+, P_-]{\text{repair implicits}} b_{out}}{b \xrightarrow[\langle b, c\rangle, (r : t), r']{\text{replace register}} b_{out}}$$

$$\frac{b' \in B \qquad lab(b') = lab(b) \qquad b' \xrightarrow[\langle b, c\rangle, (r : t), r']{\text{replace register}} b_{out}}{B \xrightarrow[\langle b, c\rangle, r, r']{\text{replace register set}} B \setminus \{b\} \cup \{b_{out}\}}$$

$$\frac{def_f(r) \in b}{b \xrightarrow[(r : t)]{\text{imp def}} \emptyset} \qquad \frac{def_f(r) \notin b}{b \xrightarrow[(r : t)]{\text{imp def}} \{(r : t)\}} \qquad \frac{|use_b(r)| > 1}{b \xrightarrow[(r : t)]{\text{imp use}} \emptyset} \qquad \frac{|use_b(r)| = 1}{b \xrightarrow[(r : t)]{\text{imp use}} \{(r : t)\}}$$

Figure 7.14: Rewriting uses of a register violating SSA form.

by Braun et al. [12] as discussed in Section 5.4. The differences here are mostly stylistic. Instead of inserting ϕ-nodes, we insert block parameters and block reference arguments. Instead of presenting the operation as pseudocode, we present it with inference notation to conform to the remainder of this work. The only functional difference in the algorithms is that the transformation presented here first replaces all definitions of a register r with a fresh register. After the algorithm has finished, no reference to the original register r remains.

Figure 7.14 formally defines this transformation. First, we create a mapping σ_D that replaces the definition of each register r with a fresh register. Then, we determine the

$$\frac{\langle f, b, d\rangle \in D \qquad d < c \qquad d \text{ defines } r \qquad R_{out} = R[b \mapsto^{c} r]}{(R, B_\phi) \xrightarrow[\langle f,b,c\rangle, D]{\text{find def} \downarrow} (R_{out}, B_\phi)} \qquad \frac{\langle f, b, d\rangle \notin D \vee d \not< c \qquad (R, B_\phi) \xrightarrow[\langle f,b\rangle, D]{\text{find def} \uparrow} (R_{out}, B_{\phi_{out}})}{(R, B_\phi) \xrightarrow[\langle f,b,c\rangle, D]{\text{find def} \downarrow} (R_{out}, B_{\phi_{out}})}$$

Figure 7.15: Finding the nearest reaching definition from a set of definitions D from the bottom of a block.

$$\frac{\langle f, b, d\rangle \in D \vee (b, d) \in R \qquad d \text{ defines } r \qquad R_{out} = R[b \mapsto r]}{(R, B_\phi) \xrightarrow[\langle f,b\rangle, D]{\text{find def} \uparrow} (R_{out}, B_\phi)}$$

$$\frac{\begin{array}{c}\langle f, b, d\rangle \notin D \wedge (b, d) \notin R \qquad P = \langle p_i \mid p_i \in pred_f(b)\rangle \qquad |P| = k \\ R_0 = R[b \mapsto r] \qquad B_{\phi_0} = B_\phi[b \mapsto r] \qquad r \text{ is fresh} \\ (R_{i-1}, B_{\phi_{i-1}}) \xrightarrow[\langle f,p_i \in P\rangle, D]{\text{find def} \uparrow} (R_i, B_{\phi_i}) \qquad (R_k, B_{\phi_k}) \xrightarrow[b,r]{\text{collapse params}} (R_{out}, B_{\phi_{out}})\end{array}}{(R, B_\phi) \xrightarrow[\langle f,b\rangle, D]{\text{find def} \uparrow} (R_{out}, B_{\phi_{out}})}$$

Figure 7.16: Finding the nearest reaching definition from a set of definitions D from the top of a block.

reaching definition for each use of r, as well as the reaching definition for all blocks along every path to the nearest definition of r. These definitions are yielded by the *find def* procedure, discussed further below, in the form of two sets: R_k and B_{ϕ_k}. The set R_k is a mapping from blocks to their reaching definition. Here, we use the following notation for convenience: $R_k^c(b) = d$ and $(b \mapsto^c d) \in R_k$ both denote a reaching definition for block b that occurs *before* the component c. When the superscript c is omitted, the mapping yields the last such reaching definition for that block. The set B_{ϕ_k} is a mapping from a block to typed registers. If $(b, r) \in B_{\phi_k}$, then two or more distinct reaching definitions at block b are disambiguated when $(r\colon t)$ is made a parameter of block b.

$$\frac{\exists o \cdot R \subseteq \{r, o\}}{(R, B_\phi) \xrightarrow[b,r]{\text{collapse params}} (R, B_\phi)}$$

$$\frac{\forall o \cdot R \not\subseteq \{r, o\} \qquad R_{out} = R[b \mapsto o][b' \mapsto o \mid (b', r) \in R] \qquad B_{\phi_{out}} = B_\phi[b \mapsto \varnothing]}{(R, B_\phi) \xrightarrow[b,r]{\text{collapse params}} (R_{out}, B_{\phi_{out}})}$$

Figure 7.17: Remove trivially-defined block parameters.

After reaching definitions are found, each block occurring in B_{ϕ_k} is given an additional block parameter such that each parameter value is the predecessor block's reaching definition of r, and the occurrences of register r in each use $u \in use_f(r)$ are replaced by the reaching definition of r at the program point where u occurs. This last step is done by application of *replace register* procedures that ensures the implicit parameters of the block remains well-formed as register references are added or removed to a block.

Along with the modified function f_{out}, the transformation yields a set of definitions D_{out} that are fresh to the function. This includes the modified original definitions of r, as well as each block parameter inserted from the mapping B_{ϕ_k}. This set will be of use in LCSSA reconstruction discussed in Section 7.2.

The majority work of this algorithm is performed in a set of *find def* procedures, formally defined in Figure 7.15, Figure 7.16, and Figure 7.17. This procedure is partitioned into *find def*$\downarrow$ (read: find def from bottom) and *find def*$\uparrow$ (read: find def from top). The bottom-up procedure takes a component c as a reference and ignores any definitions occurring *after* component c in the block. When c is a use of r, it prevents a register from being used within a block before it is defined. If such a definition occurs in the block, then the pair (b, r) is added to the mapping of reaching definitions R_{out} and no block parameters need to be

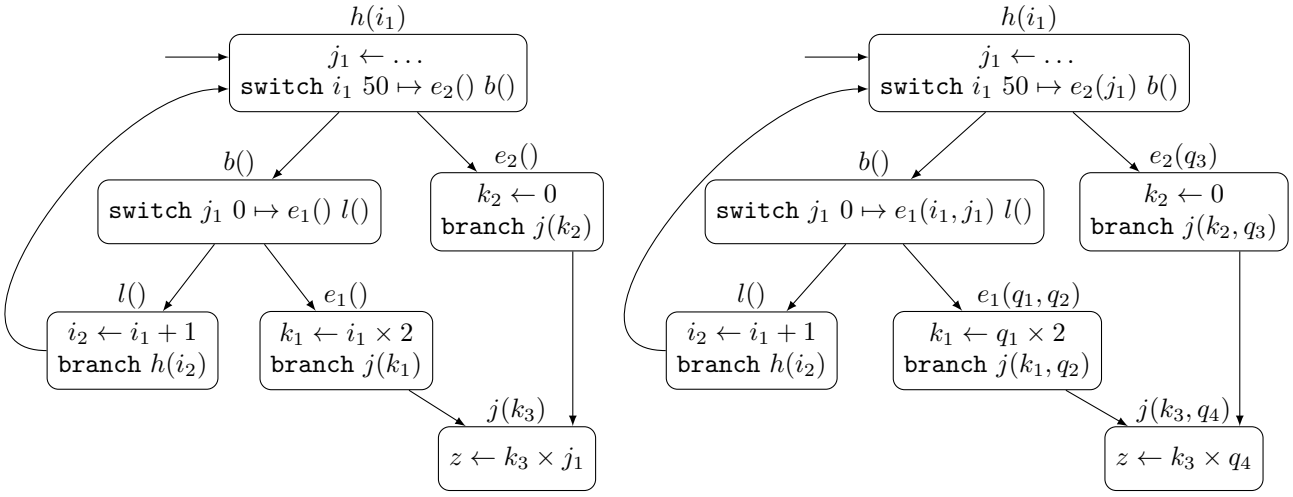


Figure 7.21: Insertion of block parameters at the loop boundary.

inserted. If no such definitions occur in the block, the bottom-up procedure falls back to a search from the top of the block. In this case, the pair (b, r) is added to the mapping of new block parameters B_ϕ and the reaching definition of each predecessor block is found (the temporary insertion of this block parameter is necessary to ensure termination). If the set of reaching definitions of the predecessors of this block contain exactly one definition apart from r denoted by o, then r is a spurious definition and can be removed (as r may have been referenced when the predecessor was searching for its reaching definition, each reference to r must also be replaced by the correct definition o). This cleanup step is done by the *collapse params* procedure in Figure 7.17.

7.2 LCSSA Reconstruction

A use of a register breaking LCSSA form can be eliminated by inserting additional ϕ-nodes or block parameters to break definition-use chains at the loop boundary and rewriting uses to reference the new reaching definitions. This process is demonstrated in Figure 7.21.

Registers i_1 and j_1 are both defined within but used outside of the loop composed of

$$\frac{l = loop_{F_f}(b) \qquad d \text{ defines } r \qquad \{u \in uses_f(r) \mid u \text{ occurs outside of } body_f(l)\} = \emptyset}{f \xrightarrow[\langle b,d\rangle]{\text{repair lcssa}} f}$$

$$\frac{\begin{array}{c} l = loop_{F_f}(b) \qquad d \text{ defines } r \qquad uses = \{u \in uses_f(r) \mid u \text{ occurs outside of } body_f(l)\} \\ uses \neq \emptyset \qquad d \text{ defines } r \qquad t = typeof_f(d) \qquad b_i \xrightarrow[(r\colon t)]{\text{add param}} b'_i \\ b_j \xrightarrow[exits_{old},(r\colon t)]{\text{add arg}} b'_j \qquad exits_{old} = \{e \in exit_f(l) \mid b \prec e\} \qquad exits_{new} = \{b'_i \mid b_i \in exits_{old}\} \\ preds_{old} = \bigcup_{b' \in exits_{old}} pred_f(b') \qquad preds_{new} = \{b'_j \mid b_j \in preds_{old}\} \\ f = (\ell, t, B, \ell_0) \qquad B' = B \setminus (exits_{old} \cup preds_{old}) \cup exits_{new} \cup preds_{new} \\ f' = (\ell, t, B', \ell_0) \qquad f' \xrightarrow[(r\colon t),\{\overline{\langle b_i,(r\colon t)\rangle}\},uses]{\text{rewrite uses}}{}^{*} (f'', \overline{D_i}) \qquad f'' \xrightarrow[\bigcup D_i]{\text{repair lcssa}}{}^{*} f_{out} \end{array}}{f \xrightarrow[\langle b,d\rangle]{\text{repair lcssa}} f_{out}}$$

Figure 7.22: Repairing LCSSA-violating uses of the register defined by d.

blocks h, b, and l. The link between the definition of i_1 in the header of the loop and its use in block e_1 is segmented by the insertion of the block parameter defining the fresh register q_1. A symmetric parameter is not necessary in e_2, as the block does not appear in any path from the definition of i_1 to one of its uses. The use of i_1 in e_1 can then be replaced by q_1.

The use of j_1 in this example appears more problematic, as both e_1 and e_2 appear on a path from the definition of j_1 to its use. This requires that a block parameter be placed on both exits, respectively defining q_2 and q_3. The use of j_1 can then be replaced by the reaching definitions of q_2 and q_3 (which, in turn, requires the insertion of an additional block parameter defining q_4 in j).

Figure 7.22 formally defines this transformation for the definition $\langle f, b, d\rangle$. If the definition does not occur within a loop, or if there are no uses of the defined register outside of the defining loop, the transformation does not modify the input. Otherwise, a fresh parameter is

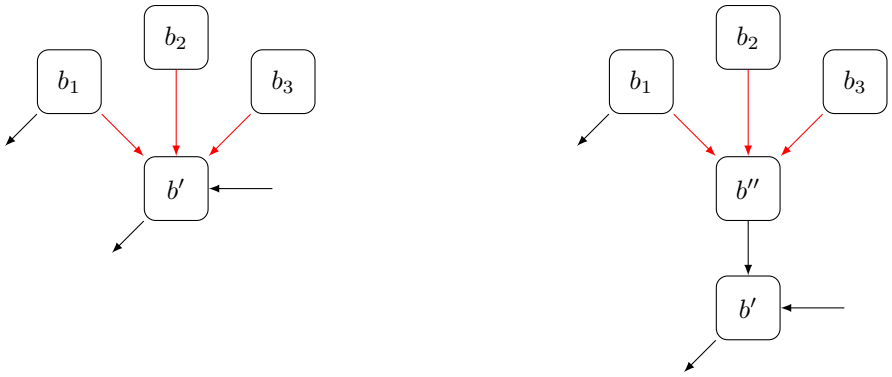


Figure 7.31: Splitting the red edge set containing $(b_1,\ b')$, $(b_2,\ b')$, and $(b_3,\ b')$.

placed on a subset of the exit blocks of the defining loop and the defined register is added to the argument list of references to these blocks. Then, each use outside of the loop is rewritten to refer to a reaching definition from the set of fresh parameters. The introduction of additional block parameters at the boundaries of a loop (or the block parameters inserted by SSA reconstruction) may themselves violate LCSSA form, so the same procedure is applied recursively to all new definitions. All new definitions are returned as a member of the set D_i yielded by the *rewrite uses* transformation.

7.3 Edge Set Splitting

A simple example of edge set splitting is illustrated in Figure 7.31. The set of edges being split, denoted E, originate from a set of blocks sharing a common terminal successor, denoted b'. An empty block b'' is inserted onto the edges such that b'' replaces b' in the edge set E. As b'''s only successor is b', the parameters of b'' can be isomorphic in number and type to the parameters of b'. During evaluation, b'' simply proxies the values from a successor of b'' to b' unchanged. Performing this transformation reduces the number of predecessors of b' by $|E| - 1$ blocks.

Figure 7.32 formally defines this transformation. All edges in the set E must have the

$$E \subseteq \overleftarrow{E}(G_f) \vee E \subseteq \overrightarrow{E}(G_f) \qquad lab(b') = \ell_{b'} \qquad \ell_{b''} \in \mathcal{L}^b \text{ is fresh} \qquad r'_i \text{ is fresh}$$

$$b'_i = b_i[\ell_{b'}/\ell_{b''}] \qquad b'' = (\ell_{b''}, \langle (r'_i \colon t_i) \mid (r_i \colon t_i) \in param(b') \rangle, \emptyset, \langle \texttt{branch}\ \ell_{b'}(\overline{r'_i}) \rangle)$$

$$f = (\ell, t, B, \ell_0) \qquad B' = (B \setminus dom(E)) \cup \{\overline{b'_i}\} \cup \{b''\} \qquad f_{out} = (\ell, t, B', \ell_0)$$

$$D_{out} = D_f[b'' \mapsto nca_D(dom(E))][b' \mapsto b'' \colon \{b \in pred_f(b') \mid b' \not\prec_f b\} = \{b''\}]$$

$$l = nca_{F_f}(\{loop_{F_f}(b) \mid b \in dom(E) \cup \{b'\}\}) \qquad H_{out} = H_{F_f}$$

$$L_{out} = L_{F_f}[b'' \mapsto l] \qquad exits = \{l' \in F_f \mid b' \in exit_f(l') \wedge dom(E) \cap body_f(l') \neq \emptyset\}$$

$$X_{old} = \{(\ell_{b'}, lab(l')) \mid l' \in exits \wedge (pred_f(b') \setminus dom(E)) \cap body_f(l') = \emptyset\})$$

$$X_{new} = \{(\ell_{b''}, lab(l')) \mid l' \in exits\} \qquad X_{out} = (X_{F_f} \setminus X_{old} \cup X_{new}$$

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[E=\{\overline{(b_i,b')}\}]{\text{split edge set}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Figure 7.32: Inserting a block onto each edge in the set E.

same direction – all edges must be either forward edges or backedges. If the set contains edges in both directions, the resulting graph may become irreducible, as illustrated in Figure 7.33. In this example, the introduction of block b'' creates a path into the strongly connected component $\{h, b, l_1, l_2, b''\}$ that bypasses the loop header.

The block b'' is introduced as a predecessor of b', and references to block b' from a block in $dom(E)$ are replaced with a reference to block b'' with the same parameters. The addition

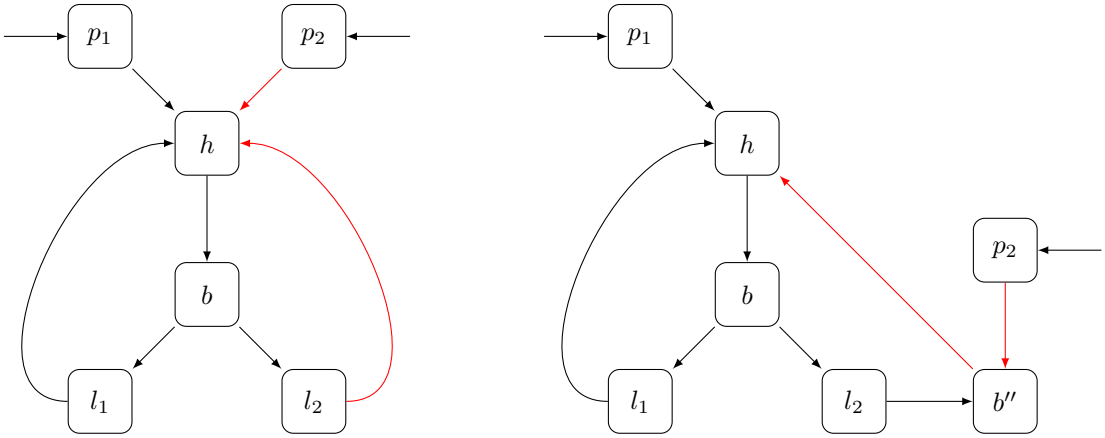


Figure 7.33: Splitting a heterogeneous-direction edge split may create an irreducible region.

of block b'' necessitates an update the dominator tree as well as the set of loop bodies. The immediate dominator of b'' can be calculated by the dominators of b'''s predecessors, as shown by Alstrup and Laurisden [9]. In addition, the new immediate dominator of b' is b'' when all forward edges terminating at b' are split. Generally, b'' belongs to the same loop as b' except in the case where b' is the header of a loop and backedges are not split. In this case, b'' cannot belong to the same loop as b', as there is no path from b' back to b'' within the loop. In this case, b'' is placed in the loop common to the loops of the blocks in $dom(E)$ and block b'. The loop exits are incrementally maintained by adding b'' to the exit set of all loops for which b' is an exit and removing b' from the exit set of loops for which b'' replaces b' as an exit. The loop nesting structure does not change – no loops are created, destroyed, or unnested from their parent.

7.4 Repairing Violations

In this section, we present a set of transformations to repair canonical form property violations. Each violation can be repaired independently and in any order, so a control flow graph with n distinct violations can be converted into canonical form by n applications of the following transformations. As each transformation adds an additional block to the graph, the canonicalization of a function f with n distinct violations increases the number of blocks by n, which is bounded by above $\sum_{l \in F_f} (2 + |exit_f(l)|)$.

7.4.1 Property 4.3.1 – Unique latch

A loop l with multiple latches can be modified to have a unique latch by re-routing the target of all the former backedges into a fresh block from which a unique backedge originates, as illustrated in Figure 7.41.

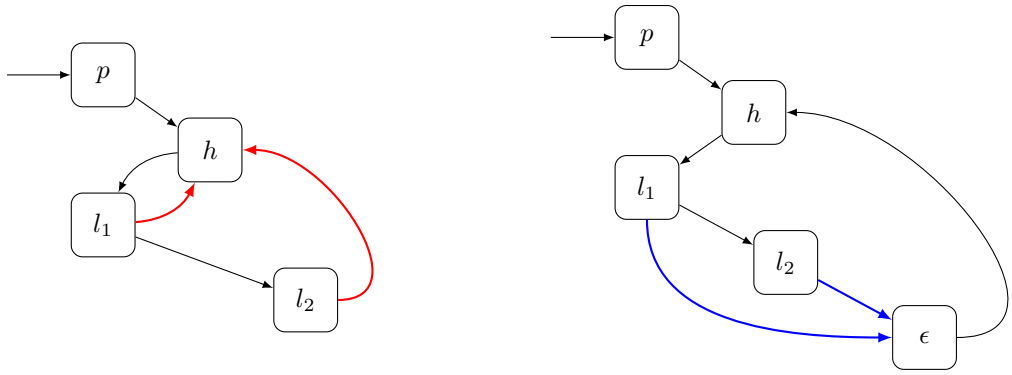


Figure 7.41: Collapsing backedges (l_1, h) and (l_2, h) into the single backedge (ϵ, h).

$$\frac{|latch_f(l)| > 1 \qquad (f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[\{(b, header_f(l)) | b \in latch_f(l)\}]{\text{split edge set}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})}{(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l \in F_f]{\text{unique latch}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})}$$

Figure 7.42: Repairing non-unique latch of loop l.

Figure 7.42 formally defines this transformation parameterized over the loop l that violates Property 4.3.1. All of the heavy lifting of this transformation is off-loaded to the split edge set transformation over the backedges.

7.4.2 Property 4.3.2 – Dedicated preheader

A loop l without a dedicated preheader can be given one by introducing a fresh block whose sole successor is the header of l and re-routing all external edges to l to the preheader, as illustrated in Figure 7.43.

Figure 7.44 formally defines two transformations parameterized over the loop l that violates Property 4.3.2. The first transformation considers graphs where there is *some* external edge terminating at l, but the source of the edge is not a dedicated preheader. This can occur if there are multiple edges entering l, or if the edge entering l originates at a block

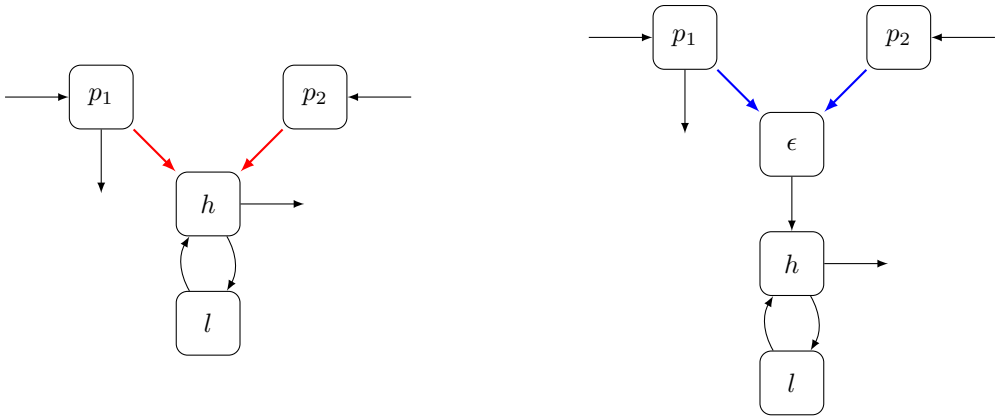


Figure 7.43: Adding a dedicated preheader for loop with the header h.

not dedicated to l. In this case, we can again offload the heavy lifting to the split edge set transformation over the edges entering l. The second transformation considers graphs where *no* external edge enters l. This occurs when the header of l is the entry block of the function. In this case, a new function entry block that immediately jumps to the loop header can be created.

7.4.3 Property 4.3.3 – Dedicated exits

A loop l with an undedicated exit block $e \in exit_f(l)$ can be modified so that this exit is no longer undedicated by placing a block on the loop boundary so that all paths from $b \in l$ to e must travel through this new dedicated exit, as illustrated in Figure 7.45.

Figure 7.46 formally defines this transformation parameterized over the loop l and the undedicated exit $e \in exit_f(l)$ violating Property 4.3.1. Again, this transformation can be expressed as an edge set split over the edges exiting l to e.

$$h = header_f(l) \qquad P = pred_f(h) \setminus body_f(l)$$

$$|P| > 1 \vee succ_f(P) \neq \{h\} \qquad (f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[\{(p,h)|p\in P\}]{\text{split edge set}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

$$\frac{}{(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l\in F_f]{\text{dedicate preheader}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})}$$

$$f = (\ell, t, B, \ell_0) \qquad h = header_f(l)$$

$$lab(h) = \ell_0 \qquad \ell_0' \text{ is fresh} \qquad r_i' \text{ is fresh} \qquad param = \langle (r_i' \colon t_i) \mid (r_i \colon t_i) \in param(h) \rangle$$

$$b' = (\ell_0', param, \emptyset, \langle \texttt{branch}\ \ell_0(\overline{r_i'}) \rangle) \qquad B' = B \cup \{b'\} \qquad f_{out} = (\ell, t, B', \ell_0')$$

$$D_{out} = D_f[b \mapsto p] \qquad H_{out} = H_{F_f} \qquad L_{out} = L_{F_f} \qquad X_{out} = X_{F_f}$$

$$\frac{}{(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l\in F_f]{\text{dedicate preheader}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})}$$

Figure 7.44: Repairing non-dedicated preheader of loop l.

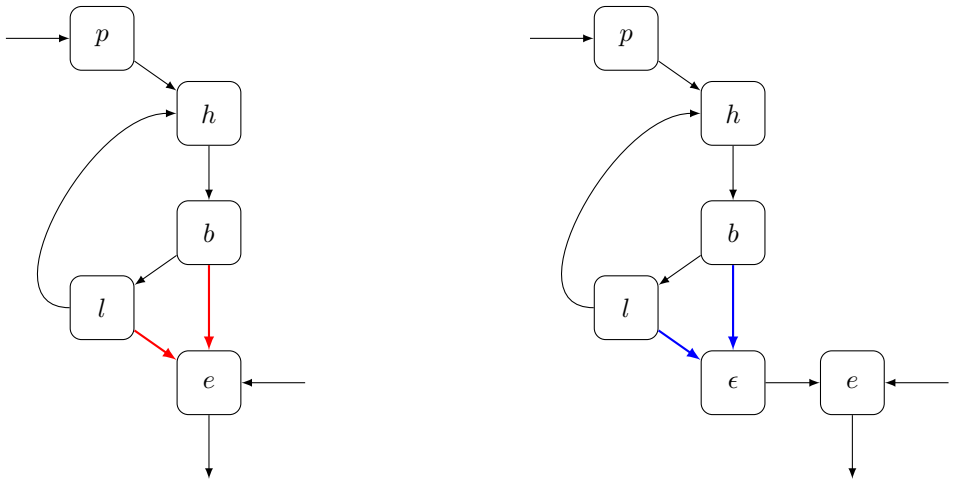


Figure 7.45: Adding a dedicated exit block to replace non-dedicated exit e of loop with the header h.

$$\frac{B = pred_f(e) \cap body_f(l) \qquad B \neq pred_f(e) \qquad (f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[\{(b,e)|b \in B\}]{\text{split edge set}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})}{(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l \in F_f,\ e \in exit_f(l)]{\text{dedicate exit}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})}$$

Figure 7.46: Repairing non-dedicated exit of loop l.

Appendix

Here we provide the proofs of maintenance properties of canonicalization transformations presented in Chapter 7. We specifically cover SSA and LCSSA reconstruction, edge set splitting, unique latch, dedicated preheader, and dedicated exit transformations. The last three transformations rely heavily on edge set splitting.

7.A SSA Reconstruction

In this section, we refer to the *rewrite uses* operation (notated below). For specific details (including names of intermediate components), refer to Figure 7.14.

$$f \xrightarrow[(r\colon t)]{\text{rewrite uses}} (f_{out}, D_{out})$$

Theorem 7.A.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. We prove by filling in the slots of Template 6.A.1.

[**Slot → Refinement**] We define an alternate version of the mapping σ_D as follows.

$$\sigma_{D_{f_t}} = \{(d_i, d_i') \mid d_i \in def_f(r) \text{ where } d_i' = d_i[{}^{r[\overrightarrow{\sigma_{f_t}}]}/_{(\sigma_R \cdot \overrightarrow{\sigma_{f_t}})(d_i)}]\}$$

The difference is subtle but critical: the registers on both sides of the replacement are with respect to the function f_t rather than the original function f. We similarly define an alternate version of the mapping R_k as $R_{f_t}(b) = R_k(b')[\overrightarrow{\sigma_{f_t}}]$ such that $b \in \mathcal{C}(b')$ and $b' \in \mathit{body}(f)$.

We refine the translation of blocks by adding parameters to blocks according to the mapping B_{ϕ_k} and mapping existing block parameter definitions of register r via the mapping $\sigma_{D_{f_t}}$. We omit the translation of implicit parameters as they do not directly affect evaluation semantics (and can be trivially re-calculated via the block's stream). In the following, we let $b_t = (\ell_{b_t}, \langle\overline{(r_i\colon t_i)}\rangle, \{\overline{(r_j\colon t_j)}\}, s_t)$ where s_t is fully qualified as $\langle f_t, b_t, s_t\rangle$.

$$\mathit{trans}(\langle f_t, b_t\rangle) = (\ell_{b_t}, \mathit{addparam}(\langle f_t, b_t\rangle), \_, \mathit{trans}(s_t))$$

$$\mathit{addparam}(\langle f_t, b_t\rangle) = \begin{cases} \langle\overline{(r_i\colon t_i)}\rangle[\sigma_{D_{f_t}}] \cup \langle r_t[\overrightarrow{\sigma_{f_t}}]\colon t\rangle & (b'_t, r_t) \in B_{\phi_k} \wedge b_t \in \mathcal{C}(b'_t) \\ \langle\overline{(r_i\colon t_i)}\rangle[\sigma_{D_{f_t}}] & \mathit{otherwise} \end{cases}$$

We refine the translation of streams (also used by the translation of blocks above) by translating definitions of register r via the mapping $\sigma_{D_{f_t}}$ and translating uses of register r via the mapping R_k. This also adds the reaching definition to the argument list of any block reference to a clone of a block in B_{ϕ_k}.

$$\mathit{trans}(\langle f_t, b_t, ((s_t, T_t), \hat{s}_t)\rangle) = (s_t[\sigma_{D_{f_t}}], \mathit{addarg}(\langle f_t, b_t, T_t\rangle))[{}^{r[\overrightarrow{\sigma_{f_t}}]}/_{R^u_{f_t}(b_t)} \mid u \in \mathit{use}_{f_t}(r[\overrightarrow{\sigma_{f_t}}])], \mathit{trans}(\hat{s}_t)$$

$$\mathit{addarg}(\langle f_t, b_t, T\rangle) = T[{}^{\ell_{b'}(\overline{e})}/_{\ell_{b'}(\overline{e}, R_{f_t}(b_t))} \mid b' \in \mathcal{C}(\mathit{dom}(B_{\phi_k})) \text{ where } \ell_{b'} = \mathit{lab}(b')]$$

We refine the translation of return contexts by replacing the target register with the reaching definition at the instruction following a call when the target register is a definition of the reconstructed register.

$$\mathit{trans}(\langle f_t, b_t, r_t, s\rangle) = \begin{cases} \langle \mathit{trans}(f_t), \mathit{trans}(\langle f_t, b_t\rangle), R^{s_1}_{f_t}(b_t), \mathit{trans}(\langle f_t, b_t, s\rangle)\rangle & r_t = r[\overrightarrow{\sigma_{f_t}}] \\ \langle \mathit{trans}(f_t), \mathit{trans}(\langle f_t, b_t\rangle), r_t, \mathit{trans}(\langle f_t, b_t, s\rangle)\rangle & \mathit{otherwise} \end{cases}$$

We add the following relationship stating that registers distinct from r and the definitions that replace it should have the same value in both register contexts, and that the register context γ_2 should have the same value as register r in register context γ_1 (at some given program point). In the following, we use $\langle f_t, b_t, c_t \rangle$ to mean the leading component of the block's stream in the *same step of evaluation* of function f as the register contexts occur.

$$\gamma_1 \setminus \mathcal{C}(r) = \gamma_2 \setminus \bigcup_{r' \text{ defined by } D_{out}} \mathcal{C}(r') \qquad \gamma_1(r[\overrightarrow{\sigma_{f_t}}]) = \gamma_2(R_k^{c_t}(b_t))$$

[**Slot → Invokability**] The number of parameters of the entry block of functions f and f_{out} must be the same, otherwise there is an ambiguous reaching definition of register r in the entry block. This implies that the implicit parameter set of the entry block of f is not empty and therefore f is not well-typed. However, if register r is a parameter of block f, then that register may be renamed by the transformation. Then, $param(b_{t_2}) = param(b_{t_1})[\sigma_{D_{f_{t_1}}}]$. Let $r' = r[\overrightarrow{\sigma_{f_{t_1}}}]$. If the entry block of function f defines register r, then register context γ_1 introduces a new value for register r' while register context γ_2 introduces the same value for register $R_{f_{t_1}}(b_{t_1})$. In either case, the same set of register-value pairs distinct from r are assigned to both register contexts. Both of these additions result in register contexts consistent with the strengthened induction.

[**Slot → Asymmetric Evaluation**] By the refined *trans* relation, all temporarily diverging steps of evaluation occur in a block that either contains a definition of r, contains a use of r, or is a join point for multiple reaching definitions of r (and thus has an element in the mapping B_{ϕ_k}).

Block Parameters Here, we consider the case where the $(n+1)$-th step of evaluation of f transfers control to a block whose parameters have been modified by the transformation by application of rule E-Switch. Let $r' = r[\overrightarrow{\sigma_{f_{t_1}}}]$, $dom(param(b_{t_1})) = \langle \overline{r_i} \rangle$, and let $\ell_b(\overline{e_i})$ be

the block reference taken by this step of evaluation.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \rightarrow^n (\langle p, f_{t_1}, b_t\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ term(b_t), \hat{s}_{t_1}) \\ \rightarrow (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_1}), \hat{s}_{t_1})$$

By our strengthened inductive hypothesis, the n'-th step of evaluation of f_{out} stops in function $f_{t_2} = trans(f_{t_1})$ on a terminator that transfers control away from $b'_t = trans(\langle f_{t_1}, b_t\rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. The $(n'+1)$-th step of evaluation of f_{out} will proceed by application of rule E-Switch.

$$(p[^f/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \rightarrow^{n'} \\ (\langle p[^f/_{f_{out}}], f_{t_2}, trans(\langle f_{t_1}, b_t\rangle)\rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ trans(\langle f_{t_1}, b_t, term(b_t)\rangle), trans(\hat{s}_{t_1})) \rightarrow \\ (\langle p[^f/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma'_2 \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_2}), trans(\hat{s}_{t_1}))$$

First, we consider the case where $(b_i, r_i) \in B_{\phi_k}$ such that $b_{t_1} \in \mathcal{C}(b_i)$. An extra argument is added to all references targeting block b_{t_1} via application of the function *addarg* in the translation defined above. Let $r'_1 = r_i[\overrightarrow{\sigma_{f_{t_1}}}]$ and $r'_2 = R_{f_{t_1}}(b_t)$. Then, $\ell_b(\overline{e_i}[r'/_{R_{f_{t_1}}(b_{t_1})}], r'_2)$ is the block reference taken by the parallel step of evaluation and $b_{t_2} = trans(\langle f_{t_1}, b_{t_1}\rangle)$. Here, $\gamma'_2 = \gamma_2[\overline{r_i \mapsto \gamma_2(e_i[r'/_{R_{f_{t_1}}(b_{t_1})}])}][r'_1 \leftarrow \gamma_2(r'_2)]$. Then, both register contexts get the same register-value pairs that are distinct from r and evaluation of f_{out} additionally assigns the value of register r'_2, the value of the reaching definition at the bottom of block b_t, to register r'_1, the reaching definition at the top of block b_{t_1}. Thus, the resulting register contexts are consistent with the strengthened induction.

Next, we consider the case where the parameter set of b_{t_1} contains a definition d of register r. Notice that these cases are mutually exclusive, as no component of block b_{t_1} can occur *before* the set of block parameters and necessitate a reaching definition from the block predecessors. Let r'_1 be the register defined by $\sigma_{D_{f_{t_1}}}(d)$. As no extra argument is added to

the reference, $\ell_b(\overline{e_i}[r'/_{R_{f_{t_1}}(b_{t_1})}])$ is the block reference taken by the parallel step of evaluation and $b_{t_2} = \mathit{trans}(\langle f_{t_1}, b_{t_1}\rangle)$. Here, $\gamma_2' = \gamma_2[\overline{r_i[r'/_{r_1'}] \mapsto \gamma_2(e_i[r'/_{R_{f_{t_1}}(b_{t_1})}])}]$. Then, both register contexts get the same register-value pairs that are distinct from r. Additionally, evaluation of f assigns a value for register r' and evaluation of f_{out} assign the same value for register r_1', both the reaching definitions at the top of blocks b_{t_1} and b_{t_2}, respectively.

Instruction Here, we consider the case where the n-th step of evaluation of f evaluates an instruction which has been modified by the transformation. Let the instruction be denoted by $\langle f_{t_1}, b_{t_1}, I\rangle = r \leftarrow O$. By our strengthened inductive hypothesis, the n'-th step of evaluation of f' stops at a similar instruction. Let $r' = r[\overrightarrow{\sigma_{f_{t_1}}}]$ and $r_1' = R^I_{f_{t_1}}(b_{t_1})$. Then, by translation, the symmetric instruction in the parallel evaluation has the form $r[r'/_{\sigma_R \cdot \overrightarrow{\sigma_{f_{t_1}}}}] \leftarrow O[r'/r_1']$. Let γ_1 and γ_2 be the register contexts in the n-th step of evaluation of f and the n'-th step of evaluation of f_{out}, respectively. At this point in evaluation, $\gamma_1(r') = \gamma_2(r_1')$ by assumption. Then, the effective value of operation O is equivalent in both evaluations. Evaluation proceeds in both streams by rule E-Inst. The full case analysis omitted here, but is nearly identical to Lemma 6.B.1 and Lemma 6.B.2.

First, we consider the case where the instruction is not a function call. The resulting streams are the input streams with their head elements removed, in which case the tails of both streams remain consistent with the strengthened induction by assumption. In the case where an instruction causes an abnormal exit, the resulting streams are both ϵ and are also consistent with the strengthened induction. The resulting register contexts either gain the same register (in the case that I was not a definition of r'), or γ_1 assigns a value for r' and γ_2 assigns the same value to the reaching definition of the following instruction.

Now, we consider the case where the instruction is a function call. The resulting functions are either identical, or another pair of instances of functions f and f_{out}. In the former

case, the resulting blocks are identical and their unmodified streams are consistent with the strengthened induction. In the latter case, the resulting functions, blocks, and streams are implied by [Slot *Invokability*]. The resulting return contexts also contain a new element referencing either the same register (in the case that I was not a definition of r'), or referencing register r' in the evaluation of f and referencing the reaching definition of the following instruction in the evaluation of f_{out}.

Terminator Here, we consider the case where the $(n+1)$-th step of evaluation of f evaluates a terminator which has been modified by the transformation. Let the terminator be denoted by $\langle f_{t_1}, b_{t_1}, T\rangle$. By our strengthened inductive hypothesis, the n'-th step of evaluation of f_{out} stops at a similar instruction. Let $r' = r[\overrightarrow{\sigma_{f_{t_1}}}]$, let $r'_1 = R^T_{f_{t_1}}(b_{t_1})$, and let γ_1 and γ_2 be the register contexts in the n-th step of evaluation of f and the n'-th step of evaluation of f_{out}, respectively. At this point in evaluation, $\gamma_1(r') = \gamma_2(r'_1)$ by assumption.

First, assume that T is a `switch`. Let $ref = \ell_b(\overline{e_i})$ be the reference taken by the $(n+1)$-th step of evaluation of f. By translation, the symmetric block reference in the terminator of the parallel evaluation is $\ell_b(\overline{e_i}[r'/r'_i])$. Control is transferred to blocks with the same label and the same set of parameters by rule E-Inst (remember that the case where parameters are modified is handled above). No parameter of either block is a definition of r' and both register contexts gain the same set of register-value pairs.

Now, assume that T is a `return` of the form $T =$ `return` e. By translation, the symmetric return of the parallel evaluation is `return` $e[r'/r'_1]$. Then, the values of expressions e and $e[r'/r'_1]$ are the same. If the return context in the evaluation of f is empty, then the resulting streams are both ϵ by application of rule E-Return-Last and are trivially consistent with the strengthened induction. Otherwise, let $\hat{s}_{t_1} = \langle f_{t_1}, b_{t_1}, r_t, s\rangle$ be the return context in the evaluation of f. The functions, blocks, and streams resulting from application of

rule E-Return are consistent with the strengthened induction by translation of $trans(\hat{s}_{t_1})$. The resulting register contexts either assign the same value for the same register (in the case that $r_t \neq r[\overrightarrow{\sigma_{f_{t_1}}}]$), or γ_1 assigns a value for r' and γ_2 assigns the same value to the reaching definition of the following instruction.

$\cdots$ We omit the remaining cases where evaluation is on an unmodified component within a modified block. These cases can be easily proven by application of the lemmas described in Section 6.B.

[Slot ← Refinement] In the following, we define a substitution that replaces registers symmetric to all registers defined by D_{out} with the register symmetric to register r in function f_t. This substitution is defined as $\sigma^{-1}_{R_{f_t}} = \{(r'[\overrightarrow{\sigma_{f_t}}], r[\overrightarrow{\sigma_{f_t}}]) \mid d_i \in D_{out} \wedge d_i \text{ defines } r'\}$.

We refine the translation of blocks by removing parameters from blocks according to the mapping B_{ϕ_k} and mapping the remaining block parameter definitions of register r via the mapping $\sigma^{-1}s_{R_{f_t}}$. In the following, we let $b_t = (\ell_{b_t}, \langle\overline{(r_i\colon t_i)}\rangle, \{\overline{(r_j\colon t_j)}\}, s_t)$ where s_t is fully qualified as $\langle f_t, b_t, s_t\rangle$.

$$trans^{-1}(\langle f_t, b_t\rangle) = (\ell_{b_t}, remparam(\langle f_t, b_t\rangle)[\sigma^{-1}_{R_{f_t}}], \_, trans(s_t))$$

$$remparam(\langle f_t, b_t\rangle) = \begin{cases} \langle p_1, \ldots, p_{k-1}\rangle & lab(b_t) \in dom(B_{\phi_k}) \wedge param(b_t) = \langle p_1, \ldots, p_k\rangle \\ param(b_t) & otherwise \end{cases}$$

We refine the translation of streams by translating the registers symmetric to the fresh registers introduced to function f_{out} back to the register symmetric to register r in function f_t. This also removes the additional block arguments added by the transformation.

$$trans^{-1}(\langle f_t, b_t, ((s_t, T_t), \hat{s}_t)\rangle) = (s_t, remarg(\langle f_t, b_t, T_t\rangle))[\sigma^{-1}_{R_{f_t}}], trans(\hat{s}_t)$$

$$remarg(\langle f_t, b_t, T\rangle) = T[{}^{\ell_{b'}(\overline{e},\_)}/_{\ell_{b'}(\overline{e})} \mid b' \in \mathcal{C}(dom(B_{\phi_k})) \text{ where } \ell_{b'} = lab(b')]$$

We refine the translation of return contexts by replacing the target register with a register symmetric to register r when the target register is symmetric to a fresh register introduced to function f_{out}.

$$trans^{-1}(\langle f_t, b_t, r_t, s\rangle) = \langle trans(f_t), trans(\langle f_t, b_t\rangle), r_t[\sigma^{-1}_{R_{f_t}}], trans(\langle f_t, b_t, s\rangle)\rangle$$

We use the same relationship for register contexts as the forward case (with the same abuses of notation).

[**Slot ← Invokability**] As in the forward case, the number of parameters of the entry block of functions f and f_{out} must be the same. Then, $param(b_{t_1}) = param(b_{t_2})[\sigma^{-1}_{R_{f_{t_2}}}]$. The remainder of this case proceeds as in the forward case.

[**Slot ← Asymmetric Evaluation**] By the refined $trans^{-1}$ relation, all temporarily diverging steps of evaluation occur in a block that contains either a definition or a use of a fresh register introduced by the transformation.

Block Parameters Here, we consider the case where the $(n'+1)$-th step of evaluation of f_{out} transfers control to a block whose parameters have been modified by the transformation by application of rule E-Switch. Let $dom(param(b_{t_2})) = \langle \overline{r_i} \rangle$ and let $\ell_b(\overline{e_i})$ be the block reference taken by this step of evaluation.

$$(p[^f/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \rightarrow^n$$

$$(\langle p[^f/_{f_{out}}], f_{t_2}, b_t\rangle \mid \gamma_2 \mid \mu' \mid \nu' \mid \Psi';\ term(b_t), \hat{s}_{t_2}) \rightarrow$$

$$(\langle p[^f/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_2}), \hat{s}_{t_2})$$

By our strengthened inductive hypothesis, the n-th step of evaluation of f stops in function $f_{t_1} = trans^{-1}(f_{t_2})$ on a terminator that transfers control away from $b'_t = trans(\langle f_{t_2}, b_t\rangle)$,

as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. The $(n+1)$-th step of evaluation of f will proceed by application of rule E-Switch.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^{n'}$$

$$(\langle p, f_{t_1}, \mathit{trans}^{-1}(\langle f_{t_2}, b_t\rangle)\rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ \mathit{trans}^{-1}(\langle f_{t_2}, b_t, \mathit{term}(b_t)\rangle), \mathit{trans}^{-1}(\hat{s}_{t_2})) \to$$

$$(\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_2' \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_1}), \mathit{trans}^{-1}(\hat{s}_{t_2}))$$

In the following, let $r_1' = R_{f_{t_1}}(b_{t_1})$. First, we consider the case where $\mathit{lab}(b_{t_2}) \in \mathit{dom}(B_{\phi_k})$. The last occurring argument is removed from all references targeting block b_{t_2} via application of the function *remarg* in the translation defined above. Let $k = |\langle \overline{r_i}\rangle|$. Then, $\ell_b(\langle e_{i_1}, \ldots, e_{i_{k-1}}\rangle[r_1'/r[\overrightarrow{\sigma_{f_{t_1}}}]])$ is the block reference taken by the parallel step of evaluation and $b_{t_1} = \mathit{trans}(\langle f_{t_2}, b_{t_2}\rangle)$. Here, $\gamma_1' = \gamma_1[\overline{r_i \mapsto \gamma_1(e_i[r_1'/r[\overrightarrow{\sigma_{f_{t_1}}}]])}^{i<k}]$ and the resulting register contexts remain consistent with the strengthened induction with reasoning symmetric to the forward case.

Next, we consider the case the block parameters of b_{t_2} contains a definition in $d \in D_{out}$ that replaced an existing block parameter. Let $e_i' = e_i[r_1'/r[\overrightarrow{\sigma_{f_{t_1}}}]]$. In the following, let $\ell_b(\overline{e_i'})$ is the block reference taken by the parallel step of evaluation and $b_{t_1} = \mathit{trans}^{-1}(\langle f_{t_2}, b_{t_2}\rangle)$. Here, $\gamma_1' = \gamma_1[\overline{r_i \mapsto \gamma_1(e_i')}]$ and the resulting register contexts remain consistent with the strengthened induction with reasoning symmetric to the forward case.

$\cdots$ The remaining cases in this direction are omitted as the only significant difference is the direction in which the registers are mapped (in the forward case it is from register r to a fresh definition; in the reverse case it is all definitions of D_{out} to register r). □

Theorem 7.A.2. If $p \mid f$ is well-typed, then $p \mid f_{out}$ is well-typed.

Proof. Let b be a block that has been changed by this transformation. Then, at least one of the following must be true: b is given a new block parameter, a definition in $d \in \mathit{def}_f(r)$

occurs in b, a successor of b is given a new block parameter, or a use in $u \in use_f(r)$ occurs in b. We show that for any combination of these changes, block b remains well-typed. In the following, note that every register in R_k and B_{ϕ_k} have type t.

First, suppose b is given a new block parameter defining register r'. If b is not further altered such that r' is later referenced by an instruction or the block's terminator, then the additional parameter simply expands the typing environment for the block's stream and b remains well-typed by rule T-Block. We discuss the cases where b is further altered to contain a reference to r' below.

Second, suppose that block b contains definition d. Then, register r is replaced by register $r' = \sigma_R(d)$ in d. The definition remains trivially well-typed by rule T-Inst.

Third, suppose that a successor s of block b is given a new block parameter (also of type t). Then, the register $r' = R_k(b)$ is added as a final argument to each block reference targeting block s in the terminator of block b. Register r' is either defined within block b (by a definition previously defining r or by a newly inserted block parameter), or r' is an implicit parameter of block b by application of the *replace register* operation. In either case, $(r' : t)$ at the time the component is typed. As the addition of the argument and the parameter have the same type, each block reference is well-typed by rule T-Ref.

Fourth and finally, suppose block b contains the use u. Then, register r is replaced by register $r' = R_k^u(b)$ in u. As in the previous case (and for the same reasons), $(r' : t)$ at the time the component is typed.

Lastly, we need to show that the nonlocals are well-formed. By construction of each R_i, every pair $(b \mapsto r') \in R_i$ it must be the case that the definition of r' dominates block b in f_{out}. If there were multiple reaching definitions at block b, they would be disambiguated by a block parameter inserted in block b (thus the definition trivially dominates the block). If there was a single reaching definition at block b, then either b defines r'(and the definition again

trivially dominates the block), or all the predecessors of b have the same reaching definition (and the same argument applies recursively). Also notice that the *find def* procedure 'gets stuck' when the input is a function that contains a use before a definition (or a use that is not dominated by a set of definitions) – as $p \mid f$ is well-typed, this is not the case. The nonlocal parameters of blocks between block b and the definition of r' gain an additional register, but this register is removed from the nonlocal set of the block where r' is defined. As definitions of all such registers dominate each of its uses, $nonlocal_{f_{out}}(entry(f_{out})) = \emptyset$ and f_{out} is well-typed by rule T-Func. □

Corollary 7.A.3. The set of paths over block labels from ℓ_0 to $\ell_{b'}$ in $G_{f_{out}}$ for some block $b' \in B$ where $lab(b') = \ell_{b'}$ is identical to the set of such paths in G_f.

Theorem 7.A.4. If f is in canonical form, then f_{out} is in canonical form.

Proof. Trivial by Corollary 7.A.3. □

Theorem 7.A.5. If r is a register defined multiple times in f_{out}, then r is also defined multiple times in f.

Proof. All definitions introduced to f_{out} come from the mapping σ_R and the set B_{ϕ_k}, both of which are constructed with fresh registers. □

Corollary 7.A.6. All definitions of register r in f are rewritten to define fresh registers in f_{out}.

Theorem 7.A.7. If there are no uses of register r violating LCSSA form in function f, then every use u' violating LCSSA form in function f_{out} has a symmetric use u violating LCSSA form in function f.

Proof. Let r' be the register used by u'. If function f also defines register r', then the set of definitions and uses of register r' are symmetric in both functions. Otherwise, the

definition of r' was introduced to function f_{out} to either replace a definition of register r or to disambiguate two unique reaching definitions of register r. In the latter case, each such point occurs on a path from the uses to the set of dominating definitions (in the reverse graph). As it was assumed that all uses of register r occur in the loop as its definitions, no such path can exit the boundary of said loop. □

Theorem 7.A.8. The unique dominator tree of $G_{f_{out}}$ is D_f.

Proof. Trivial by Corollary 7.A.3. □

Theorem 7.A.9. If f is in canonical form, then F_f is the unique loop nesting forest of $G_{f_{out}}$.

Proof. Trivial by Corollary 7.A.3. □

7.B LCSSA Reconstruction

In this section, we refer to the *repair lcssa* operation (notated below). For specific details (including names of intermediate components), refer to Figure 7.22.

$$f \xrightarrow[\langle b,d \rangle]{\text{repair lcssa}} (f_{out}, B_\phi, \sigma_\phi)$$

Lemma 7.B.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f' can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. We prove by filling in the slots of Template 6.A.1.

[**Slot → Refinement**] We refine the translation of blocks by adding parameters to clones of $exits_{old}$. We omit the translation of implicit parameters as they do not directly affect

evaluation semantics (and can be trivially re-calculated via the block's stream). In the following, we let $b_t = (\ell_{b_t}, \langle\overline{(r_i : t_i)}\rangle, \{\overline{(r_j : t_j)}\}, s_t)$ where s_t is fully qualified as $\langle f_t, b_t, s_t\rangle$.

$$trans(\langle f_t, b_t\rangle) = (\ell_{b_t}, addparam(\langle f_t, b_t\rangle), \_, trans(s_t))$$

$$addparam(\langle f_t, b_t\rangle) = \begin{cases} \langle\overline{(r_i : t_i)}, r[\overrightarrow{\sigma_{f_t}}] : t\rangle & b_t \in \mathcal{C}(exits_{old}) \\ \langle\overline{(r_i : t_i)}\rangle & otherwise \end{cases}$$

We refine the translation of streams (also used by the translation of blocks above) by adding the register r to the argument list of any block reference to a clone of block in $exits_{old}$. By construction, this set of blocks contains all references to blocks in $exits_{old}$, so we can safely apply this mapping over terminators regardless of the source block when in a clone of function f.

$$trans(\langle f_t, b_t, ((s_t, T_t), \hat{s}_t)\rangle) = \begin{cases} (s_t, addarg(\langle f_t, b_t, T_t\rangle)), trans(\hat{s}_t) & f_t \in \mathcal{C}(f) \\ (s_t, T_t), trans(\hat{s}_t) & otherwise \end{cases}$$

$$addarg(\langle f_t, b_t, T\rangle) = T[{}^{\ell_{b'}(\overline{e})}/_{\ell_{b'}(\overline{e}, r[\overrightarrow{\sigma_{f_t}}])} \mid b' \in \mathcal{C}(exits_{old}) \text{ where } \ell_{b'} = lab(b')]$$

We add the following additional relationship between register contexts.

$$\gamma_1 = \gamma_2$$

[Slot → Invokability] It cannot be the case that $b_{t_1} \in \mathcal{C}(exits_{old})$ as each block in $exits_{old}$ is strictly dominated by $entry(f)$. First, suppose $b_t \in preds_{old}$. In this case, canonical form must be broken as the entry block cannot otherwise be a member of a loop. As b_t is a direct predecessor of an exit block through which there is a path to the use of r and f is well-typed, r must be defined in the entry block. In the following, we decompose $stream(b_t)$ into (s_t, T). Let $s'_t = (s_t, addarg(\langle f_t, b_t, T\rangle))$. Then, $trans(\langle f_{t_1}, b_t\rangle) = (lab(b_t), param(b_t), \emptyset, s'_t)$, $trans(\langle f_{t_1}, b_t, stream(b_t)\rangle) = s'_t$, and the identical register contexts are consistent with the

strengthened induction. In all other cases, the identical blocks, streams, and register contexts are consistent with the strengthened induction.

[Slot → Asymmetric Evaluation] By the refined *trans* relation, all temporarily diverging steps of evaluation occur at the terminator of a clone of block in $exits_{old}$.

Altered Exit In this case, we consider the case where the $(n+1)$-th step of evaluation of f transfers control from a block in $b_t \in \mathcal{C}(preds_{old})$ by application of rule E-Switch. First, we assume that control is transferred to a block $\mathcal{C}(exits_{old})$. Here, $dom(param(b_{t_1})) = \langle \overline{r_i} \rangle$. Let $T = \texttt{switch}\ v\ \overline{cv_i \mapsto ref_i}\ ref_d$ and let $ref = \ell_{b'}(\overline{e_i})$ be the block reference taken by this step of evaluation.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n\ (\langle p, f_{t_1}, b_t \rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ T, \hat{s}_{t_1})$$
$$\to\ (\langle p, f_{t_1}, b_{t_1} \rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_1}), \hat{s}_{t_1})$$

Let $r_t = r[\overrightarrow{\sigma_{f_{t_1}}}]$ in the following. By our strengthened inductive hypothesis, the n'-th step of evaluation of f' stops in function $f_{t_2} = trans(f_{t_1})$ on a terminator that transfers control away from $trans(\langle f_{t_1}, b_t \rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. Then, the $(n'+1)$-th step of evaluation of f' proceeds from application of rule E-Switch by taking the symmetric switch case. Here, $b_{t_2} = trans(\langle f_{t_1}, b_{t_1} \rangle)$ and as $b_t \in exits_{old}$, $param(b_{t_2}) = param(b_{t_1}) \cup \langle (r_t : t) \rangle$. Let $\gamma_2' = \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}]$ so that it can be referred to by name in the following.

$$(p[f/f'] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f'(\overline{v_{t_i}})) \to^{n'}$$
$$(\langle p[f/f'], f_{t_2}, trans(\langle f_{t_1}, b_t \rangle) \rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ addarg(\langle f_{t_1}, b_{t_1}, T \rangle), trans(\hat{s}_{t_1})) \to$$
$$(\langle p[f/f'], f_{t_2}, b_{t_2} \rangle \mid \gamma_2'[r_t \mapsto \gamma_2'(r_t)] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_2}), trans(\hat{s}_{t_1}))$$

The additional assignment to γ_2 has no effect on the content of the context and can be effectively ignored. Then, as register contexts γ_1 and γ_2 are consistent with the strengthened

induction, they remain consistent after the parallel addition of registers in the last step of evaluation.

In the other case, we assume that control is transferred to another block distinct from the set $\mathcal{C}(\mathit{exits}_{old})$. Here, $\mathit{dom}(\mathit{param}(b_{t_1})) = \langle \overline{r_i} \rangle$ and the evaluation of function f proceeds as in the case above. By our strengthened inductive hypothesis, the n'-th step of evaluation of f' stops in function $f_{t_2} = \mathit{trans}(f_{t_1})$ on a terminator that transfers control away from $\mathit{trans}(\langle f_{t_1}, b_t \rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. Then, the $(n'+1)$-th step of evaluation of f' proceeds from application of rule E-Switch by taking the symmetric switch case. Here, $b_{t_2} = \mathit{trans}(\langle f_{t_1}, b_{t_1} \rangle)$ and as $b_t \notin \mathit{exits}_{old}$, $\mathit{param}(b_{t_2}) = \mathit{param}(b_{t_1})$.

$$(p[f/f'] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f'(\overline{v_{t_i}})) \to^{n'}$$

$$(\langle p[f/f'], f_{t_2}, \mathit{trans}(\langle f_{t_1}, b_t \rangle) \rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ \mathit{addarg}(\langle f_{t_1}, b_{t_1}, T \rangle), \mathit{trans}(\hat{s}_{t_1})) \to$$

$$(\langle p[f/f'], f_{t_2}, b_{t_2} \rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_2}), \mathit{trans}(\hat{s}_{t_1}))$$

As register contexts γ_1 and γ_2 are consistent with the strengthened induction, they remain consistent after the parallel addition of registers in the last step of evaluation.

Altered Block Next, we cover the cases where the n-th step of evaluation of f is earlier within a block $b \in \mathcal{C}(\mathit{preds}_{old})$ (but not yet at the terminator). This case is trivial by application of Lemma 6.C.1.

[Slot ← Refinement] We refine the translation of blocks and streams as the reverse of the forward direction. To translate blocks, we remove the additional parameters and implicits added by the transformation. Removing a parameter simply chops off the last item in the parameter lists. We omit the translation of implicit parameters as they do not directly affect

evaluation semantics (and can be trivially re-calculated via the block's stream).

$$trans(\langle f_t, b_t \rangle) = (\ell_{b_t}, remparam(\langle f_t, b_t \rangle), \_, trans(s_t))$$

$$remparam(\langle f_t, b_t \rangle) = \begin{cases} \langle p_1, \ldots, p_{k-1} \rangle & b_t \in \mathcal{C}(exits_{new}) \wedge param(b_t) = \langle p_1, \ldots, p_k \rangle \\ param(b_t) & otherwise \end{cases}$$

Similarly, to translate streams, we remove the additional arguments added by the transformation. Removing an argument simply chops off the last item in the argument lists.

$$trans(\langle f_t, b_t, ((s_t, T_t), \hat{s}_t) \rangle) = \begin{cases} (s_t, remarg(\langle f_t, b_t, T_t \rangle)), trans(\hat{s}_t) & b_t \in \mathcal{C}(preds_{new}) \\ (s_t, T_t), trans(\hat{s}_t) & otherwise \end{cases}$$

$$remarg(\langle f_t, b_t, T \rangle) = T[{}^{\ell_{b'}(\overline{e}, \_)}/_{\ell_{b'}(\overline{e})} \mid b' \in \mathcal{C}(exits_{new}) \text{ where } \ell_{b'} = lab(b')]$$

We add the following additional relationship between register contexts.

$$\gamma_1 = \gamma_2$$

[**Slot ← Invokability**] First, suppose $b_t \in preds_{new}$. Let $r' = r[\overrightarrow{\sigma_{f_{t_2}}}]$ be the register symmetric to r in this instance of function f'. This register must be defined in block b_t as r is defined in the entry block of f and no definitions are removed in the construction of function f'. We decompose $stream(b_t)$ into (s_t, T). Let $s'_t = (s_t, remarg(\langle f_{t_2}, b_t, T \rangle))$. Then, $trans^{-1}(\langle f_{t_2}, b_t \rangle) = (lab(b_t), param(b_t), \emptyset, s'_t)$, $trans^{-1}(\langle f, b_t, stream(b_t) \rangle) = s'_t$, and the identical register contexts are consistent with the strengthened induction. In all other cases, $trans^{-1}(\langle f_{t_2}, b_t \rangle) = b_t$, $trans^{-1}(\langle f_{t_2}, b_t, stream(b_t) \rangle) = stream(b_t)$, and the identical register contexts are consistent with the strengthened induction.

[**Slot ← Asymmetric Evaluation**] By the refined $trans^{-1}$ relation, all temporarily diverging steps of evaluation occur at the terminator of a clone of a block in $exits_{new}$.

Altered Exit In this case, we consider the case where the $(n'+1)$-th step of evaluation of f' transfers control from a block in $b_t \in \mathcal{C}(preds_{new})$ by application of rule E-Switch. First, we assume that control is transferred to a block $\mathcal{C}(exits_{new})$. Here, $dom(param(b_{t_2})) = \langle \overline{r_i} \rangle$. Let $T = \texttt{switch}\ v\ \overline{cv_i \mapsto ref_i}\ ref_d$ and let $ref = \ell_{b'}(\overline{e_i})$ be the block reference taken by this step of evaluation.

$$(p[f/f'] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f'(\overline{v_{t_i}})) \rightarrow^{n'}$$

$$(\langle p[f/f'], f_{t_2}, b_t \rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ T, \hat{s}_{t_2}) \rightarrow$$

$$(\langle p[f/f'], f_{t_2}, b_{t_2} \rangle \mid \gamma_2[r_t \mapsto \gamma_2(r_t)] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_2}), \hat{s}_{t_2}))$$

By our strengthened inductive hypothesis, the n-th step of evaluation of f stops in function $f_{t_1} = trans^{-1}(f_{t_2})$ on a terminator that transfers control away from $trans^{-1}(\langle f_{t_2}, b_t \rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. Then, the $(n+1)$-th step of evaluation of f proceeds from application of rule E-Switch by taking the symmetric switch case. Here, $b_{t_1} = trans^{-1}(\langle f_{t_2}, b_{t_2} \rangle)$ and as $b_t \in exits_{new}$, $param(b_{t_1})$ consists of all but the last element of $param(b_{t_2})$. Let $k = |param(b_{t_2})|$ in the following.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \rightarrow^{n}$$

$$(\langle p, f_{t_1}, trans^{-1}(\langle f_{t_2}, b_t \rangle) \rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ remarg(\langle f_{t_2}, b_{t_2}, T \rangle), trans^{-1}(\hat{s}_{t_2})) \rightarrow$$

$$(\langle p, f_{t_1}, b_{t_1} \rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}^{(i<k)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_1}), trans^{-1}(\hat{s}_{t_2}))$$

The additional assignment in register context γ_2 simply reassigns a register its old value, and the absence of this assignment in register context γ_1 has no observable effects on its contents. Then, both register contexts remain consistent with the strengthened induction after the parallel addition of registers.

In the other case, we assume that control is transferred to another block distinct from the set $\mathcal{C}(exits_{new})$. Here, $dom(param(b_{t_2})) = \langle \overline{r_i} \rangle$ and the evaluation of function f' proceeds

as in the case above. By our strengthened inductive hypothesis, the n-th step of evaluation of f stops in function $f_{t_1} = \mathit{trans}^{-1}(f_{t_2})$ on a terminator that transfers control away from $\mathit{trans}^{-1}(\langle f_{t_2}, b_t \rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. Then, the $(n+1)$-th step of evaluation of f proceeds from application of rule E-SWITCH by taking the symmetric switch case. Here, $b_{t_1} = \mathit{trans}^{-1}(\langle f_{t_2}, b_{t_2} \rangle)$ and as $b_t \notin \mathit{exits}_{new}$, $\mathit{param}(b_{t_1}) = \mathit{param}(b_{t_2})$.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n$$

$$(\langle p, f_{t_1}, \mathit{trans}^{-1}(\langle f_{t_2}, b_t \rangle) \rangle \mid \gamma_1 \mid \mu' \mid \Psi';\ \mathit{remarg}(\langle f_{t_2}, b_{t_2}, T \rangle), \mathit{trans}^{-1}(\hat{s}_{t_2})) \to$$

$$(\langle p, f_{t_1}, b_{t_1} \rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_1}), \mathit{trans}^{-1}(\hat{s}_{t_2}))$$

As register contexts γ_1 and γ_2 are consistent with the strengthened induction, they remain consistent after the parallel addition of registers in the last step of evaluation.

Altered Block Next, we cover the cases where the n'-th step of evaluation of f' is earlier within a block $b \in \mathcal{C}(\mathit{preds}_{new})$ (but not yet at the terminator). This case is trivial by application of Lemma 6.C.1. □

Theorem 7.B.2. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. By application of Lemma 7.B.1 and Theorem 7.A.1. □

Corollary 7.B.3. The set of paths over block labels from ℓ_0 to $\ell_{b'}$ in $G_{f_{out}}$ for some block $b' \in B$ where $\mathit{lab}(b') = \ell_{b'}$ is identical to the set of such paths in G_f.

Theorem 7.B.4. If f is in canonical form, then f_{out} is in canonical form.

Proof. Trivial by Corollary 7.B.3. □

Lemma 7.B.5. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f'$ is well-typed.

Proof. Consider a block $b_{old} \in \mathit{exits}_{old} \cup \mathit{preds}_{old}$ and the corresponding block in the intermediate function $b_{new} = \mathit{block}_{f'}(\mathit{lab}(b_{old}))$. First, suppose b_{new} is given a new block parameter defining r of type t. As f is in SSA form, the only other definition of register r gives it a value of type t. If register r is used in b_{new}, then this parameter takes the place of an implicit register in r and the typing context used to type the block's stream is the same as b_{old}. If register r is unused in b_{new}, then an additional register exists in the register environment typing that Lemma 3.A.5 shows is not meaningful. Now, suppose that an argument is added to some number of references in b_{new}. Each reference targets a block constructed from exits_{old}, each such block gaining an additional parameter of type t. Each additional argument is the register r of type t, and each reference remains well-typed by rule T-Ref. If register r is not otherwise used in this block, then it is added as an implicit parameter by application of the *add arg* procedure. This also adds register r to the nonlocal parameters of blocks on the path from b_{new} to the definition of r. However, as $p \mid f$ is well-typed and f is in SSA form, the block containing the original definition necessarily dominates b_{new}. Then, $\mathit{nonlocal}(\mathit{entry}(f_{out})) = \emptyset$ and the function is well-typed by rule T-Block. □

Theorem 7.B.6. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f_{out}$ is well-typed.

Proof. Trivial by repeated application of Theorem 7.A.2 and Lemma 7.B.5. □

Theorem 7.B.7. The unique dominator tree of $G_{f_{out}}$ is D_{out}.

Proof. Trivial by Corollary 7.B.3. □

Theorem 7.B.8. If f is in canonical form, then F_{out} reconstructed from $(H_{out}, L_{out}, X_{out})$ is the unique loop nesting forest of $G_{f_{out}}$.

Proof. Trivial by Corollary 7.B.3. □

Theorem 7.B.9. If f_{out} contains more than one definition of register r, then f also contains more than one definition of register r.

Proof. All definitions introduced to f_{out} are subsequently an argument to the *rewrite uses* procedure which modifies all such definitions to assign a fresh register. No such violations are introduced by the *rewrite uses* procedure by Theorem 7.A.5. □

Theorem 7.B.10. Let $\langle f_{out}, b', d' \rangle$ be a definition of register r and let $l = loop_f(b')$. If there exists a use u of register r in f_{out} occurring outside of l, then there exists a symmetric use u' of register r occurring outside of l in f.

Proof. First, suppose register r is also defined in f. Then the definition and uses of this register are identical in f and f_{out} and the violations are the same in both functions. Now, suppose that register r is not defined in f. Then, the definition of r in f_{out} must have been introduced by an application of the *rewrite uses* procedure. If d' is the definition symmetric to definition d, then all uses of r outside of l were rewritten to use a register defined by a set of dominating definitions outside of l (block parameters placed at exits of l). In this case, no such violating use can exist in f_{out}. If d' is not the definition symmetric to definition d, then d' must have been a block parameter introduced by application of the *rewrite uses* procedure. All such definitions are subject to a recursive application of the *repair lcssa* procedure which by the same logic above and thus no such violating use can exist in f_{out}. □

7.C Edge Set Splitting

In this section, we refer to the *split edge set* operation (notated below). For specific details (including names of intermediate components), refer to Figure 7.32.

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[E=\{\overline{(b_i,b')}\}]{\text{split edge set}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Theorem 7.C.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. We prove by filling in the slots of Template 6.A.1.

[**Slot → Refinement**] We refine the translation of blocks and streams by replacing references to clones of block b' with a reference to the relevant clone of block b'' in all clones of $dom(E)$.

$$trans(\langle f_t, b_t \rangle) = \begin{cases} b_t[\ell_{b'}/\ell_{b''}] & b_t \in \mathcal{C}(dom(E)) \\ b_t & otherwise \end{cases}$$

$$trans(\langle f_t, b_t, s_t, \hat{s}_t \rangle) = \begin{cases} s_t[\ell_{b'}/\ell_{b''}], trans(\hat{s}_t) & b_t \in \mathcal{C}(dom(E)) \\ s_t, trans(\hat{s}_t) & otherwise \end{cases}$$

We add the following additional relationship between register contexts, which allows registers of the fresh block in the register context of the parallel evaluation.

$$\gamma_1 \subseteq \gamma_2$$

[Slot → Invokability] If $b_t \in \mathcal{C}(dom(E))$, then $b_{t_2} = b_{t_1}[\ell'_b/\ell''_b]$. Otherwise, $b_{t_2} = b_{t_1}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[Slot → Asymmetric Evaluation] By the refined *trans* relation, all temporarily diverging steps of evaluation occur at the terminator of a clone of a block in E.

Altered Switch We cover the case where the $(n+1)$-th step of evaluation of f transfers control from block $b_t \in \mathcal{C}(dom(E))$. Let $ref = \ell_{b'}(\overline{e_i})$ be the block reference taken by this step of evaluation and let $dom(param(b_{t_1})) = \langle \overline{r_i} \rangle$. For brevity, let $T = term(b_t)$.

$$\begin{aligned}(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \rightarrow^n\ & (\langle p, f_{t_1}, b_t \rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ T, \hat{s}_{t_1}) \\ \rightarrow\ & (\langle p, f_{t_1}, b_{t_1} \rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_1}), \hat{s}_{t_1})\end{aligned}$$

By our strengthened inductive hypothesis, the n'-th step of evaluation of f_{out} stops in function $f_{t_2} = trans(f_{t_1})$ on a terminator that transfers control from block $trans(\langle f_{t_1}, b_t \rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. The $(n'+1)$-th step of evaluation of f_{out} proceeds by applications of rule E-Switch by first taking the symmetric switch case. First, we consider the case where $b_{t_1} \in \mathcal{C}(b')$. Here, $b'_t \in \mathcal{C}(b'')$. Let $dom(param(b'_t)) = \langle \overline{r'_i} \rangle$ and, for brevity, let $\gamma'_2 = \gamma_2[\overline{r'_i \mapsto \gamma_2(e_i)}]$ and $\gamma''_2 = \gamma'_2[\overline{r_i \mapsto \gamma'_2(r'_i)}]$.

$$(p[f/f_{out}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \rightarrow^{n'}$$

$$(\langle p[f/f_{out}], f_{t_2}, trans(\langle f_{t_1}, b_t \rangle) \rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ T[\ell_{b'}/\ell_{b''}], trans(\hat{s}_{t_1})) \rightarrow$$

$$(\langle p[f/f_{out}], f_{t_2}, b'_t \rangle \mid \gamma'_2 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{branch}\ \ell_{b'}(\overline{r'_i}), trans(\hat{s}_{t_1})) \rightarrow$$

$$(\langle p[f/f_{out}], f_{t_2}, b_{t_2} \rangle \mid \gamma''_2 \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_2}), \hat{s}_{t_1} \rangle))$$

Then, $b_{t_2} = trans(\langle f_{t_1}, b_{t_1} \rangle)$ and the set of register-value pairs added to register context γ_2 is a strict superset of the register-value pairs added to register context γ_1.

Next, we consider the case where $b_{t_1} \notin \mathcal{C}(b')$. Then, $b_{t_2} = trans(\langle f_{t_1}, b_{t_2} \rangle)$.

$$(p[f/f_{out}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \rightarrow^{n'}$$

$$(\langle p[f/f_{out}], f_{t_2}, trans(\langle f_{t_1}, b_t \rangle) \rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ T[\ell_{b'}/\ell_{b''}], trans(\hat{s}_{t_1})) \rightarrow$$

$$(\langle p[f/f_{out}], f_{t_2}, b_{t_2} \rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ trans(\langle f_{t_1}, b_{t_1}, stream(b_{t_1}, \hat{s}_{t_1} \rangle)))$$

The resulting register contexts are extended with the same register-value pair, which does not violate the strengthened induction.

Altered Block Next, we cover the cases where the n-th step of evaluation of f is earlier within a block $b \in \mathcal{C}(dom(E))$ (but not yet at the terminator). This case is trivial by application of Lemma 6.C.1.

[**Slot ← Refinement**] We refine the translation of blocks and streams by replacing references to clones of block b'' with a reference to the relevant clone of block b'.

$$trans(\langle f_t, b_t \rangle) = \begin{cases} b'[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(b'') \\ b_i[\ell_{b''}/\ell_{b'}] & b_t \in (b'_i) \\ b_t & otherwise \end{cases}$$

$$trans(\langle f_t, b_t, s_t, \hat{s}_t \rangle) = \begin{cases} stream(b')[\overrightarrow{\sigma_{f_t}}], trans(\hat{s}_t) & b_t \in \mathcal{C}(b'') \\ s_t[\ell_{b''}/\ell_{b'}], trans(\hat{s}_t) & b_t \in \mathcal{C}(b'_i) \\ s_t, trans(\hat{s}_t) & otherwise \end{cases}$$

Now, we define the relationship following stating that registers occurring in γ_2 but not γ_1 must be a parameter to a clone of block b''. We additionally allow the register context γ_2 to temporarily break the superset relation with register context γ_1 when evaluation of f_{out} is within a clone of block b' – it is not until the following step of evaluation that γ_2 will assign

the set of registers defined by the parameters of the successor block. In the following, we use f_t and b_t to mean the function and block in the *same step of evaluation* of function f_{out} as the register contexts occur.

$$\gamma_1 \subseteq \gamma_2 \text{ if } b_t \notin \mathcal{C}(b') \qquad \{(r, cv) \in \gamma_1 \mid r \notin \{dom(param(succ_{f_t}(b_t)))\}\} \subseteq \gamma_2 \text{ if } b_t \in \mathcal{C}(b')$$

$$dom(\gamma_2 \setminus \gamma_1) \subseteq \bigcup_{b_i \in \mathcal{C}(b'')} dom(param(b_i))$$

[**Slot ← Invokability**] As the entry block is not modified, it cannot be the case that $b_{t_2} \in \mathcal{C}(b'')$. If $b_{t_2} \in \mathcal{C}(b'_i)$, then $b_{t_1} = b_i[\overrightarrow{\sigma_{f_{t_2}}}]$. Otherwise, $b_{t_1} = b_{t_2}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[**Slot ← Asymmetric Evaluation**] By the refined *trans* relation, all temporarily diverging steps of evaluation occur at the terminator of a clone of a block in b'_i or within a clone of block b''.

Altered Switch We cover the case where the $(n'+1)$-th step of evaluation of f_{out} transfers control from block $b_t \in \mathcal{C}(\{\overline{b'_i}\})$. First, we consider the case where control is transferred to a block $b_{t_2} \in \mathcal{C}(b'')$. Let $dom(param(b_{t_2})) = \langle \overline{r'_i} \rangle$. For brevity, let $T = term(b_t)$.

$$(p[^f/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \rightarrow^{n'}$$

$$(\langle p[^f/_{f_{out}}], f_{t_2}, b_t \rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ T, \hat{s}_{t_2}) \rightarrow$$

$$(\langle p[^f/_{f_{out}}], f_{t_2}, b_{t_2} \rangle \mid \gamma_2[\overline{r'_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_2}), \hat{s}_{t_2} \rangle)$$

By our strengthened inductive hypothesis, the n-th step of evaluation of f stops in function $f_{t_1} = trans(f_{t_2})$ on a terminator that transfers control from block $trans^{-1}(\langle f_{t_2}, b_t \rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. The $(n+1)$-th step of evaluation of f proceeds by application of rule E-Switch by first taking the symmetric

switch case. Let $dom(param(b_{t_1})) = \langle \overline{r_i} \rangle$. Here, $b_{t_1} \in \mathcal{C}(b')$ and the register contexts, while gaining values for a disjoint set of registers, are consistent with the strengthened induction.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n$$
$$(\langle p, f_{t_1}, trans^{-1}(\langle f_{t_2}, b_t \rangle) \rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ T, trans^{-1}(\hat{s}_{t_2})) \to$$
$$(\langle p, f_{t_1}, b_{t_1} \rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_1}), trans^{-1}(\hat{s}_{t_2}))$$

Next, we consider the case where $b_{t_2} \notin \mathcal{C}(b'')$. Let $dom(param(b_{t_2})) = \langle \overline{r_i} \rangle$ and again let $T = term(b_t)$ for brevity.

$$(p[f/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \to^{n'}$$
$$(\langle p[f/_{f_{out}}], f_{t_2}, b_t \rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ T, \hat{s}_{t_2}) \to$$
$$(\langle p[f/_{f_{out}}], f_{t_2}, b_{t_2} \rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_2}), \hat{s}_{t_2} \rangle)$$

By our strengthened inductive hypothesis, the n-th step of evaluation of f stops in function $f_{t_1} = trans(f_{t_2})$ on a terminator that transfers control from block $trans^{-1}(\langle f_{t_2}, b_t \rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. The $(n+1)$-th step of evaluation of f proceeds by applications of rule E-Switch by first taking the symmetric switch case. Here, $b_{t_2} = trans(\langle f_{t_2}, b_{t_2} \rangle)$ and the resulting register contexts are extended with the same register-value pair, which does not violate the strengthened induction.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n$$
$$(\langle p, f_{t_1}, trans^{-1}(\langle f_{t_2}, b_t \rangle) \rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ T, trans^{-1}(\hat{s}_{t_2})) \to$$
$$(\langle p, f_{t_1}, b_{t_1} \rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_1}), trans^{-1}(\hat{s}_{t_2}))$$

Altered Block Next, we cover the cases where the n'-th step of evaluation of f_{out} is earlier within a block $b \in \mathcal{C}(\{\overline{b'_i}\})$ (but not yet at the terminator). This case is trivial by application of Lemma 6.C.1.

Fresh Block First, we cover the case where the $(n' + 1)$-th step of evaluation of f_{out} transfers control from block $b_t \in \mathcal{C}(b'')$ to a block $b_{t_2} \in \mathcal{C}(b')$.

$$(p[^{f}/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \to^{n'}$$

$$(\langle p[^{f}/_{f_{out}}], f_{t_2}, b_t\rangle \mid \gamma_2 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{branch}\ \ell_{b'}(\overline{r'_i}), \hat{s}_{t_2}) \to$$

$$(\langle p[^{f}/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(r'_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_2}), \hat{s}_{t_2})$$

By our strengthened inductive hypothesis, the n-th step of evaluation of f stops in function $f_{t_1} = \mathit{trans}^{-1}(f_{t_2})$ on a terminator that transfers control from block $\mathit{trans}^{-1}(\langle f_{t_2}, b_t\rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. The $(n + 1)$-th step of evaluation of f proceeds by application of rule E-Switch by taking the symmetric switch case. Here, $b_{t_1} \in \mathcal{C}(b')$.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^{n}$$

$$(\langle p, f_{t_1}, \mathit{trans}^{-1}(\langle f_{t_1}, b_t\rangle)\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ \mathit{trans}^{-1}(\langle f_{t_1}, b_t, \mathit{stream}(b_t)\rangle, \mathit{trans}^{-1}(\hat{s}_{t_2}))$$

No additional steps of evaluation are necessary. The blocks and their unmodified streams are consistent with the strengthened induction, and γ_2 asymmetrically gains the set of registers defined by the parameters of a block in $\mathcal{C}(b'')$. □

Corollary 7.C.2. The set of paths over block labels from ℓ_0 to ℓ_b in $G_{f_{out}}$ for some block $b \in B$ where $\mathit{lab}(b) = \ell_b$ can be constructed by replacing the edge $(\mathit{lab}(b_i), \ell_{b'})$ by the path $\langle \mathit{lab}(b_i), \ell_{b''}, \ell_{b'}\rangle$ for every block $b_i \in \mathit{dom}(E)$ in the set of such paths in G_f.

Because the input of the canonicalization procedure accepts a function which is not necessarily in canonical form, we must prove an alternate form of Theorem 6.2.2 below. For edge set splitting, we state that no *new* canonical form violation is introduced by the procedure.

Theorem 7.C.3. Let F_{out} be the loop nesting forest reconstructed from $(H_{out}, L_{out}, X_{out})$. Let $l \in F_f$ and $l' \in F_{out}$ be loops that share a header label. If l' violates a canonical form property, then l must violate the same property.

Proof. We prove no additional property violations are introduced by contradiction:

1. Suppose loop l has a single latch in f but l' has multiple latches in f_{out}. For this to occur, the edge $(b'', header_{f_{out}}(l'))$ and $(latch_{f_{out}}(l'), header_{f_{out}}(l'))$ must both exist in $G_{f_{out}}$ such that $header_{f_{out}}(l') \prec_{f_{out}} b''$. If $(latch_f(l), header_f(l))$ is split, then no other edge can be split by the input precondition and b'' is the unique latch of l'. For any other split, $b'' \notin body_{f_{out}}(l')$ and $header_{f_{out}}(l') \not\prec_{f_{out}} b''$.

2. Suppose loop l has a dedicated preheader but l' does not. First, suppose b'' is a second preheader of l'. Then, $b'' \in pred_{f_{out}}(header_{f_{out}}(l'))$ and $b'' \notin body_{f_{out}}(l)$ but splitting any backedge of l places b'' in $body_{f_{out}}(l')$ and splitting $(preheader_f(l), header_f(l))$ makes b'' the unique preheader of l'. Next, suppose that the unique preheader of l' is not dedicated. Then, $b'' \in succ_{f_{out}}(preheader_{f_{out}}(l'))$ but splitting $(preheader_f(l), header_f(l))$ makes b'' the dedicated preheader of l'.

3. Suppose $e \in exit_{f_{out}}(l')$ is an undedicated exit of l', but l has no undedicated exits. First, suppose $e = b''$. It must be the case that some $b_i \in body_f(l)$ and some $b_i \notin body_f(l)$. This implies that $b' \in exit_f(l)$ is an undedicated exit of l. Otherwise, suppose e was created by mutation of $e \in B$ and the number of edges terminating at e in f_{out} must be strictly greater than the number of edges terminating at e in f. By Corollary 7.C.2, no such additional edge exists in f_{out}.

□

Theorem 7.C.4. If f is in LCSSA form, then f_{out} is in LCSSA form.

Proof. Let $l \in F_f$ and $l' \in F_{out}$ be loops with the same header label. If $b'' \notin body_{f'}(l')$, then the set of uses that occur in l and l' are identical. Otherwise, l' additionally contains the uses which occur in z, but the registers used are defined in b'' and thus do not escape l'. □

Theorem 7.C.5. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f_{out}$ is well-typed.

Proof. As b'' contains no uses of registers outside its own parameters, $implicit(b'') = \emptyset$. As $succ_{f'}(b'') = \{b'\}$, $nonlocal(b'') = nonlocal(b')$. The fresh block b'' has the same number and type of parameters as block b'. The block reference from each block $b_i \in dom(E)$ to b' is well-typed by application of rule T-Ref. Therefore, the block reference from each block t_i to b'' is also well-typed by application of rule T-Ref. Block b'' is well-typed by application of rules T-Block, T-Switch, and T-Ref. Any other rule used to type f differs only by the inclusion of the fresh registers in the register environment typing, which Lemma 3.A.5 shows are not meaningful. □

Theorem 7.C.6. The unique dominator tree of $G_{f_{out}}$ is D_{out}.

Proof. It follows from Corollary 7.C.2 that $idom_{f_{out}}(block_{f_{out}}(\ell_b)) = block_{f_{out}}(p_D(b))$ for each block $b \in B \setminus \{b'\}$. D and D_{out} differ only by the parent of $\ell_{b''}$ and (conditionally) the parent of $\ell_{b'}$. As shown by Alstrup and Laurisden [9], $idom_{f_{out}}(b'') = nca_D(pred_{f'}(b''))$. Lastly, $idom_{f'}(block_{f'}(\ell_{b'})) = z$ when E forms the set of forward edges terminating at b' in G_f and $idom_{f'}(block_{f'}(\ell_{b'})) = block_{f_{out}}(lab(p_D(b')))$ otherwise. □

Theorem 7.C.7. F_{out} reconstructed from $(H_{out}, L_{out}, X_{out})$ is the unique loop nesting forest of $G_{f_{out}}$.

Proof. It follows from Corollary 7.C.2 that for every strongly connected component $C \subseteq B$ of G_f, there is a strongly connected component C' of $G_{f_{out}}$ such that C' contains blocks with a label of every block in C and (conditionally) block b''. All edges terminating at b''

are necessarily forward edges, as the fresh block b'' cannot possibly dominate any block in $dom(E)$, thus b'' cannot be a loop header. Then, b'' must belong to the strongly connected components described by the ancestors of $l = nca_{F_f}(\{loop_{F_f}(b) \mid b \in dom(E) \cup \{b'\}\})$ as $header_{f_{out}}(l) \rightsquigarrow_{f_{out}} b''$ and $b'' \rightsquigarrow_{f_{out}} latch_{f_{out}}(l)$.

Let $l \in F_f$ be the loop that describes C and $l' \in F_{out}$ be the loop that describes C'. If $b' \notin exit_f(l)$ or $b' \notin body_f(l)$ for every $b_i \in dom(E)$ (if no edge exiting l is split), then the exits of l and the exits of l' consist of the same labels. Otherwise, $y \in exit_f(l)$, $b' \not\rightsquigarrow_f latch_f(l)$, b'' is reachable by all blocks that can reach b', and $b'' \not\rightsquigarrow_{f_{out}} latch_{f_{out}}(l')$, therefore b'' must be an exit of l'. Then, b' remains an exit of l' only if $b'' \notin b \rightsquigarrow_{f_{out}} y'$ for some $b \in body_{f_{out}}(l')$ (if not all exits of l terminated at b' are split). □

7.D Unique Latch

In this section, we refer to the *unique latch* operation (notated below). For specific details (including names of intermediate components), refer to Figure 7.42.

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l \in F_f]{\text{unique latch}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Theorem 7.D.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. Trivial by Theorem 7.C.1. □

Theorem 7.D.2. If f is in LCSSA form, then f_{out} is in LCSSA form.

Proof. Trivial by Theorem 7.C.4. □

Theorem 7.D.3. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f_{out}$ is well-typed.

Proof. Trivial by Theorem 7.C.5. □

Theorem 7.D.4. The unique dominator tree of $G_{f_{out}}$ is D_{out}.

Proof. Trivial by Theorem 7.C.6. □

Theorem 7.D.5. If f is in canonical form, then F_{out} reconstructed from $(H_{out}, L_{out}, X_{out})$ is the unique loop nesting forest of $G_{f_{out}}$.

Proof. Trivial by Theorem 7.C.7. □

Now, we show that each application of this operation makes a step of progress toward canonicalizing the graph. We refer to blocks b' and b'' from Figure 7.32.

Theorem 7.D.6. Let F_{out} be the loop nesting forest reconstructed from $(H_{out}, L_{out}, X_{out})$. Let $l' \in F_{out}$ be the loop that shares a header label with loop l. Then, $|latch_{f_{out}}(l')| = 1$.

Proof. The *split edge set* operation is applied such that $b' = header_f(l)$. By Corollary 7.C.2, $pred_{f_{out}}(b') \cap body_{f_{out}}(l') = latch_{f_{out}}(l') = \{b''\}$. □

Corollary 7.D.7. By Theorem 7.C.3 and Theorem 7.D.6, if f has n distinct canonical form violations, then f_{out} has $n - 1$ distinct canonical form violations.

7.E Dedicated Preheader

In this section, we refer to the *dedicate preheader* transformation (notated below). For specific details (including names of intermediate components), refer to Figure 7.44.

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l \in F_f]{\text{dedicate preheader}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Theorem 7.E.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be

evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. If $header_f(l)$ is not the entry of f, then f_{out} is constructed from an edge set split of f and this property follows from Theorem 7.C.1. In the other case, we prove by filling in the slots of Template 6.A.1.

[**Slot Refinements**] We add the following additional relationship between register contexts, which allows registers of the fresh block in the register context of the parallel evaluation. We do not modify the *trans* relation from the template.

$$\gamma_1 \subseteq \gamma_2$$

[**Slot → Invokability**] Invocation of function $f' \in \mathcal{C}(f_{out})$ proceeds with the following additional step of evaluation.

$$\rightarrow \ (\langle p[{}^{f}/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r'_{t_i} \mapsto v_{t_i}}] \mid \mu \mid \nu \mid \Psi;\ \texttt{branch}\ \ell_0(\overline{r'_i}), trans(\hat{s}_{t_1}))$$

$$\rightarrow \ (\langle p[{}^{f}/_{f_{out}}], f_{t_2}, b_{t_1}\rangle \mid \gamma_2[\overline{r_i \mapsto r'_i}] \mid \mu \mid \nu \mid \Psi;\ stream(b_{t_1}), trans(\hat{s}_{t_1}))$$

The resulting blocks are identical, and the set of register-value pairs added to register context γ_2 is a strict superset of the register-value pairs added to register context γ_1.

[**Slot → Asymmetric Evaluation**] By the refined *trans* relation, all temporarily diverging steps of evaluation occur at the entry of function f. Both the initial evaluation step of the program (via the base case) and recursive calls to the function are trivially implied by [Slot *Invokability*].

[Slot ← Refinement] We refine the translation of blocks and streams by mapping clones of the new entry block b' to clones of the original entry block h.

$$trans(\langle f_t, b_t \rangle) = \begin{cases} h[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(b') \\ b_t & otherwise \end{cases}$$

$$trans(\langle f_t, b_t, (s_t, \hat{s}_t) \rangle) = \begin{cases} stream(h)[\overrightarrow{\sigma_{f_t}}], trans(\hat{s}_t) & b_t \in \mathcal{C}(b') \\ s_t, trans(\hat{s}_t) & otherwise \end{cases}$$

Now, we define the following relationship stating that registers occurring in γ_2 but not γ_1 must be a parameter to a clone of block b'. We additionally allow the register context γ_2 to temporarily break the superset relation with register context γ_1 when evaluation of f_{out} is within a clone of block b'. It is not until the following step of evaluation that γ_2 will assign the set of registers defined by the parameters of the successor block. In the following, we use f_t and b_t to mean the function and block in the *same step of evaluation* of function f_{out} as the register contexts occur.

$$\gamma_1 \subseteq \gamma_2 \text{ if } b_t \notin \mathcal{C}(b') \qquad \{(r, cv) \in \gamma_1 \mid r \notin \{dom(param(succ_{f_t}(b_t)))\}\} \subseteq \gamma_2 \text{ if } b_t \in \mathcal{C}(b')$$

$$dom(\gamma_2 \setminus \gamma_1) \subseteq \bigcup_{b_i \in \mathcal{C}(b')} dom(param(b_i))$$

[Slot ← Invokability] As a new entry block was created, $b_{t_2} \in \mathcal{C}(b')$ and $b_{t_1} \in \mathcal{C}(h)$. The unmodified streams are consistent with the strengthened induction without an additional step of evaluation, as the translation of blocks and streams simply *cut out* clones of block b' in the evaluation of f. The resulting register contexts gain the set of registers defined by the parameters of blocks b_{t_2} and b_{t_1} for evaluation of function f_{out} and f, respectively. As evaluation of function f_{out} remains in block b', both of these contexts are consistent with the strengthened induction.

[**Slot ← Asymmetric Evaluation**] By the refined $trans^{-1}$ relation, all temporarily diverging steps of evaluation occur at the entry of function f_{out}. Both the initial evaluation step of the program (via the base case) and recursive calls to the function are trivially implied by [Slot *Invokability*]. □

If $header_f(l)$ is not the entry of f, then f_{out} is constructed from an edge set split of f and the following maintenance theorems follow from application of Theorem 7.C.7 through Theorem 7.C.4. In the following proofs we assume $header_f(l)$ is the entry of f.

Corollary 7.E.2. The set of paths over bock labels from ℓ any block reachable in $G_{f_{out}}$ can be constructed by prepending the label ℓ'_0 to the set of such paths in G_f.

Theorem 7.E.3. If f is in LCSSA form, then f_{out} is in LCSSA form.

Proof. Block b' introduces uses which are all defined within block b'. □

Theorem 7.E.4. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f_{out}$ is well-typed.

Proof. Otherwise, $implicit(b') = \emptyset$ as b' contains no uses of registers outside its own parameters and $nonlocal(b') = nonlocal(header_f(l))$ as $succ_{f_{out}}(b') = \{header_f(l)\}$. The fresh block b' has the same number and type of parameters as the header of f. Block b' is well-typed by application of rules T-Block, T-Switch, and T-Ref. Any other rule used to type f differs only by the inclusion of the fresh registers in the register environment typing, which Lemma 3.A.5 shows are not meaningful. □

Theorem 7.E.5. The unique dominator tree of $G_{f_{out}}$ is D_{out}.

Proof. Otherwise, it follows from Corollary 7.E.2 that b' dominates all blocks in B' and $idom_{f_{out}}(block_{f_{out}}(\ell_b)) = block_{f_{out}}(p_D(b))$ for each block $b \in B$. The only difference between D and D_{out} is the root, which is b' in D_{out}. □

Theorem 7.E.6. F_{out} reconstructed from $(H_{out}, L_{out}, X_{out})$ is the unique loop nesting forest of $G_{f_{out}}$.

Proof. Otherwise, it follows from Corollary 7.E.2 that the strongly connected components of G_f and $G_{f_{out}}$ are isomorphic. F and F_{out} are similarly isomorphic. □

Now, we show that this each application of this operation makes progress toward canonicalizing the graph (at this point we no longer assume that $header_f(l)$ is the entry of f). In the following, we refer to blocks b' and b'' from Figure 7.32.

Theorem 7.E.7. Let F_{out} be the loop nesting forest reconstructed from $(H_{out}, L_{out}, X_{out})$. Let $l' \in F$ and $l'' \in F_{out}$ be loops that share a header label. If l' violates a canonical form property, then l must violate the same property.

Proof. If $header_f(l)$ is not the entry of f, then f_{out} is constructed from an edge set split of f and this property follows from Theorem 7.C.3. Otherwise, l'' must violate the same unique latch and dedicated exit properties as l', as $E(G_{f_{out}}) = E(G_f) \cup \{(b', entry(f))\}$. If l'' has no dedicated preheader, then $l' \neq l$ by Theorem 7.E.8 and l' violates the dedicated preheader property as $header(l') \neq entry_f(f)$ and $pred_{f_{out}}(block_{f_{out}}(lab(l'))) = pred_f(header_f(l'))$. □

Theorem 7.E.8. Let F_{out} be the loop nesting forest reconstructed from $(H_{out}, L_{out}, X_{out})$. Let $l' \in F_{out}$ be the loop that has the header label $lab(l)$. Then, l' has a unique dedicated preheader in f_{out}.

Proof. $b' = header_{f'}(l')$ and $pred_{f_{out}}(b') = latch_{f_{out}}(l') \cup \{b''\}$. If $header_f(l)$ is not the entry of f then this property follows from Corollary 7.C.2. Otherwise, this property follows from Corollary 7.E.2 otherwise. □

Corollary 7.E.9. By Theorem 7.E.7 and Theorem 7.D.6, if f has n distinct canonical form violations, then f_{out} has $n - 1$ distinct canonical form violations.

7.F Dedicated Exits

In this section, we refer to the *dedicate exit* operation (notated below). For specific details (including names of intermediate components), refer to Figure 7.46.

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l \in F_f,\ e \in exit_f(l)]{\text{dedicate exit}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Theorem 7.F.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. Trivial by Theorem 7.C.1. □

Theorem 7.F.2. If f is in LCSSA form, then f_{out} is in LCSSA form.

Proof. Trivial by Theorem 7.C.4. □

Theorem 7.F.3. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f_{out}$ is well-typed.

Proof. Trivial by Theorem 7.C.5. □

Theorem 7.F.4. The unique dominator tree of $G_{f_{out}}$ is D_{out}.

Proof. Trivial by Theorem 7.C.6. □

Theorem 7.F.5. If f is in canonical form, then F_{out} reconstructed from $(H_{out}, L_{out}, X_{out})$ is the unique loop nesting forest of $G_{f_{out}}$.

Proof. Trivial by Theorem 7.C.7. □

Now, we show that each application of this operation makes a step of progress toward canonicalizing the graph. In the following, we refer to blocks b' and b'' from Figure 7.32.

Theorem 7.F.6. Let F_{out} be the loop nesting forest reconstructed from $(H_{out}, L_{out}, X_{out})$. Let $l' \in F_{out}$ be the loop that shares a header label with loop l. Then, assuming l has k undedicated exits, l' has $k-1$ undedicated exits with respect to f_{out}.

Proof. Block b'' occurs before block e on every path from any block $b \in body_{f_{out}}(l')$ by Corollary 7.C.2. Therefore, e (an undedicated exit of l) is not an exit of l'. Additionally, b'' is a dedicated exit of l' as $dom(E) \subseteq body_f(l)$. □

Corollary 7.F.7. By Theorem 7.C.3 and Theorem 7.D.6, if f has n distinct canonical form violations, then f_{out} has $n-1$ distinct canonical form violations.

8 Operations

In this chapter we define a set of procedures that are used by the transformations present in Chapter 9.

8.1 Block Ejection

During a transformation that may change the set of paths over the control flow graph, a block may find itself in the wrong loop. This situation can be easily created, for example, by simply deleting an edge and removing the only path from a block to the latch of its loop. This section details the repair operation when such a situation occurs.

This operation takes as input a *claimed* loop nesting forest F that may not be equivalent to a deconstruction of the correct loop nesting forest F_f. Here, we make the following five assumptions about the nature of the input.

1. The actual loops of the function and loops in the claimed loop nesting forest have a one-to-one correspondence, although the loop may have an inaccurate parent, body set, or exit set. Formally, $\{lab(l) \mid l \in F_f\} = \{lab(l) \mid l \in F\}$.

2. The body sets of the claimed loop nesting forest maintain a subset lattice (just as well-formed loop nesting forests do). Formally, $body(l) \subset body(p_F(l))$ for every $l \in F$ where $l \neq l_\varnothing$.

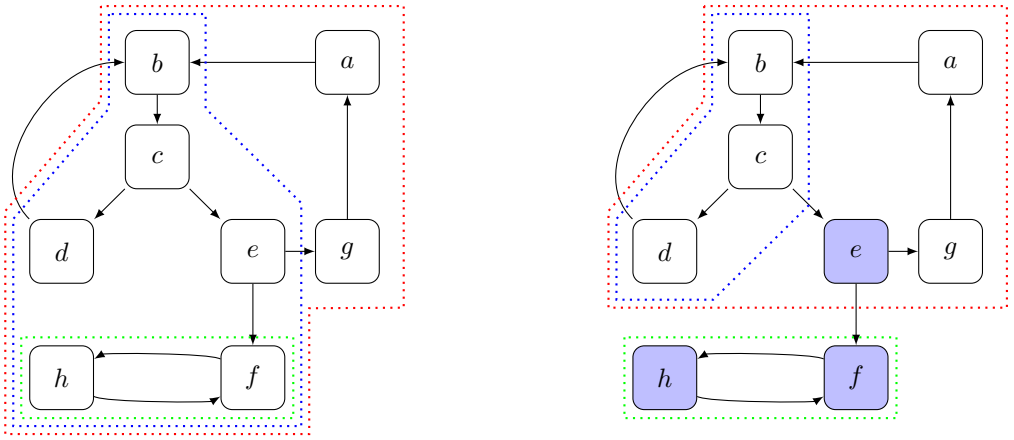
3. The body set of each violating loop in the claimed loop nesting forest is a superset of the body set of the loop in the function with the same header. Formally, $body(l') \supseteq body(l)$ for every $l' \in F_f$, $l \in F$ such that $lab(l') = lab(l)$.

4. The exit set of each violating loop in the claimed loop nesting forest is correct with respect to body set of the same loop – specifically, the exit set must consist of all the blocks that have a predecessor in the body set but are not themselves in the body set. Formally, $exit(l) = \{b \in body(l) \mid b \notin body(l) \wedge pred(b) \cap body(l) \neq \emptyset\}$ for every $l \in F$.

5. Every loop with an inaccurate body set is an ancestor of the deepest inaccurate loop in the claimed loop nesting forest. Alternatively, the set of all loops with an inaccurate body set can be found on a single path in the claimed loop nesting forest.

These assumptions may appear rather strict, but are the natural properties of the loop nesting forest after an edge deletion and after insertion of blocks on an edge (specifically, see function inlining discussed in Section 9.42). Figure 8.11 illustrates a repair of such a loop nesting forest. The graph of this example may have been created by removing the edge (f, d) or by inserting the subgraph consisting of blocks e, f, and h on the edge (c, g), among other possibilities. To repair the graph, block e is ejected from the blue loop (but remains in the red loop as there exists a path to the latch), and blocks f and h are ejected from both loops. This transformation also alters the exit sets so that g is no longer an exit of the red loop but f is, and e becomes the sole exit of the red loop. Notice that as the entire body of the green loop is ejected from its inner loop, the parents of the loops in the loop nesting forest also change. Notice that the exit sets of loops may also need to be repaired.

This operation is formally described by Figure 8.12. Note that we pass in the loop triple (H_F, L_F, X_F) distinct from $(H_{F_f}, L_{F_f}, X_{F_f})$, as the input loop nesting forest does not necessarily describe the input function. It is recursive in nature and begins in the deepest violating loop, working its way up to a root of the forest (the no-op base case). For each loop, we partition the body of the loop into the set B_{in}, the true body of the loop, and the set B_{out}, the extraneous blocks. The extraneous blocks are pruned from the body of the loop and, if a loop structure was ejected with it, then children loop entries are adopted by



(a) The broken loop nesting forest. (b) The repaired loop nesting forest.

Figure 8.11: Repairing loop nesting forest by ejecting blocks from a loop.

the loop's current parent. In either case, this does not remove any blocks from the body of the loop's current parent (which is taken care of in a subsequent call). Then, the loop's exit set is recalculated to determine which newly ejected blocks are new exits and which old exits no longer have predecessors in the loop body.

After ejection, the dedicated exit property and loop-closed SSA form may be broken. Figure 8.13 illustrates the former case in which a block and one of its predecessors are ejected from the loop. The *dedicate exit* operation discussed in Section 7.4.3 is applied over violating exit blocks in order to repair the temporary violation of canonical form. Figure 8.14 illustrates the latter case in which a block containing a legal use of a register is ejected from the loop that defines that register. The LCSSA reconstruction operation discussed in Section 7.2 is applied over such violating uses to repair the temporary violation of LCSSA form.

8.2 Edge Deletion

A program may have a control flow edge that is provably *dead* such that no set of inputs will cause the evaluation of the program to take that edge. Such edges often occur as a

$$\frac{D_{out} = D_f \qquad H_{out} = H_F \qquad L_{out} = L_F \qquad X_{out} = X_F}{(f, D_f, H_F, L_F, X_F) \xrightarrow[\langle l_\varnothing \rangle]{\text{eject}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})}$$

$$\frac{\begin{array}{c}(l_p = l_\varnothing) \vee (l \subset l_p) \qquad B_{in} = \{b \in body_f(l) \mid b \rightsquigarrow_{G_f \langle body_f(l) \rangle} latch_f(l)\} \\ B_{out} = body_f(l) \setminus B_{in} \qquad H' = H[header_f(l') \mapsto header_f(l_p) \mid l' \in F \wedge header_f(l') \in B_{out}] \\ L' = L[b \mapsto l_p \mid b \in B_{out}] \qquad X_{old} = \{(\ell, \ell_h) \in X \mid pred_f(block_f(\ell)) \cap B_{in} = \emptyset\} \\ X_{new} = \{(lab(b), lab(l)) \mid b \in B_{out}, pred_f(b) \cap B_{in} \neq \emptyset\} \qquad X' = X \setminus X_{old} \cup X_{new} \\ exits = \{block_f(\ell_1) \mid (\ell_1, \ell_h) \in X' \wedge pred_f(block_f(\ell_1)) \cap B_{out} = \emptyset\} \\ defs = \{d \in def_f(r) \mid d \text{ occurs in block of } B_{in} \wedge \text{use of } r \text{ occurs in block of } B_{out}\} \\ (f, D_f, H', L', X') \xrightarrow[\{(l,e) \mid e \in exits\}]{\text{dedicate exits}}{}^{*} \cdot \xrightarrow[\{(l,d) \mid d \in defs\}]{\text{repair lcssa}}{}^{*} (f', D', H'', L'', X'') \\ (f', D', H'', L'', X'') \xrightarrow[\langle l_p, \ldots, l_\varnothing \rangle]{\text{eject}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})\end{array}}{(f, D_f, H_F, L_F, X_F) \xrightarrow[\langle l, l_p, \ldots, l_\varnothing \rangle]{\text{eject}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})}$$

Figure 8.12: Ejecting nodes to restore loop nesting forest.

result of another transformation, such as function inlining or loop unswitching. Removing this edge results in a reduction of program size, and may allow additional opportunities for optimization (e.g. reducing the size of a function below the inline limit, removing dead side effects from a loop for more precise analysis). This section details the restoration of canonical form after the deletion of such a dead edge.

The deletion of an edge has what appears to be non-local changes to the graph. Many

Figure 8.13: Block ejection creating an undedicated exit.

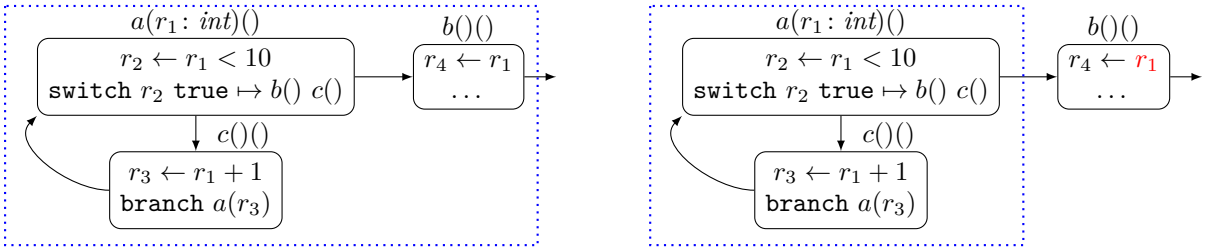
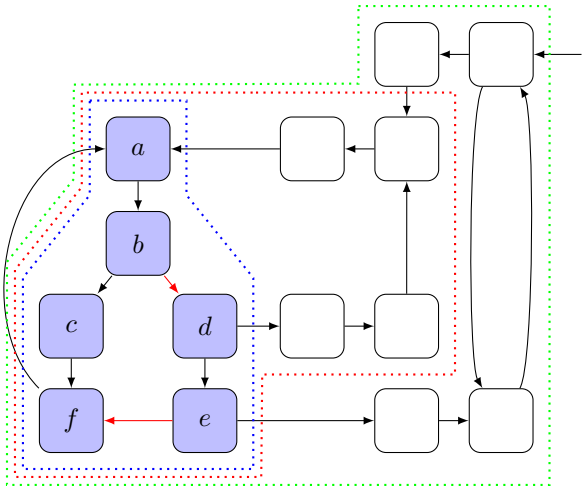


Figure 8.14: Block ejection creating an LCSSA violation.

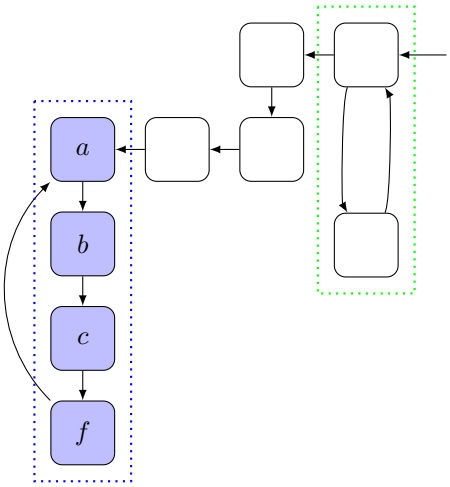
nodes can become unreachable if the only paths to them from the block contain the deleted edge. Entire loops may be destroyed and unnested from their parent. Figure 8.21 illustrates such changes. Removing the edge (b, d) from the initial graph causes the intermediate red loop to lose its backedge, causing the set of blocks composing the loop body to become unconnected. The blue and green loops survive, but the inner loop is no longer nested within the outer loop as there is no longer a path that connects the inner loop to the latch of the outer loop. Removing the edge (e, f) from the initial graph causes part of the body of the inner loop to spill out into its parent loops. Blocks that once belonged to the inner loop no longer have a path to the loop's latch, forcing them out into upper levels of the loop nesting forest.

This operation requires the condition that, in order to remove edge (a, b) from the graph, block a must transfer control to block b through a switch case whose value is `unit`. This ensures that the deletion of an edge neither creates an unterminated block by removing the last edge out of a block, nor arbitrarily changes the semantics of the program (as this case can never be evaluated).

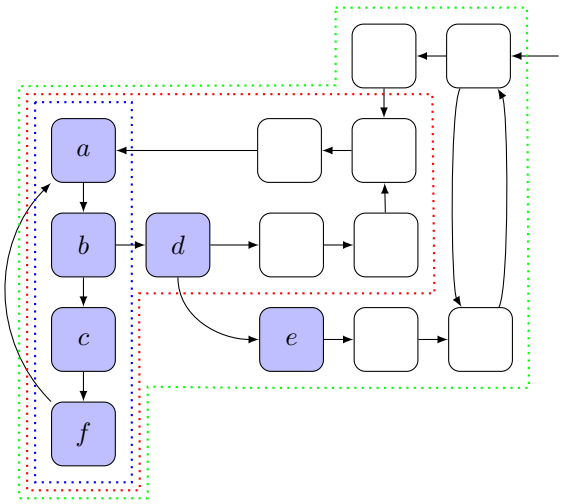
We present this operation in two parts. Section 8.2.1 presents the situation where the paths of the graph change only in *multiplicity*, but the set of paths over the graph are otherwise unchanged. Section 8.2.2 presents the situation where the last edge between two adjacent blocks is removed.



(a) The initial graph.



(b) Graph after removing the edge (b, d).



(c) Graph after removing the edge (e, f).

Figure 8.21: Examples of loop structure changes that can occur after an edge deletion.

8.2.1 Change in Path Multiplicity

First, we cover the situation where one of several edges between blocks a and b is deleted. This case of the operation is formally described by Figure 8.22.

The bulk of the work is offloaded onto an operation that removes a block reference from the terminator of block a. The case that is removed from the switch must refer to block b and must have a switch value of **unit**, but does not necessarily need to be the first or only such case in the terminator. The only interesting part of this operation is the construction of

$$\frac{(a,b) \in^{k} E(f) \qquad k > 1 \qquad f \xrightarrow[(a,b)]{\text{delete case}} f_{out} \qquad D_{out} = D_f \qquad H_{out} = H_{F_f} \qquad L_{out} = L_{F_f} \qquad X_{out} = X_{F_f}}{(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[(a,b)]{\text{delete edge}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})}$$

$$\frac{\begin{array}{c} f = (\ell, t, B, \ell_0) \qquad term(a) = \texttt{switch}\ e\ \overline{cv_i \mapsto ref_i}\ (\texttt{unit} \mapsto ref_b)\ \overline{cv_j \mapsto ref_j}\ ref_d \\ lab(b) = \ell_b \qquad ref_b = \ell_b(\overline{e_k}) \\ T = \texttt{switch}\ e\ \overline{cv_i \mapsto ref_i}\ \overline{cv_j \mapsto ref_j}\ ref_d \qquad imp = \{(r\colon t) \in implicit(a) \mid use_a(r) \neq \{ref_b\}\} \\ a' = (lab(a), param(a), imp, inst(a) \cup \langle T \rangle) \qquad B' = B \setminus \{a\} \cup \{a'\} \qquad f_{out} = (\ell, t, B', \ell_0) \end{array}}{f \xrightarrow[(a,b)]{\text{delete case}} f_{out}}$$

Figure 8.22: Replacing a dead block reference with a syntactically unreachable switch case.

the implicit values of the modified block a' that are constructed by removing typed registers from $implicit(a)$, whose only use occurred in the removed block reference.

8.2.2 Change in Paths

Now, we cover the situation where the last edge between blocks a and b are deleted. We organize this situation in two parts. First, we remove the last block reference to block b from block a via *delete case* operation. Then, we prune the loop body, exit, and nesting structures to remove references to blocks that are unreachable and remove references to loops that no longer have a stable backedge. This case of the operation is described by Figure 8.23. Notice that this transformation also takes a dominator tree and loop nesting forest that do not necessarily describe the input function. The resulting loop triple (H', L', X') forms a good approximation of the correct resulting function (in which the properties of the input of the *eject* transformation discussed in Section 8.1 are validated). Finally, we use the *eject* procedure presented in Section 8.1 to perform local updates over the loop nesting structures

$$\frac{(a,b) \in^1 E(f) \qquad f \xrightarrow[(a,b)]{\text{delete case}} f' \qquad (f', D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[(a,b)]{\text{post-delete edge}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})}{(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[(a,b)]{\text{delete edge}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})}$$

$$\frac{\begin{array}{c}(f, D) \xrightarrow[(a,b)]{\text{repair dominator}} (R, D') \qquad f = (\ell, t, B, \ell_0) \qquad B' = B \setminus R \\ f' = (\ell, t, B', \ell_0) \qquad dead = \{l \in F \mid latch_f(l) \in R \vee (a = latch_f(l) \wedge b = header_f(l))\} \\ A = \{l \mapsto p \mid l \in F \wedge p \text{ is lowest ancestor of } l \text{ such that } p \notin dead\} \\ H' = \{(lab(header_f(l)), lab(header_f(A(p_F(l))))) \mid l \in F \setminus dead\} \\ L' = \{(\ell, lab(A(l))) \mid (\ell, lab(l)) \in L \wedge block_f(\ell) \notin R\} \\ X' = \{(\ell, lab(l)) \in X \mid block_f(\ell) \notin R \wedge l \notin dead\} \\ (f', D', H', L', X') \xrightarrow[\langle A(loop(a)), A(A(loop(a))), \ldots, l_\varnothing \rangle]{\text{eject}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})\end{array}}{(f, D, H_F, L_F, X_F) \xrightarrow[(a,b)]{\text{post-delete edge}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})}$$

Figure 8.23: Restoring canonical properties after deletion of the dead edge (a, b).

to correct inconsistencies in the approximate function.

This case of the operation is formally described by Figure 8.23. Notice that the input dominator tree and loop nesting triple are not necessarily the correct structures for the input function. To repair the dominator tree after edge deletion, we use the algorithm described by Ramalingam and Reps [53] that yields a set of unreachable blocks, denoted by R, and a dominator tree D'.

The intermediate function f' is then constructed by simply removing all unreachable blocks in the set R. No block or control flow edge is otherwise modified. Thus, for all $\ell \in \{lab(b) \mid b \in B \setminus R\}$, $block_f(\ell) = block_{f'}(\ell)$. Every block in $B \setminus R$ is necessarily reachable from $block_{f'}(\ell_0)$ in $G_{f'}$ and D' is the correct dominator tree of f'.

Let *dead* be the set of loops from the claimed loop nesting forest F whose backedges are

unreachable after the deletion of edge (a, b). The triple (H', L', X') describing an approximate loop structure of f' is constructed by removing all references to unreachable blocks and removing all references to loops in *dead* from the loop structure of f. The set H' is constructed so that a live loop with a dead parent is made a child of its nearest living ancestor. The set L' is constructed so that a block b such that $loop_f(b) = l \in dead$ remains in the bodies of its living ancestors. The set X' is constructed by removing references to unreachable blocks and removing references to loops that were destroyed by the deletion of their latch.

8.3 Loop Duplication

In contrast to deletion, a transformation may also add additional paths and/or blocks to an existing control flow graph. However, a valid transformation will never add an edge between two *arbitrary* blocks of an existing graph, nor make an *arbitrary* subgraph reachable from an existing subgraph. Instead, additive transformations generally start by *duplicating* a portion of an existing graph (either the same function in the case of loop transformations, or another function in the case of function inlining), then *linking* the copy of the subgraph into the existing control flow graph. This section details the common operation of duplicating a targeted subgraph induced by the body of a loop. As linking the copy of the subgraph differs with the transformation context, the remainder of the operation will be detailed in Chapter 9.

This operation is formally described by Figure 8.31. It takes as input a function along with its dominator tree and loop nesting forest as well as a target loop l. The resulting function contains a copy of the blocks contained in loop l as well as a copy of any edges leaving a duplicated block. Specifically, if an edge from a duplicated block to another duplicated

$$\sigma_L = \{(lab(b), \ell'_i) \mid b \in body_f(l)\}$$

ℓ'_i is fresh $\qquad \sigma_B = \{(b, (\sigma_L(lab(b)), param(b), implicit(b), stream(b)[\sigma_L])) \mid b \in body_f(l)\}$

$$f = (\ell, t, B, \ell_0) \qquad B' = B \cup \{b' \mid (b, b') \in \sigma_B\}$$

$f_{out} = (\ell, t, B', \ell_0) \qquad l' = l[\sigma_L] \qquad D_{out} = D_f \cup \{(lab(\sigma_B(b)), \sigma_L(p_D(b))) \mid b \in body_f(l)\}$

$$headers = \{lab(header(l')) \mid l' \in F_f\langle l \rangle\}$$

$$H_{out} = H_{F_f} \cup \{(\ell_c, \ell_p) \in H_{F_f} \mid \ell_c \in headers\}[\sigma_L]$$

$$L_{out} = L_{F_f} \cup \{(\ell_b, \ell_h) \in L_{F_f} \mid \ell_h \in headers\}[\sigma_L]$$

$$X_{out} = X_{F_f} \cup \{(\ell_b, \ell_h) \in X_{F_f} \mid l_h \in headers\}[\sigma_L]$$

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l]{\text{duplicate}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out}), (l', \sigma_L)$$

Figure 8.31: Duplicating the body (and auxiliary data structures) of a loop to be later linked into the graph.

block exists, it will also exist between the two copies, and if an edge from a duplicated block to a non-duplicated block exists, it will also exist between the copied block and the original block. However, no edges are created such that a non-duplicated block points to a new block. Thus, the resulting function contains a strongly connected component unreachable from the entry block. This behavior is illustrated in Figure 8.32.

In order to duplicate blocks, a mapping σ_L from block labels to block labels is populated.

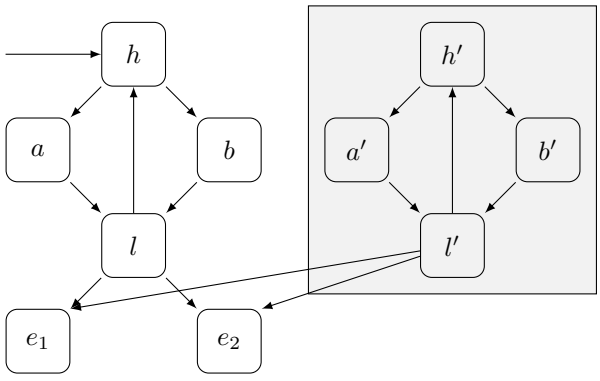


Figure 8.32: Illustration of *new* edges after duplicating a loop structure. The copy of the subgraph is enclosed in a gray box, unreachable from the original graph.

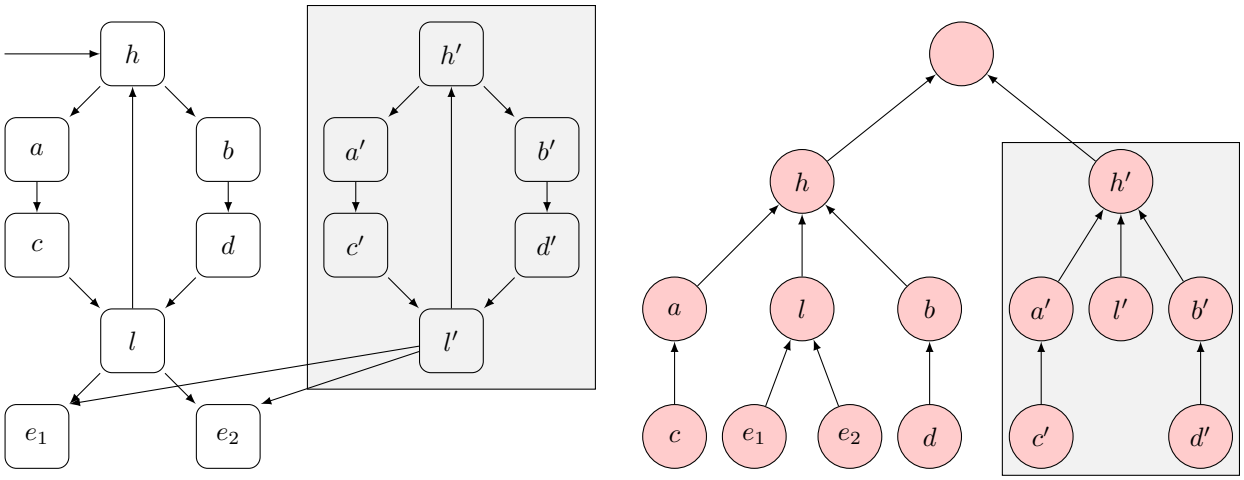


Figure 8.33: Illustration of the new dominator subtree (enclosed in a gray box) after duplicating the body of a loop.

This mapping correlates the label of a duplicated block to the label of its copy. When a block is duplicated, all block label references are substituted with respect to σ_L. The pair of originating blocks and its clone (b, b') are added to the mapping σ_B. This implicitly creates the edges as described above – targets of a copied block's terminator are translated so that the branching structure of the subgraph remains symmetric to the original. The populated mapping is also returned as a side-effect of the operation.

The resulting dominator tree is the input dominator tree unioned with the dominator tree of the cloned subgraph. The new subtree is placed such that $header(l)$ and its duplicated block share the same immediate dominator. This will generally be correct, but some transformations (loop peeling, to name one) may need to refine this assumption. It is also worth nothing that no existing block changes dominators with respect to the entry block of the function. Figure 8.33 illustrates the duplication of the dominator tree.

The parent, body set, and exit set of each loop nested under l are duplicated and translated with respect to the block relation σ_L. This results in a loop nesting forest with an additional root, returned as l', encoding the structure of the not-yet-reachable subgraph. Figure 8.34 illustrates the duplication of the loop nesting forest.

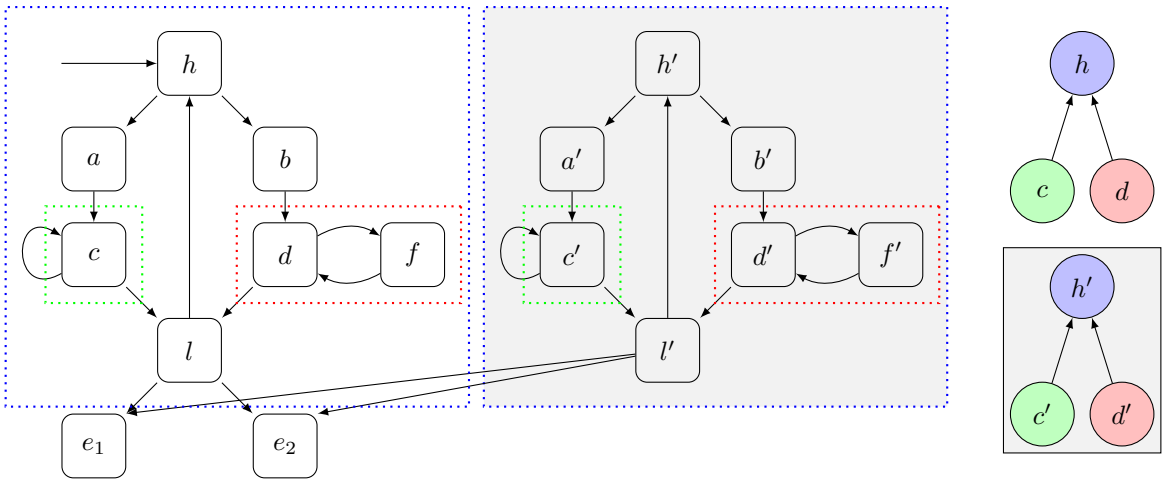


Figure 8.34: Illustration of the new loop nesting forest subtree (enclosed in a gray box) after duplicating the body of the outermost shown loop.

Appendix

Here we provide the proofs of maintenance properties of operations presented in Chapter 8.

8.A Block Ejection

In this section, we refer the *eject* transformation (notated below). For specific details (including names of intermediate components), refer to Figure 8.12.

$$(f, D_f, H_F, L_F, X_F) \xrightarrow[\langle l, l_p, \ldots, l_\varnothing \rangle]{\text{eject}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Theorem 8.A.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. Trivial by repeated application of Theorem 7.F.1 and Theorem 7.B.2. □

Theorem 8.A.2. If f is in canonical form with respect to the F, then f_{out} is in canonical form (with respect to $F_{f_{out}}$).

Proof. Let $l' \in F_{out}$ and $l'' \in F$ be loops that share a header label. We prove no property violations for loop l' exist. First, l' and l'' have the same dedicated preheaders and latch. As these blocks are not in violation in F, they are also not in violation in F_out. Now, suppose

that $e \in \mathit{exit}_{f_{out}}(l')$ is not a dedicated exit of l'. To remain undedicated, it must be the case that $\mathit{lab}(e) \notin \mathit{exit}$. This means that $\mathit{pred}_f(e) \cap B_{out} = \emptyset$ and the predecessor that causes e to be undedicated to l' also makes it undedicated to l''. □

In the following, we refer to the triples (H', L', X') and (H'', L'', X''), the sets of blocks B_{in}, B_{out}, and exit, and the set of definitions defs from Figure 8.12.

Theorem 8.A.3. If all uses of each register defined in function f occur within the same loop of F as its definition, then f_{out} is in LCSSA form.

Proof. Let $l' \in F_{out}$ and $l'' \in F$ be loops that share a header label such that loop l' contains a definition d of register r and a use of r occurs in block $b \notin \mathit{body}_{f_{out}}(l)$. By construction of F_{out}, $\mathit{body}(l') \subseteq \mathit{body}(l'')$. If $b \notin B_{out}$, then by assumption that uses of r is closed with respect to F, no such uses of r occur outside of l'. Otherwise, $d \in \mathit{defs}$ by construction and all such uses of r occurring outside of l' are replaced with fresh registers defined outside of l' by application of the *repair lcssa* operation. □

Theorem 8.A.4. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f_{out}$ is well-typed.

Proof. Trivial by repeated application of Theorem 7.F.1 and Theorem 7.B.2. □

Theorem 8.A.5. The unique dominator tree of $G_{f_{out}}$ is D_{out}.

Proof. Trivial by repeated application of Theorem 7.F.1. □

In the following, let F' be the intermediate loop nesting forest (in which body sets, exit sets, and loop nesting is altered) reconstructed from (H', L', X'), let F'' be the loop nesting forest (in which exits are re-dedicated and LCSSA violations are repaired) reconstructed from (H'', L'', X''), and let F_{out} be the loop nesting forest (yielded from the recursive call) reconstructed from $(H_{out}, L_{out}, X_{out})$. Now, we show that *eject* can repair the input loop

nesting forest as long as it is broken in the expected manner with respect to the input function. We prove this in three parts: once for F', once for F'', and once for F_{out}.

Lemma 8.A.6. Let $l' \in F$ be a loop nested in l. If $b \in \mathit{body}(l') \in O$, then $\mathit{body}(l') \subseteq O$.

Proof. Suppose another block $b' \in \mathit{body}(l') \notin O$. Then, there is a path from b' to the latch of l in the subgraph induced by $\mathit{body}(l)$. However, as $\mathit{body}(l')$ is a strongly connected subset of $\mathit{body}(l)$, there must be such a path for b as well. □

Lemma 8.A.7. Let $l' \in F'$ and $l'' \in F_f$ be loops such that $\mathit{lab}(l') = \mathit{lab}(l'') = \mathit{lab}(l)$. Then, if F validates the five assumptions given in Section 8.1, the body and exit sets of l' are correct with respect to function f and F' validates the same assumptions.

Proof. First, we show that l' has the correct body set. Suppose this is not the case and $b \in \mathit{body}(l') \setminus \mathit{body}(l'')$ (i.e. the body set of l' contains an extra block). However, b's presence in $\mathit{body}(l')$ ensures a path from b to the latch of l' so it must be the case that $b \in \mathit{body}(l'')$. In the other direction, suppose $b \in \mathit{body}(l'') \setminus \mathit{body}(l')$ (i.e. the body set of l' is missing a block). By assumption, $\mathit{body}(l) \supseteq \mathit{body}(l'')$ and $\mathit{body}(l') = \mathit{body}(l) \setminus O$ by construction. Thus, it must be the case that $b \in O$. However, b's presence in O indicates the absence of a path from b to the latch of l', so it must be the case that $b \notin \mathit{body}(l'')$.

Next, we show that l' has the correct exit set. Suppose this is not the case and $b \in \mathit{exit}(l') \setminus \mathit{exit}(l'')$ (i.e. the exit set of l' contains an extra block). As b is not an exit of l'', then either $b \in \mathit{body}(l'')$ or $\mathit{pred}(b) \cap \mathit{body}(l'') = \emptyset$. In the former case, $b \in I$ and $b \notin \mathit{exit}(l')$ as $(\mathit{lab}(b), \mathit{lab}(l)) \notin X$ and $b \notin O$. The latter case cannot occur, as X' must have a predecessor in I by construction. In the other direction, suppose $b \in \mathit{exit}(l') \setminus \mathit{exit}(l'')$ (i.e. the exit set of l'' is missing a block). By construction of X', $b \notin O$ and $b \notin \mathit{exit}(l)$. Then, $\mathit{pred}(b) \cap I = \emptyset$ and $b \notin \mathit{exit}(l')$.

Finally, we show that F' validates the same five assumptions as F.

1. F' neither adds a loop not present in F nor removes a loop present in F, therefore $\{lab(l) \mid l \in F_f\} = \{lab(l) \mid l \in F'\}$.

2. The body sets of F' are also a subset lattice. The removal blocks from the body set of l does not violate the subset property between l and $p_F(l)$, but it may violate the subset property between l and one if its children, l_c. By Lemma 8.A.6, if any block of l_c is ejected from l then all blocks of l_c are ejected and l_c becomes immediately nested in $p_F(l)$ by construction of H'. This change also does not violate the subset property of the forest as $body(l_c) \subseteq body(p_F(l))$.

3. The remaining violating loops were not modified and trivially retain this property.

4. The remaining violating loops were not modified and trivially retain this property.

5. No body sets apart from loop l were modified (nor were any paths through the graph), so no additional violation could have been introduced. Loop l no longer has a violation, so the remaining violations must all occur on the ancestor path to $p_{F'}(l)$.

□

Lemma 8.A.8. Let $l' \in F'$ and $l'' \in F_f$ be loops such that $lab(l') = lab(l'') = lab(l)$. Then, if F validates the five assumptions given in Section 8.1, the body and exit sets of l' are correct with respect to function f and F'' validates the same assumptions.

Proof. F'' is constructed from F' through a sequence of *dedicate exit* operations. A series of *repair lcssa* operations are also performed, but do not modify the control flow paths or the loops of its input. To show that these operations maintain these loop properties, we look directly at the *split edge set* operations performed exit dedication. In the following, we refer to block b'' from Figure 7.32 and the set *exits* from Figure 8.12.

For every correct exit e (in X') that has a predecessor in O, the edges between blocks in $pred(e) \cap I$ and e are split. This places a fresh block b'' into the body set of the nearest common ancestor of l and the loop containing e, adds b'' to the exit sets of the loops in which e is an exit, and removes e as an exit from loops in which b'' occurs on every path a block in the loop to e. Loop l'' must have the correct body set as l' has the correct body set by Lemma 8.A.7 and the body set of l' was not modified (only the body set of an ancestor of l' could be). Loop l' has the correct exit set by the same lemma and the set *exit* is the set of exits which are undedicated. The *split edge set* operation replaces precisely these blocks with fresh, dedicated exits in this set. No additional edges leaving the body of this loop are introduced, and therefore l'' must have the correct exit set.

Finally, we show that F'' validates the same five assumptions as F (or, equivalently by Lemma 8.A.7, the same five assumptions as F').

1. The *split edge set* operation does not change the set of loops in the forest, therefore $\{lab(l) \mid l \in F_f\} = \{lab(l) \mid l \in F''\}$.

2. The addition of each new block was added uniformly to one block and all of its ancestors, therefore the resulting forest remains a subset lattice.

3. By Theorem 7.C.7, each additional block added to the function is also added to the correct body set in the input loop nesting forest. Therefore, the body set of each violating loop remains a superset of the body set of the corresponding loop in the function.

4. After the addition of a block, *split edge set* operation repairs exit sets using only predecessor information and the body sets of the claimed loop nesting forest. It follows by Theorem 7.C.7 that the modified exit sets should remain consistent with the modified body sets of the claimed loop nesting forest.

5. For the same reasons as above, any claimed loop in the forest with an accurate body set remains accurate after the operation and, as all loop exit sets are consistent with the body set, the same loops have an accurate exit set. Therefore, no additional violations are introduced and any violating loop in the claimed loop nesting forest remains on one ancestor path.

□

Theorem 8.A.9. If f is in canonical form, F validates the five assumptions given in Section 8.1, and l is a loop with an inaccurate body set, exit set, or parent, then F_{out} is the unique loop nesting forest of $G_{f_{out}}$.

Proof. The loop nesting forest F_{out} is constructed by applying the eject procedure to $l_p(l)$ with F'' as the input loop nesting forest. By Lemma 8.A.8, the forest F'' validates the five assumptions given in Section 8.1. By recursive application of this procedure, the resulting loop nesting forest also validates these assumptions. As each violating loop occurs on the same ancestor path and l is a violating loop, this set of applications operate directly on all possible violations. By Lemma 8.A.7 and Lemma 8.A.8, the loop nesting forest is repaired so that the body and exit sets of the target loop of the procedure is correct. It follows that once the base case is reached, no loop has an inaccurate body set. Additionally, as the output loop nesting forest forms a subset lattice, the nesting properties must also be correct. □

8.B Delete Edge

We first prove that the evaluation of a function after application of the *delete case* transformation (notated below) remains the same. For specific details (including names of intermediate

components), refer to Figure 8.22.

$$f \xrightarrow[a,b]{\text{delete case}} f_{out}$$

Lemma 8.B.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. We prove by filling in the slots of Template 6.A.1.

[Slot → Refinement] We refine the translation of blocks by mapping clones of block a to clones of block a'. We refine the translation of streams by removing the unmatchable block reference from clones of block a.

$$trans(\langle f_t, b_t \rangle) = \begin{cases} a'[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(a) \\ b_t & otherwise \end{cases}$$

$$trans(\langle f_t, b_t, (s_t, T_t, \hat{s}_t) \rangle) = \begin{cases} (s_t, T[\overrightarrow{\sigma_{f_t}}]), trans(\hat{s}_t) & b_t \in \mathcal{C}(a) \\ (s_t, T_t), trans(\hat{s}_t) & otherwise \end{cases}$$

We add the following additional relationship between register contexts.

$$\gamma_1 = \gamma_2$$

[Slot → Invokability] If $b_{t_1} \in \mathcal{C}(a)$, then $b_{t_2} = a'[\overrightarrow{\sigma_{f_{t_1}}}]$. Otherwise, $b_{t_2} = b_{t_1}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[Slot → Asymmetric Evaluation] By the refined *trans* relation, all temporarily diverging steps of evaluation occur at the terminator of a clone of block a.

Simplified Switch In the asymmetric case, the $(n+1)$-th step of evaluation of f transfers control away from block $b_t \in \mathcal{C}(a)$ by application of rule E-SWITCH. By the conditions of the transformation, control is transferred to block b_{t_1}. Here, $dom(param(b_{t_1})) = \langle \overline{r_i} \rangle$. For brevity, we let $T_t = \texttt{switch}\ v\ \overline{cv_i \mapsto ref_i}\ (\texttt{unit} \mapsto ref_b)\ \overline{cv_j \mapsto ref_j}\ ref_d$.

$$\begin{aligned}(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \rightarrow^n\ & (\langle p, f_{t_1}, b_t \rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ T_t, \hat{s}_{t_1}) \\ \rightarrow\ & (\langle p, f_{t_1}, b_{t_1} \rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_1}), \hat{s}_{t_1})\end{aligned}$$

By our strengthened inductive hypothesis, the n'-th step of evaluation of f_{out} stops in function $f_{t_2} = trans(f_{t_1})$ on a terminator that transfers control away from $trans(\langle f_{t_1}, b_t \rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. Then, the $(n'+1)$-th step of evaluation of f_{out} proceeds from application of rule E-SWITCH. As rule E-SWITCH never matches on a `unit` value, the block reference ref_b is never selected during evaluation of function f in the rule above. Additionally, as $\gamma_1(e) = \gamma_2(e)$, the same block reference is selected in evaluation of function f_{out}.

$$\begin{gathered}(p[^f/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \rightarrow^{n'} \\ (\langle p[^f/_{f_{out}}], f_{t_2}, trans(\langle f_{t_1}, b_t \rangle) \rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ T[\overrightarrow{\sigma_{f_{t_2}}}], trans(\hat{s}_{t_1})) \rightarrow \\ (\langle p[^f/_{f_{out}}], f_{t_2}, b_{t_2} \rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_2}), trans(\hat{s}_{t_1}))\end{gathered}$$

First, suppose that $b_{t_1} \in \mathcal{C}(a)$. In this case, $b_{t_1} = b_t$ and $b_{t_2} = trans(\langle f_{t_1}, b_t \rangle)$. Additionally, $stream(b_{t_2}) = stream(a)[\overrightarrow{\sigma_{f_{t_1}}}]$, and $stream(b_{t_2}) = stream(a')[\overrightarrow{\sigma_{f_{t_2}}}]$. These streams differ only by the terminator by construction and are consistent with the strengthened induction. In all other cases, the blocks and streams are identical in both evaluations and both pairs of components are consistent with the strengthened induction. In either case, as register contexts γ_1 and γ_2 are consistent with the strengthened induction, they remain consistent after the parallel addition of registers in the last step of evaluation.

Simplified Block Next, we cover the cases where the n-th step of evaluation of f is earlier within block b (not yet at the terminator). This case is trivial by application of Lemma 6.C.1.

[Slot ← Refinement] We refine the translation of blocks by mapping clones of block a' to clones of block a. We refine the translation of streams by replacing the terminator of clones of block a' with the original terminator of block a – this simply adds extra switch cases which is unmatchable during any execution. For brevity, we decompose the stream s_t into $(s'_t, \hat{s}_t)$.

$$trans(\langle f_t, b_t \rangle) = \begin{cases} a[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(a') \\ b_t & otherwise \end{cases}$$

$$trans(\langle f_t, b_t, (s_t, \hat{s}_t) \rangle) = \begin{cases} (s'_t, term(a)[\overrightarrow{\sigma_{f_t}}]), trans(\hat{s}_t) & b_t \in \mathcal{C}(a) \\ s_t, trans(\hat{s}_t) & otherwise \end{cases}$$

We add the following additional relationship between register contexts.

$$\gamma_1 = \gamma_2$$

[Slot ← Invokability] If $b_{t_2} \in \mathcal{C}(a')$, then $b_{t_1} = a[\overrightarrow{\sigma_{f_{t_2}}}]$. Otherwise $b_{t_1} = b_{t_2}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[Slot ← Asymmetric Evaluation] By the refined $trans^{-1}$ relation, all temporarily diverging steps of evaluation occur at the terminator of a clone of block a'.

Simplified Switch In the asymmetric case, the $(n' + 1)$-th step of evaluation of f_{out} transfers control away from block $b_t \in \mathcal{C}(a')$ by application of rule E-Switch. For brevity,

we let T_t have the following form

$$\texttt{switch}\ v\ \overline{cv_i \mapsto ref_i}\ (\texttt{unit} \mapsto ref_b)\ \overline{cv_j \mapsto ref_j}\ ref_d$$

Here, $dom(param(b_{t_2})) = \langle \overline{r_i} \rangle$.

$$(p[^f/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \rightarrow^{n'}$$

$$(\langle p[^f/_{f_{out}}], f_{t_2}, b_t \rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ T[\overrightarrow{\sigma_{f_{t_2}}}], \hat{s}_{t_2}) \rightarrow$$

$$(\langle p[^f/_{f_{out}}], f_{t_2}, b_{t_2} \rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_2}), \hat{s}_{t_2})$$

By our strengthened inductive hypothesis, the n'-th step of evaluation of f_{out} stops in function $f_{t_1} = trans^{-1}(f_{t_2})$ on a terminator that transfers control away from $trans^{-1}(\langle f_{t_1}, b_t \rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. Then, the $(n'+1)$-th step of evaluation of f_{out} proceeds from application of rule E-Switch. As rule E-Switch never matches on a `unit` value, the block reference ref_b is never selected during evaluation of function f in the rule above. Additionally, as $\gamma_1(e) = \gamma_2(e)$, the same block reference is selected in evaluation of function f_{out}.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \rightarrow^{n}$$

$$(\langle p, f_{t_1}, trans^{-1}(\langle f_{t_2}, b_t \rangle) \rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ T_t, trans^{-1}(\hat{s}_{t_2})) \rightarrow$$

$$(\langle p, f_{t_1}, b_{t_1} \rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_1}), trans^{-1}(\hat{s}_{t_2}))$$

First, suppose that $b_{t_2} \in \mathcal{C}(a')$. In this case, $b_{t_1} = b_t$ and $b_{t_2} = trans^{-1}(\langle f_{t_2}, b_t \rangle)$. Additionally, $stream(b_{t_2}) = stream(a')[\overrightarrow{\sigma_{f_{t_2}}}]$, and $stream(b_{t_1}) = stream(a)[\overrightarrow{\sigma_{f_{t_1}}}]$. These streams differ only by the terminator by construction and are consistent with the strengthened induction. In all other cases, the blocks and streams are identical in both evaluations and both pairs of components are consistent with the strengthened induction. In either case, as register contexts γ_1 and γ_2 are consistent with the strengthened induction, they remain consistent after the parallel addition of registers in the last step of evaluation.

Simplified Block Next, we cover the cases where the n'-th step of evaluation of f_{out} is earlier within block a' (not yet at the terminator). This case is trivial by application of Lemma 6.C.1. □

In the remainder of this section, we refer the *delete edge* transformation (notated below) applied when paths only change in multiplicity. For specific details (including names of intermediate components), refer to Figure 8.22 and Figure 8.23.

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[(a,b)]{\text{delete edge}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Corollary 8.B.2. The set of paths over block labels from ℓ_0 to $\ell_{b'}$ in $G_{f_{out}}$ for some block $b' \in B \setminus R$ that do not traverse the edge $(lab(a), lab(b))$ where $lab(b') = \ell_{b'}$ are equivalent to the set of such paths in G_f.

Before we can state the maintenance properties of the *delete edge* procedure, we need to define several lemmas to show that the approximate loop nesting forest constructed directly before the *eject* procedure is in the form assumed by ejection. First, we show that the approximate loop nesting forest contains the correct number of loops and each loop has the correct header.

Lemma 8.B.3. If f is in canonical form, then the following three sets are equivalent:

$$dom(H') = \{lab(l) \mid l \in F \setminus H_K\} = \{lab(l') \mid l' \in F_{f'}\}$$

Proof. As a backedge that does not exist in G_f cannot be introduced to $G_{f'}$ by the deletion of an edge, $\{lab(l') \mid l' \in F_{f'}\} \subseteq \{lab(l) \mid l \in F\}$. Let $l \in F$ be a loop of G_f and let $S \subseteq G_f\langle body_f(l)\rangle$ describe the subgraph induced by the set of blocks reachable from $entry(f)$ after removing the edge (a, b). S is strongly connected if it contains the backedge of l. This is the case when the latch of l is still reachable and (a, b) is not the backedge of l, and is also precisely the condition that satisfies $l \notin H_K$. □

Now we show that the approximate loop nesting forest forms a subset lattice (the nesting structure of loops is somewhat correct – an inner loop with too many blocks in its body set may be nested under a descendant of its true parent).

Lemma 8.B.4. The loop nesting forest reconstructed from (H', L', X') forms a subset lattice.

Proof. The set L' is constructed from L_F by removing blocks from loops such that blocks are also removed from all descendant loops. As F_f is a subset lattice by definition, the loop nesting forest reconstructed from (H', L', X') must also form a subset lattice. □

Now we show that every loop in the approximate loop nesting forest has a body set containing a superset of the correct blocks. This is stated formally in Lemma 8.B.5. This property ensures that blocks must only be *ejected* from the loop to correct the body set, and no additional blocks must be brought in from the outside.

Lemma 8.B.5. If f is in canonical form, then $body_{f'}(l') \subseteq S$ for each $l' \in F'$ where S is the set of blocks composing the body of the loop with header $header_{f'}(l')$ reconstructed from H' and L'.

Proof. Let $l \in F$ be the loop with header label ℓ_h. $l \notin H_K$ by Lemma 8.B.3. Then, $A(l) = l$, $A(p_F(l'')) \in F$ is a descendant of l in F if l'' is a proper descendant of l in F, and the set of blocks S is equivalent to the following.

$$\begin{aligned}
S &= \{block_f(\ell_b) \mid (\ell_b, \ell_l) \in L' \wedge \ell_l \in N_{H'}(\ell_h)\} \\
&= \{block_f(\ell_b) \mid (\ell_b, \ell_l) \in L_F \wedge block_f(\ell_b) \notin R \wedge A(\ell_l) \in N_{H'}(lab(l))\} \\
&= \{block_f(\ell_b) \mid (\ell_b, \ell_l) \in L_F \wedge block_f(\ell_b) \notin R \wedge A(\ell_l) \in F\langle l \rangle \setminus H_K\} \\
&= body_f(l) \setminus R
\end{aligned}$$

By Corollary 8.B.2, $body_{f'}(l') \subseteq body_f(l)$ and $body_{f'}(l') \subseteq body_f(l) \setminus R$ as f' contains no blocks in R. □

Now we show that the exits of each loop are correct with respect to the claimed body set. Naturally, this implies the exit sets for loops with an accurate body set are correct.

Lemma 8.B.6. If f is in canonical form, then the exit set of a loop l' is consistent with its body set for every loop l' in the loop nesting forest reconstructed from (H', L', X').

Proof. Let $lab(b') = \ell_{b'}$, and let $L_{l'}$ and $X_{l'}$ be the approximate body and exit sets of l', respectively. First, suppose $pred_{f'}(b') \cap L_{l'} = \emptyset$ but $(\ell_{b'}, lab(header(l'))) \in X'$. By construction of X', this pair must also exist in X_F and therefore all edges from loop l' to block b' must have become unreachable through the deletion an edge. However, as f is in canonical form, b' is a dedicated exit and must have become unreachable by the deletion of the edge. Then, $b' \in R$ and this pair cannot exist in X' by construction. Now, suppose $pred_{f'}(b') \cap L_{l'} \neq \emptyset$ but $(\ell_{b'}, lab(header(l'))) \notin X'$. The absence of this pair implies that b' was not an exit of the loop with the same header in function f and no such exit path can be constructed by the deletion of an edge by Corollary 8.B.2. □

Now we show that all inaccurate loops can be found on a single path in the claimed loop nesting forest.

Lemma 8.B.7. If f is in canonical form, then all loops with an inaccurate body set exist on the same ancestor path in the loop nesting forest reconstructed from (H', L', X').

Proof. Suppose l and l' are disjoint violating loops in the claimed loop nesting forest after the deletion of the edge (a, b). Without loss of generality, let $a \in body(l)$. Let b' be a block previously in the body of l' that no longer belongs to l' after the deletion of the edge. Then, either b' is no longer reachable from the header of l' (within the body set of l'), or the latch of

l' is no longer reachable from b' (within the body set of l'). As the edge (a, b) does not occur within l', it must be the case that b' has become unreachable. However, as l' is a reducible loop and is not an ancestor of l, all blocks of l' must have become unreachable and l' is not part of the claimed loop nesting forest after deletion of the edge. □

Finally, we can state the maintenance theorems for the *delete edge* procedure.

Theorem 8.B.8. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. Let $b' \in R$ be a block removed by the *post-delete edge* procedure. For no evaluation of function f can evaluation enter block b' as every path must necessarily take the block reference removed by the *delete case* procedure, but the switch value for this reference is `unit` and can thus never be selected. It follows that removal of the block set R from the function f then does not influence evaluation and the remainder of the proof is then trivial by application of Theorem 8.B.1 and Theorem 8.A.1. □

Corollary 8.B.9. In the case where $(a, b) \in 1E(f)$, the set of paths over block labels from ℓ_0 to $\ell_{b'}$ in $G_{f_{out}}$ for some block $b' \in B$ where $lab(b') = \ell_{b'}$ is identical to the set of such paths in G_f.

Theorem 8.B.10. If f is in canonical form, then f_{out} is in canonical form.

Proof. If $(a, b) \in^k E(f)$ where $k > 1$, then this theorem is trivial by Corollary 8.B.9. Otherwise, this theorem is trivial by Theorem 8.A.2. □

Theorem 8.B.11. If f is in LCSSA form, then f_{out} is in LCSSA form.

Proof. First, we cover the case where $(a, b) \in^k E(f)$ where $k > 1$. Let $b \in B$ and $b' \in B'$ be blocks that share the same label. No registers are defined in f' that are not also defined in f and the uses which occur in b' are a subset of the uses which occur b. In the other case, this theorem is trivial by Theorem 8.A.3. □

Theorem 8.B.12. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f_{out}$ is well-typed.

Proof. First, we cover the case where $(a, b) \in^k E(f)$ where $k > 1$. In this case, block a''s terminator is well-typed by rule T-Switch as each block reference of a is well-typed and $inst(a') = inst(a)$ (thus the same registers are defined at the time the terminator is typed). The set $implicit(a')$ is constructed by removing references to registers defined only in ref_b. Then, the implicits of block a' are well-typed and $nonlocal_{f'}(a') \subseteq nonlocal_f(a)$. Then, according to Lemma 3.A.5, block a' is well-typed by rule T-Block. In the other case, this theorem is trivial by additional application of Theorem 8.A.4. □

Theorem 8.B.13. The unique dominator tree of $G_{f_{out}}$ is D_{out}.

Proof. If $(a, b) \in^k E(f)$ where $k > 1$, then this theorem is trivial by Corollary 8.B.9. Otherwise, D' is the unique dominator tree of f' by Ramalingam and Reps [53], and the rest follows form Theorem 8.A.5. □

Theorem 8.B.14. F_{out} reconstructed from $(H_{out}, L_{out}, X_{out})$ is the unique loop nesting forest of $G_{f_{out}}$.

Proof. If $(a, b) \in^k E(f)$ where $k > 1$, then this theorem is trivial by Corollary 8.B.9. Otherwise, the loop nesting forest reconstructed from (H', L', X') validates the five assumptions given in Section 8.1 by Lemma 8.B.3 through Lemma 8.B.7. Then, the proof is concluded by application of Theorem 8.A.9. □

9 Optimizations

In this chapter we outline proper control flow graph transformations that are the result of classic compiler optimization techniques. These optimizations include straightening (removing unconditional branches), jump simplifications (reducing block branch factor or reducing the number of branches on a particular path), function inlining, and loop optimizations (unswitching, unrolling, and peeling).

9.1 Straightening

A program may contain a control flow edge between two blocks a and b such that the only successor of a is b and the only predecessor of b is a. Such an edge may occur at synthesis, but is more likely to occur due to a result of another transformation (e.g. canonicalization as discussed in Chapter 7 or inlining a subset of a control flow graph at a program point as discussed in Section 9.4). These edges can be collapsed through a process called *straightening* so that a and b are combined into a single block. This reduces branching logic and places unconditionally subsequent code physically near its caller, improving instruction cache locality.

An example of this transformation is illustrated in Figure 9.11. Here, block p's sole

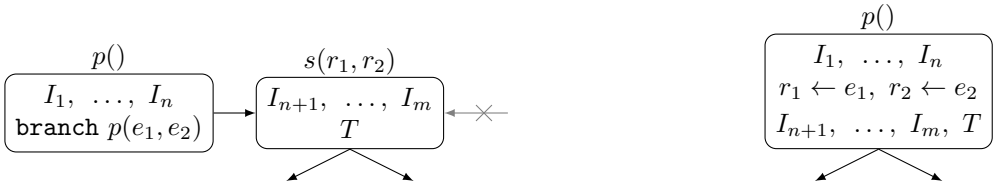


Figure 9.11: Straightening the edge (p, s) combines the instructions from both blocks into one. Block parameters of block s and the arguments from block p are replaced by move instructions.

$$term(a) = \texttt{branch}\ ref$$

$$ref = \ell_b(\overline{e_i}) \qquad lab(b) = \ell_b \qquad pred_f(b) = \{a\} \qquad loop_f(a) = loop_f(b)$$

$$imp = implicit(a) \cup (implicit(b) \ominus def(a)) \qquad s' = \langle r_i \leftarrow \texttt{move}(e_i) \mid (r_i \colon t_i) \in param(b) \rangle$$

$$\ell_a = lab(a) \qquad a' = (\ell_a, param(a), imp, inst(a) \cup s' \cup stream(b)) \qquad f = (\ell, t, B, \ell_0)$$

$$B' = B \setminus \{a, b\} \cup \{a'\} \qquad f' = (\ell, t, B', \ell_0) \qquad D_{out} = D_f[a' \mapsto a \mid p_D(a') = b][b \mapsto \varnothing]$$

$$H_{out} = H_{F_f} \qquad L_{out} = \{(\ell_1, \ell_2) \in L_{F_f} \mid \ell_1 \neq lab(b)\} \qquad X_{out} = X_{F_f}$$

$$\frac{}{(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[a,b]{\text{straighten}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})}$$

Figure 9.12: Collapsing the blocks of a non-critical edge.

successor is block s and block s's sole predecessor is block p. The edge can be collapsed, combining the instruction streams from both blocks into a single block (retaining the label of block p).

This transformation is formally described by Figure 9.12. The preconditions for this transformation require that a single edge exists between blocks a and b and that the edge (a, b) does not cross a loop boundary. The first condition prevents a second target of a from being spuriously removed and a second edge terminating at b from losing its target. The second condition prevents de-canonicalization, as such edges are often a result of adding a preheader or a dedicated exit. These edges are prime candidates for this transformation, but must only be straightened once preservation of canonical form is no longer a concern (as a last step during the optimization phase of compilation directly before synthesis to the target platform).

The bulk of the transformation is simple. A new block a' is created with the same label, parameters, and initial instruction stream as a. Sharing the same block label ensures that block references targeting a now transparently target a' in the new function. We replace the terminator of a with the instruction stream of b. Between these two streams, we simulate

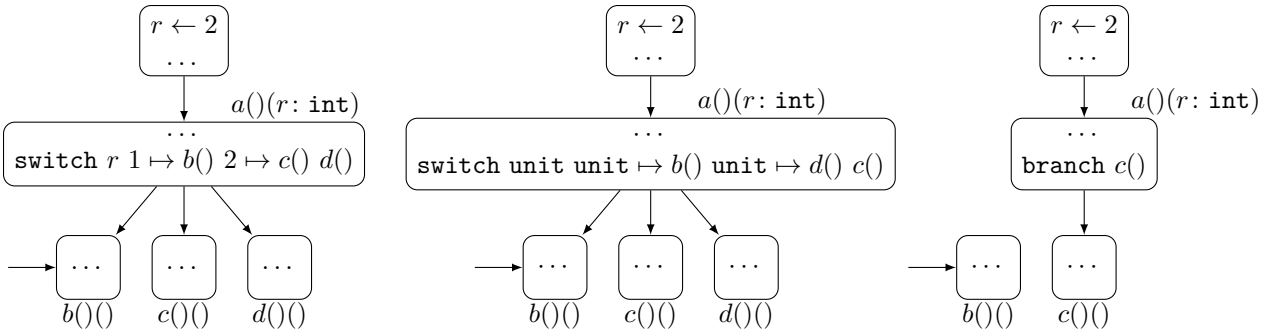


Figure 9.21: Simplifying a switch condition in two stages.

the passing of control flow from a to b by explicitly moving the expressions in the terminator of a the parameters of b. We also take care to add the implicit parameters of b (that are not defined in a) to the new block.

The preconditions of this transformation ensure that $idom(b) = a$ and we can maintain dominators by simply collapsing this symmetric edge in the dominator tree. All children of b are linked directly to a and b is unlinked from the tree. The loop bodies are maintained by simply removing references to block b.

9.2 If Simplification

A program may contain a switch terminator containing references to multiple blocks (or to the same block with differing arguments to block parameters) that is statically known to select only one specific reference regardless of program input. This can happen due to constant definitions and feature flags at the source level, or may come about due to other transformations, such as inlining (discussed in Section 9.4) or loop unswitching (discussed in Section 9.5). These terminators can be replaced with an unconditional branch containing the same block reference. This transformation may in turn create additional optimization opportunities. For example, the deletion of an edge may reduce the size or complexity of a loop or function past the threshold that allows loop unrolling or function inlining.

$$term(b) = \texttt{switch}\ v\ \overline{cv_i \mapsto ref_i}\ ref_d$$

$$v = cv \text{ over all execution traces} \qquad (cv; \overline{cv_i \mapsto ref_i}\ ref_d) \to ref$$

$$T = \texttt{switch unit}\ \langle \texttt{unit} \mapsto ref_i \mid ref_i \in term(b) \wedge ref_i \neq ref \rangle\ ref$$

$$imp = \{(r : t) \in implicit(b) \mid use_a(r) \neq \{term(b)\}\}$$

$$b' = \langle lab(b), param(b), imp, inst(b) \cup T \rangle$$

$$f = (\ell, t, B, \ell_0) \qquad B' = B \setminus \{b\} \cup \{b'\} \qquad f' = (\ell, t, B', \ell_0)$$

$$ref = \ell_t(\overline{e_i}) \qquad t = block_f(\ell_t) \qquad targets = \{block_f(\ell)_{\langle k \rangle} \mid \ell \in^k target(b)\} - \{t\}$$

$$(f', D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[\{(b,t') \mid t' \in targets\}]{\text{delete edge}}{}^{*} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

$$\overline{(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[\langle f,b \rangle]{\text{if simplify}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})}$$

Figure 9.22: Simplifying a switch terminator with only one possible target.

An example of this transformation is illustrated in Figure 9.21. This transformation works in two parts. First, the terminator is altered so that the block reference that is known to be used becomes the default and the remaining block references switch on the `unit` value. Next, the *delete edge* operation is performed iteratively over the terminator until only the live block reference remains.

This optimization is formally described by Figure 9.22. The only non-trivial precondition for this transformation is that the switch value v must always evaluate to cv over all evaluations of f regardless of input and nondeterministic state. This allows us to statically select the block reference taken during any execution from the terminator of b, called *ref*. The terminator of block b can then be 'rotated' so that *ref* becomes the default case, and all other references are selected with the value `unit` rather than their original constant value. This puts the terminator in the correct form required by the *delete edge* operation, which does the remainder of the heavy lifting. Additionally, as the switch value of the terminator (which may have been a register) is replaced by the value `unit`, an implicit parameter must

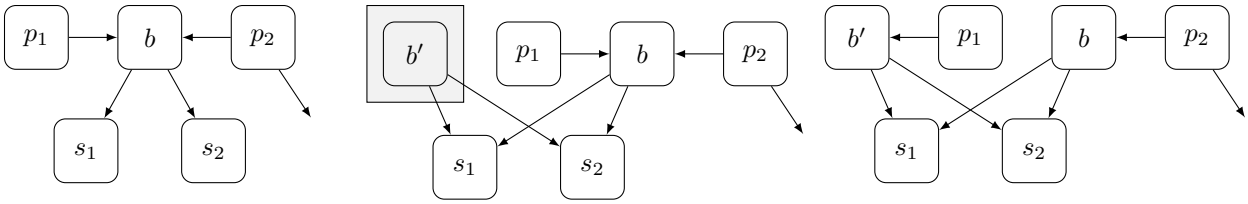


Figure 9.31: Simplifying a jump to an empty block in two parts. First, the empty block b is duplicated. Then, the path (p_1, b) is replaced with (p_1, b') so that both edges terminating at b and b' are eligible for straightening.

be removed from the containing block if it is not otherwise referenced. All edges in the multiset *targets*, composed of the outgoing edges of b *except* for the edge described by *ref*, are removed in sequence.

9.3 Jump Simplification

In this section we detail a transformation that simplifies the jump of a critical edge (in which case straightening cannot be applied). This transformation can be applied in two circumstances. The first case is applied in relation to an instruction-less block with multiple predecessors and multiple successors. Such a block can occur naturally from a source translation (the join point between consecutive if statements), but may also occur from previous transformations (for example, a split block is no longer necessary if the loop boundary is moved). The second case, referred to as *jump threading*, is applied when the final target of a series of jumps is not known statically for all paths, but is known statically for a particular path. In either case, we can create an immediate opportunity for straightening by duplicating a single block on the problematic path and routing control flow from a subset of predecessors to the new block. This allows the original path and the new path to be optimized independently.

An example of the transformation applied in the first case is illustrated in Figure 9.31. In this example, all edges terminating at block b are critical and cannot be collapsed without

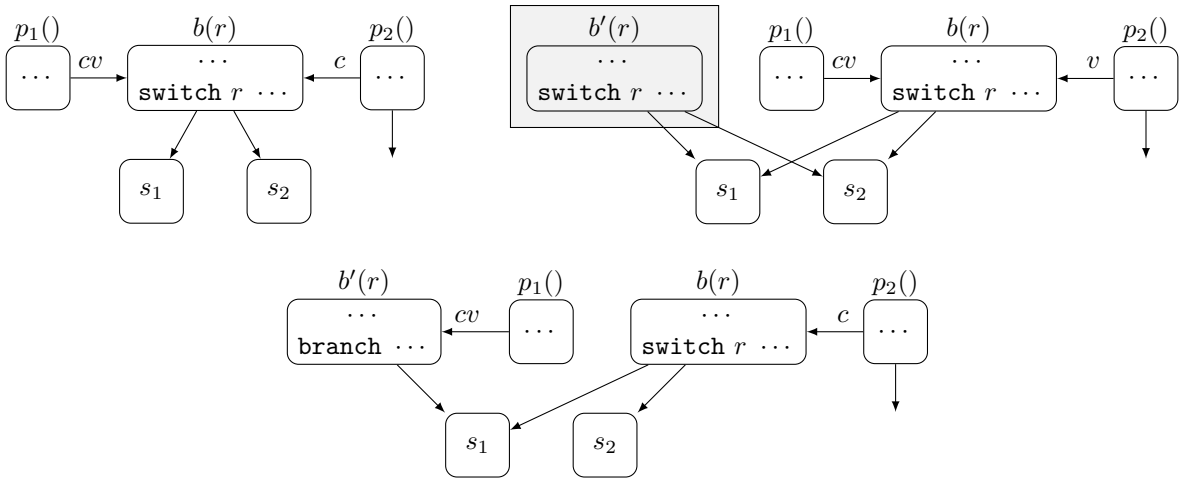


Figure 9.32: Duplicating a single control flow path where switching values are known statically. This transformation creates additional optimization opportunities as analysis of block b' is influenced by only one control flow path.

destroying an unrelated edge. To simplify the jumps we add block b', a clone of block b, to the graph and replace references to block b in block p_1 with a reference to block b'. The only changes to both the dominator tree and the loop nesting forest are local, This creates an immediate opportunity for the edge (p_1, b') to be straightened so that the number of branch instructions on paths $p_1 \rightsquigarrow s_1$ and $p_1 \rightsquigarrow s_2$ are reduced by one. This may also create additional opportunities for straightening when b has a single remaining predecessor.

An example of this transformation applied in the second case is illustrated in Figure 9.32. In this example, the target in block b is known statically to be s_1 if the edge (p_1, b) was taken as register r holds the constant value cv. However, if the edge (p_2, b) was taken the target is not known. This prevents the terminator of b from being simplified in general when considering all paths. To enable simplification, we add b', a clone of block b, to the graph and replace references to block b in block p_1 with a reference to block b'. This makes the block b' unique to the path $p_1 \rightsquigarrow s_1$ and the terminator of b' can be replaced by an unconditional branch. This allows the edge (p_1, b') to be straightened, reducing the number branch instructions on this path by one.

$$\frac{\begin{array}{c} term(p) = \texttt{branch}\ \ell_b(\overline{e_i}) \qquad |pred_f(b)| > 1 \qquad |succ_f(b)| > 1 \qquad b \neq header(loop(b)) \\ b \neq latch(loop(b)) \qquad f = (\ell, t, B, \ell_0) \qquad b' = (\ell_{b'}, param, implicit(b), stream(b)) \\ p' = p[^{lab(b)}/_{\ell_{b'}}] \qquad \ell_{b'} \text{ is fresh} \qquad B' = B \setminus \{p\} \cup \{p', b'\} \qquad f' = (\ell, t, B', \ell_0) \\ D' = D_f[b' \mapsto p][b \mapsto nca_{D_f}(b)] \qquad D_{out} = D'[c \mapsto nca_{D'}(pred(c)) \mid c \in child_{D_f}(b)] \\ H_{out} = H_{F_f} \qquad L_{out} = L_{F_f}[b' \mapsto loop(b)] \qquad X_{out} = X_{F_f} \cup \{(\ell_{b'}, \ell) \mid (lab(b), \ell) \in X_{F_f})\} \\ f' \xrightarrow[\{(r_i : typeof(d_i)) | d_i \in def(b) \text{ where } d_i \text{ defines } r_i\}]{\text{rewrite uses}}{}^{*} (f_{out}, D_{out}) \end{array}}{(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[p,b]{\text{jump simplify}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})}$$

Figure 9.33: Simplifying a sequence of jumps.

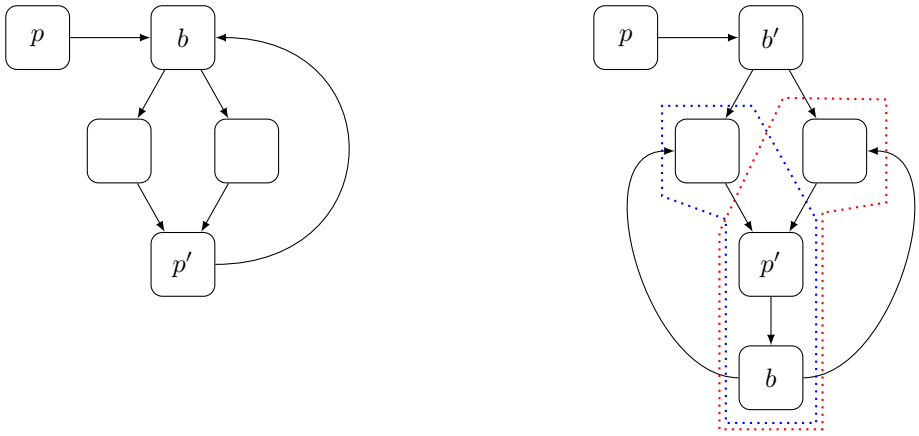


Figure 9.34: Creation of an irreducible region when simplifying the jump to a loop header.

This transformation is formally described by Figure 9.33. The transformation is applied to a block b and one of its predecessors p. After the transformation, the edge leaving p will be eligible for straightening. This transformation is not applicable when b is the latch of a loop, as this would introduce a latch and backedge pair. This transformation is also not applicable when b is the header of a loop, as this may introduce an irreducible region as illustrated in Figure 9.34. This transformation is also not applicable when b is the header or a preheader, as they have a unique predecessor and successor, respectively.

The first case described above is applicable (for benefit) when $inst(b) = \langle\rangle$, and the second

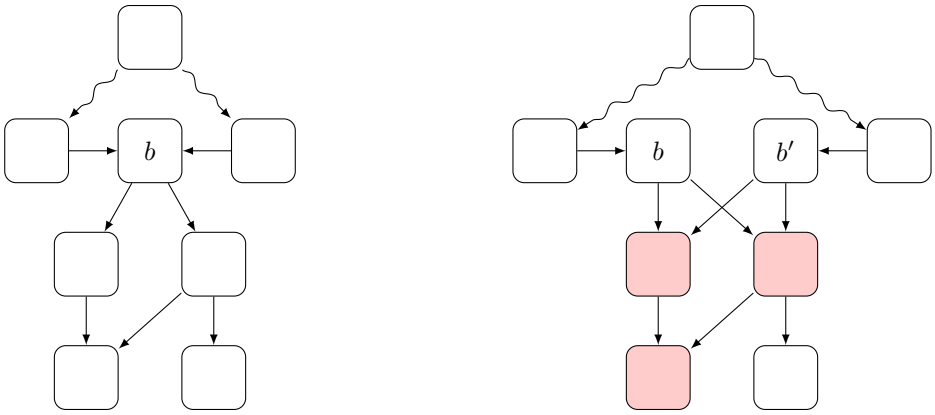


Figure 9.35: Domination change after jump simplification. An alternate path to the shaded blocks (which were previously immediately dominated by b) has been introduced.

case is applicable when $term(b) = \texttt{switch}\ r\ \overline{cv_i \mapsto ref_i}\ ref_d$ and $r = cv$ on path (p, b). The transformation proceeds the same in both cases.

First, a fresh block b' is created with a fresh label and its parameters and instruction stream mapped with fresh registers (this same technique is applied to all blocks during loop duplication as discussed in Section 8.3). Then, the edge (p, b) is replaced with the edge (p, b') in the resulting graph. If block b defines a register, either in a parameter or as the target of an instruction, then downstream uses of that register must be rewritten to be the reaching definition of the register defined in block b, or the symmetric register defined in block b'. To rewrite these uses, we can use the SSA reconstruction algorithm described by Braun et al. [12] in Section 7.1. The dominator tree is repaired by inserting the new block b' as a child of its sole predecessor p, and recalculating the immediate dominator of the blocks that used to be immediately dominated by b. An example set of affected blocks is shown in Figure 9.35. Loop bodies are maintained by adding block b' to the same loop as block b. Loop exits are maintained by adding block b' to all exit sets to which block b belongs. An additional illustration of jump simplification near a loop boundary is given in Figure 9.36.

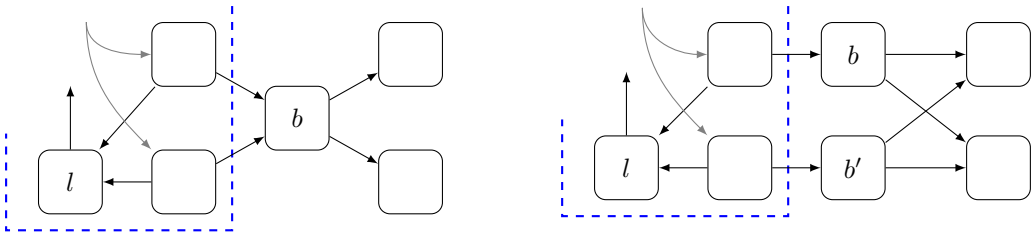


Figure 9.36: Addition of a loop exit after jump simplification.

9.4 Function Inlining

Function inlining replaces a function call site with the body of the called function. This requires replacing the call instruction with an unconditional branch to the entry of the inlined function and replacing all return instructions of the inlined function with unconditional branches back to the instruction immediately following the call. Generally, function inlining increases speed with a small increase in code size, but overly aggressive inlining may increase the pressure of the instruction cache and reduce overall performance.

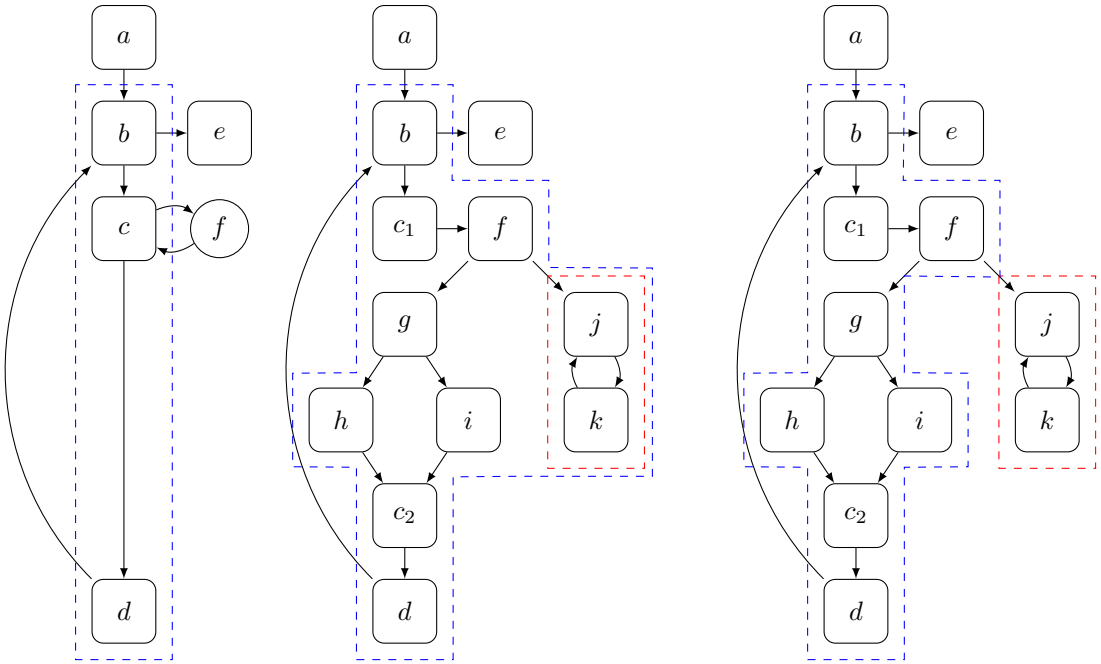
A major benefit of function inlining comes from the additional optimization opportunities it provides. For example, suppose function $f(a, b, cv)$ is called inside a loop such that a is defined in the loop, b is loop-invariant, and cv is a constant value. Further suppose that the value of a is used very broadly as a switch. For example, the result of the function is determined largely by $a \leq 100$. If interval analysis shows that this holds for all values of a around the call-site, then the body of the inlined function can be simplified due to more accurate value propagation, as the result of the function call is now known to be loop-invariant, the remaining inlined body can be hoisted to the loop's preheader.

This transformation is implemented in Waddle in two small steps. First, the block containing the call instruction is 'split' into blocks b_1 and b_2 so that the call instruction is at the end of one basic block and the subsequent instruction is at the beginning of another. Then, a clone of the target function is inserted between split blocks – the call instruction in b_1 is replaced by a branch to the clone's entry block, and each exit of the cloned function

branches back to block b_2. Repairs to the dominator tree and loop nesting forest can be performed, for the most part, in a local manner. Domination properties inside of the inlined function do not change - the only changes occur at the boundaries of the inlined call. The entry of the inlined function is dominated by the block containing the call site, block b_1, and the block containing the subsequent instruction, block b_2, is dominated by the lowest common ancestor of the inlined function's exits. Generally, the structure of the loop nesting forest also does not change – the entire loop nesting forest of the inlined function can be embedded as a child of the loop containing the call instruction. An exception to this method occurs only when the inlined function contains a loop with no exits, in which case no blocks of that loop can reach a return statement, and therefore can reach neither b_2 nor the latch of the loop containing the call instruction. Fortunately, this can be easily repaired by re-purposing the block ejection procedure discussed in Section 8.1.

An example of this transformation is illustrated in Figure 9.41. Block c, containing the callsite, is split and the function body is made reachable by the process described above.

This transformation is formally described by Figure 9.42. First, an isomorphic clone, denoted f', of the call target $\ddot{f}$ is created. Then, block b containing the call being inlined is split into two blocks, b_1 and b_2. The former block contains the original parameters as well as the instruction stream *up to* the call instruction. The latter block contains the instruction stream *following* the call instruction. This block also has a single, newly constructed block parameter, the register and type of which matches the target of the call instruction. The blocks of the cloned function are mapped so that all return instruction instead branch to block b_2, passing the return value as a block argument. The dominator tree is constructed by merging the dominator trees of both functions and linking $entry(f')$ and b_2 to their immediate dominators. The modification of the loop nesting forest is similar: the loop nesting structure is constructed by merging the two nesting sets together and making the



(a) Pre-inlining. (b) Post-inlining. (c) Post-ejection.

Figure 9.41: Function inlining in two parts. First, the source block is split around the callsite and a duplicated function body is inserted between these block halves. Then, blocks that cannot reach an exit of the inlined function are ejected from the loop containing the call instruction.

roots of the cloned loop nesting forest a child of the loop containing the call instruction, and the loop bodies are constructed by merging the two body relations and adding all cloned blocks not contained in a loop of the inlined function to the loop containing the call instruction. Lastly, the fake edge (b_1, b_2) is removed such that if the inlined function contains no return instructions, block b_2 and its dominating subtree will be found to be unreachable and removed from the graph. This has the beautiful side-effect of also performing ejection on the loop so that non-exiting loops will be moved out into the proper level of the loop nesting forest.

$$f = (\ell, t, B, \ell_0) \qquad l = \mathit{loop}_f(b) \qquad \mathit{stream}(b) = \langle I_1, \ldots, I_n, T \rangle$$
$$I_k = r \leftarrow \texttt{call}(r_c, \overline{e_i}) \qquad \mathit{def}_f(r_c) = \texttt{addr}(\ddot{f}) \qquad f' \in \mathcal{C}(\ddot{f}) \qquad f' = (\ell', t', B', \ell'_0)$$
$$b' = \mathit{entry}(f') \qquad \mathit{lab}(b') = \ell_{b'} \qquad \mathit{imp}_1 = \{(r_i \colon t_i) \in \mathit{implicit}(b) \mid r_i \in \mathit{use}(b_1)\}$$
$$\mathit{imp}_2 = \{(r_i \colon t_i) \in \mathit{implicit}(b) \mid r_i \in \mathit{use}(b_2)\} \qquad s_1 = \langle I_1, \ldots, I_{k-1}, \texttt{branch}\ \ell_{b'}(\overline{e_i}) \rangle$$
$$s_2 = \langle I_{k+1}, \ldots, I_n, T \rangle \qquad b_1 = (\ell_b, \mathit{param}(b), \mathit{imp}_1, s_1) \qquad b_2 = (\ell_{b_2}, \langle r \colon t' \rangle, \mathit{imp}_2, s_2)$$
$$\ell_{b_2} \text{ is fresh} \qquad B'' = B \setminus \{b\} \cup \{b_1, b_2\} \cup B'[{}^{\texttt{return}\ e_i}/_{\texttt{branch}\ \ell_{b_2}(e_i)}] \qquad f'' = (\ell, t, B'', \ell_0)$$
$$D' = (D_f \cup D_{f'})[b' \mapsto b_1][b_2 \mapsto \mathit{nca}_{D_{f'}}(\mathit{exit}(f'))][c \mapsto b_2 \mid c \in \mathit{child}_{D_f}(b)]$$
$$H' = (H_{F_f} \cup H_{F'_f})[l' \mapsto l \mid l' \in \mathit{loop}(f')]$$
$$L' = (L_{F_f} \cup L_{F'_f})[b'' \mapsto l \mid b'' \in B' \wedge \mathit{loop}_{f'}(b'') = l_\varnothing]$$
$$X' = X_{F_f} \cup X_{F'_f} \qquad (f'', D', H', L', X') \xrightarrow[b_1, b_2]{\text{post delete edge}} (f_{out}, D_{out}, H, {}_{out}\, L_{out}, X_{out})$$
$$\overline{(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[\langle f, b, r \leftarrow \texttt{call}(r_c, \overline{e_i}) \rangle]{\text{inline}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})}$$

Figure 9.42: Inserting the body of an external function at a callsite.

9.5 Loop Unswitching

A program may contain a conditional branch in a loop such that the branching value is non-constant (so that it cannot be trivially optimized via if-simplification), but invariant with respect to the containing loop. This creates a situation where once a loop is entered during evaluation, every iteration of the loop will traverse the same edge of the terminator. Further path optimizations can be enabled by *unswitching* the loop. First, the calculation of the branching value is hoisted into the loop's preheader if it is not already defined outside of the loop. Next, the entire loop body is cloned once for each target of the conditional branch. Then, the preheader terminator is modified so that it mirrors the semantics of the conditional branch and references each cloned loop header. Now, the each cloned instance of the original conditional branch can be independently simplified via if-simplification – the comparison performed in the preheader statically ensures that each loop will unconditionally

```
<T> boolean contains(T[] arr, T v) {
    for (T e : arr) {
        if (v != null && v.equals(e)) {
            return true;
        }
    }

    return false;
}
```

```
<T> boolean contains(T[] arr, T v) {
    if (v != null) {
        for (T e : arr) {
            if (v.equals(e)) {
                return true;
            }
        }
    } else {
        for (T e : arr) {
            /* empty body */
        }
    }

    return false;
}
```

Figure 9.51: A Java source-level example of loop unswitching.

take the symmetric edge in the loop body.

Figure 9.51 gives a source-level illustration of this transformation. In this example, the null-check of the variable v can be factored out so that one instance of the loop (where v is non-null) can omit the null check on each iteration, and one instance of the loop (where v is null) can be simplified to a no-op body (and, subsequently, removed as it is free of side-effects[1]).

Figure 9.52 illustrates this transformation using an abstract control flow graph. In this example, the register r is defined prior to the entry of the loop. This means that once control flow reaches block h, the condition $r = cv$ will remain unchanged until control flow reaches block e. The loop body is cloned as discussed in Section 8.3. This cloned subgraph is made reachable via the original preheader, whose terminator is modified from an unconditional branch to the original header to a conditional branch to either the original header or its clone. Notice that this undedicates the preheader and all exits of the loop that must be subsequently re-dedicated. After this transformation, if control flow reaches block h, it must be the case that $r = cv$, thus the edge to s_1 is always taken (and the edge to s_2 can be deleted). Similarly, if control flow reaches block h', it must be the case that $r \; not = cv$ and

[1]Except for when the array is `null`, in which case a null-pointer exception is thrown at the time of iteration. We ignore this to keep the example clear.

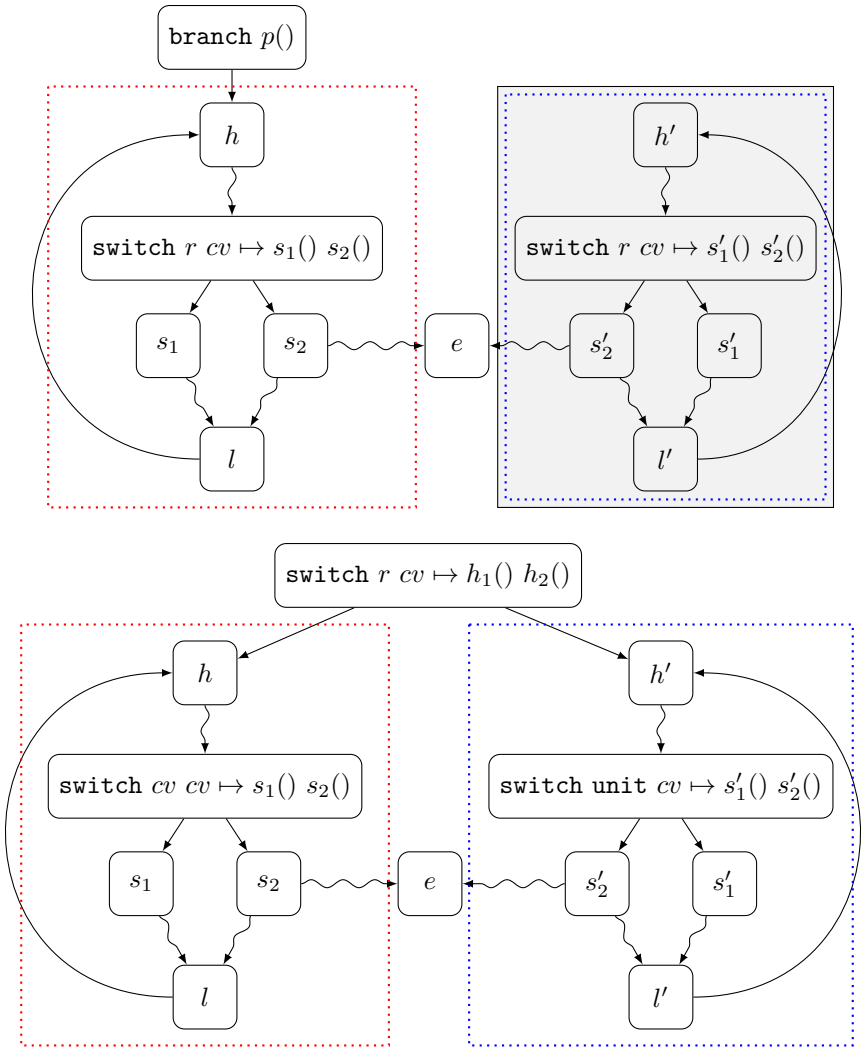


Figure 9.52: Unswitching a loop by duplicating the body and hoisting the loop-invariant condition to the preheader. Subsequently, the condition on each copy of the loop can be simplified independently.

the edge to s'_1 can be deleted.

This transformation is formally described by Figure 9.53. The precondition of the operation requires that the terminator of block b in loop l is a switch that branches on a non-constant value defined independently of the effects of evaluating loop l. In brief, the terminator of the preheader is replaced by a branch to one of $k+1$ copies of the preheader (one of which being the original, and the other k being a root of a cloned subgraph). In order to more immediately enable if-simplification, the terminator of block b is changed so that the

$$p = \mathit{preheader}(l) \qquad \mathit{term}(p) = \texttt{branch}\ \ell_h(\overline{e_i}) \qquad \mathit{term}(b) = \texttt{switch}\ r\ \overline{cv_i \mapsto \mathit{ref}_i}^{\,k}\ \mathit{ref}_d$$

$$r \text{ not defined in } l \qquad r \text{ is not known to be constant}$$

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l]{\text{duplicate}} (f'_1, D'_1, H'_1, L'_1, X'_1), (l'_1, \sigma_{L_1}) \xrightarrow[l]{\text{duplicate}} \cdots$$

$$\cdots \xrightarrow[l]{\text{duplicate}}{}^{*} (f'_k, D'_k, H'_k, L'_k, X'_k), (l'_k, \sigma_{L_k}) \qquad T' = \texttt{switch}\ r\ \overline{cv_i \mapsto \sigma_{L_i}(\ell_h)(\overline{e_i})}\ \ell_h(\overline{e_i})$$

$$p' = p[{}^{\mathit{term}(p)}/_{T'}] \qquad b' = b[{}^{\mathit{term}(b)}/_{\texttt{switch unit}\ \overline{cv_i \mapsto \mathit{ref}_i}\ \mathit{ref}_d}]$$

$$b_j = \mathit{block}_{f'_k}(\sigma_{L_j}(\mathit{lab}(b))) \qquad b'_j = b_j[{}^{\mathit{term}(b)}/_{\texttt{switch}\ cv_j\ \overline{cv_i \mapsto \mathit{ref}_i}\ \mathit{ref}_d}]$$

$$f'_k = (\ell, t, B, \ell_0) \qquad B' = B \setminus \{p, b, \overline{b_j}\} \cup \{p', b', \overline{b'_j}\} \qquad f' = (\ell, t, B', \ell_0)$$

$$D' = D'_k[b' \mapsto p' \mid b' \in \mathit{idom}(b') \in \mathit{body}_f(l) \wedge b' \notin \mathit{body}_f(l)] \qquad H' = H'_k[l'_k \mapsto p_{F_f}(l)]$$

$$(f', D'', H', L', X'_k) \xrightarrow[\{l, l'_1, \ldots, l'_k\}]{\text{dedicate preheader}}{}^{*} \cdot \xrightarrow[\{l, l'_1, \ldots, l'_k\} \times \mathit{exit}(l)]{\text{dedicate exit}}{}^{*} (f'', D_{out}, H_{out}, L_{out}, X_{out})$$

$$f'' \xrightarrow[\bigcup \mathit{def}(l'_i)]{\text{rewrite uses}}{}^{*} (f_{out}, \_)$$

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l, \langle f, b \rangle]{\text{unswitch}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Figure 9.53: Duplicate a loop containing a loop-invariant branch so each loop can be optimized independently.

switch value is `unit` (enforcing the default branch to be taken), and the terminator of each copy b_j of block b is changed so that the switch value is cv_j (enforcing the j-th edge to be taken). The dominator tree is constructed by taking the forest of partial dominator trees produced by cloning the loop and linking the blocks $\mathit{exit}(l)$ and the clones of blocks $\mathit{header}(l)$ to their immediate dominators. The loop nesting structure is constructed by attaching the root of each cloned loop nesting forest in the tree as a sibling of l. The predecessor of l (which now necessarily has multiple successors in each copy of the loop) and the exits of l (which now necessarily have a predecessor in each copy of the loop) are now undedicated and must be re-dedicated. In order to keep the graph in SSA form, each register defined by any cloned loop is rewritten with a fresh register and its uses within the cloned loop are updated. This is kept as two separate steps in theory to help aid the proof structures, but in practice

```
int i = 0;
for (; i < 100; i++) {
    f(i);
}
```

```
int i = 0;
for (; i < 100; i++) {
    f(i);
    i++;
    if (i >= 100) break;
    f(i);
}
```

```
int i = 0;
for (; i < 100; i++) {
    f(i);
    i++;
    f(i);
}
```

Figure 9.61: A Java source-level example of (manual) loop-unrolling.

loops are cloned in such a way that the duplicated version already has fresh registers for all definitions (and saves a walk over the graph).

9.6 Loop Unrolling

In loop bodies with no or very predictable branching performance that operate on memory in patterns causing infrequent cache misses, the loop condition and backedge may contribute a significant portion of the loop's total runtime. At the cost of code size, the loop body may be cloned and performed twice per loop iteration so that the end-of-loop tests occur once every two iterations. This transformation also stacks exponentially, such that applying this transformation n times on a single loop will cause the body to be performed 2^n times per iteration and reduce the number of backedge traversals by a factor of 2^n.

Figure 9.61 gives a source-level illustration of this transformation. In the first revision of this code, the function f is called with arguments in the range $[0, 100)$. Each iteration increments the counting variable i and checks that it is within bounds before the subsequent iteration. In second revision of this code, the loop post-condition and condition are copied explicitly into the body of the loop. In the third version of this code, the condition check within the body is removed, as it is redundant. This reduces the number of branch instructions required to perform this loop by half (and as the number of iterations are evenly divisible by 4, they could be reduced by an additional factor of two by applying the same procedure a second time).

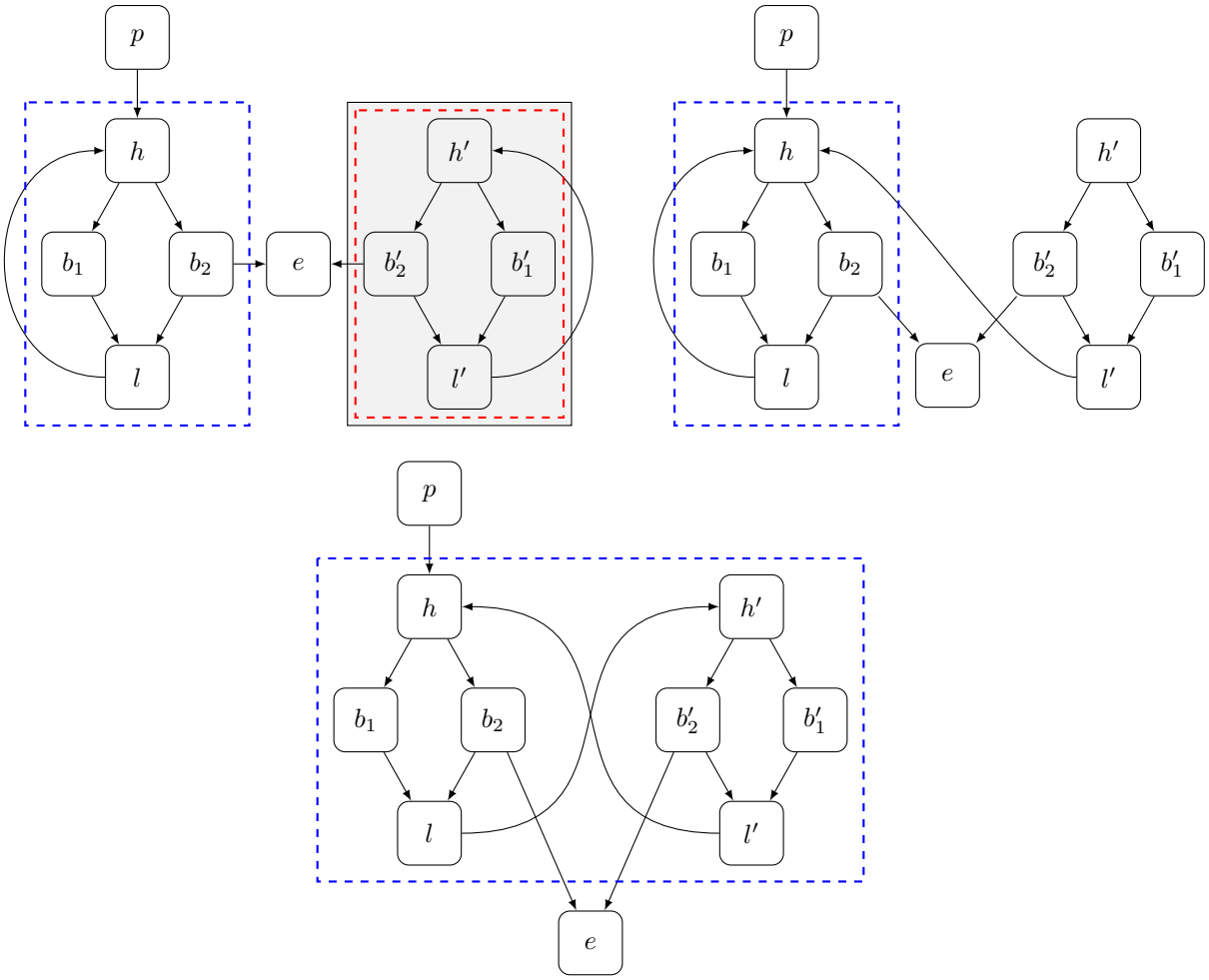


Figure 9.62: Unrolling the loop by duplicating the body and placing it on the loop's backedge. This creates an immediate opportunity to straighten the edge (l, h').

Figure 9.62 illustrates this transformation using an abstract control flow graph. In this example, the loop body is cloned as discussed in Section 8.3. Then, the backedge (l', h') of the cloned loop is replaced by an edge terminating in the original header, (l', h). Note that this destroys the outermost loop in the cloned subgraph. Symmetrically, the original backedge (l, h) is replaced by an edge terminating in the cloned header, (l, h'). In this step, the cloned subgraph is made reachable such that all cloned blocks are brought into the loop body. In this example, the edge (l, h') is immediately eligible for straightening. In loops where the latch has more than one successor, additional processing may be required to remove such a branch completely.

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l]{\text{duplicate}} (f', D', H', L', X'), (l', \sigma_L)$$

$$b_h = header(l) \qquad b'_h = header(l') \qquad \ell_{b_h} = lab(b_h) \qquad \ell_{b'_h} = lab(b'_h) \qquad b_l = latch(l)[\ell_h/\ell'_h]$$

$$b'_l = latch(l')[\ell'_h/\ell_h] \qquad f' = (\ell, B, t, \ell_0) \qquad B' = B \setminus \{latch(l), latch(l')\} \cup \{b_l, b'_l\}$$

$$f'' = (\ell, B', t, \ell_0) \qquad B_D = \{b' \mid idom(b') \in body_f(l_d) \wedge b' \notin body_f(l_d) \wedge l_d \in F_f\langle l \rangle\}$$

$$D'' = D'[b'_h \mapsto b_l][b' \mapsto nca_{D'}(pred(b')) \mid b' \in B_D]$$

$$H'' = H'[b' \mapsto b_h \mid b' \mapsto b'_h \in H'][b'_h \mapsto \varnothing]$$

$$L'' = L'[b \mapsto b_h \mid loop_{f'}(b) = l'] \qquad X'' = X' \setminus \{(\ell, \ell_{b'_h}) \in X\}$$

$$(f'', D'', H'', L'', X'') \xrightarrow[\{(l_d, e) \mid l_d \in F_f\langle l \rangle \wedge l_d \neq l \wedge e \in exit(l_d)\}]{\text{dedicate exit}}{}^{*} (f''', D_{out}, H_{out}, L_{out}, X_{out})$$

$$f''' \xrightarrow[def(l')]{\text{rewrite uses}}{}^{*} (f_{out}, \_)$$

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l]{\text{unroll}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Figure 9.63: Modify a loop so the body is performed twice per backedge traversal.

This transformation is formally described by Figure 9.63. This transformation starts, unsurprisingly, by duplicating the loop body. The resulting function is constructed by replacing the backedge reference in the original latch with a reference to the cloned header block, and replacing the backedge reference in the cloned latch with a reference to the original header block. The dominator tree is constructed by taking dominator forest produced by cloning the loop, linking the new latch to the old header and the old latch to the new header, and repairing the dominators that were broken by duplicating the predecessors of exit nodes (notice that these are not necessarily the exits themselves). The loop nesting forest produced from cloning the loop is repaired by adding the blocks belonging to loop l' to the body of loop l, nesting the cloned inner (non-root) loops of the new subgraph under loop l, and removing all references to the new loop l' from the exit relation, as it is no longer a loop distinct from l. In order to keep the graph in SSA form, each register defined by the cloned loop is rewritten with a fresh register and its uses within the cloned loop are

```
boolean first = true;
for (int i = 0; i < 10; i++) {
    if (first) {
        x[i] = y[i] * 2;
    } else {
        x[i] = y[i] * 3;
    }

    first = false;
}
```

```
x[1] = y[1] * 2;

for (int i = 1; i < 10; i++) {
    x[i] = y[i] * 3;
}
```

Figure 9.71: A Java source-level example of peeling the first iteration from a loop.

updated.

9.7 Loop Peeling

Loop *peeling* is a transformation that unrolls a constant number of loop iterations from the beginning (or end) of the loop and performs them outside of the loop body. In this section we present the form of loop peeling in which the first iteration is split from the loop. The form of loop peeling in which the last iteration is split behaves similarly, but is complicated in the case where a loop has multiple exits. This transformation can be beneficial in tandem with loop unrolling when the trip count of the loop is not a multiple of the target unrolling factor (in which case loop peeling can be applied until the trip count is a multiple of the target unrolling factor).

Figure 9.71 gives a source-level illustration of this transformation. In this example, the consequent of the if statement is taken only on the first iteration, after which the condition is flipped and the alternative is taken for the remainder of the loop iterations. Peeling this loop presents opportunities for both the peeled iteration and the remaining iterations to be simplified – the boolean flag no longer needs to be checked on either path, and the need for the variable is eliminated.

Figure 9.72 illustrates this transformation using an abstract control flow graph. In this example, the loop body is cloned as discussed in Section 8.3. Then, the backedge (l, h) of the

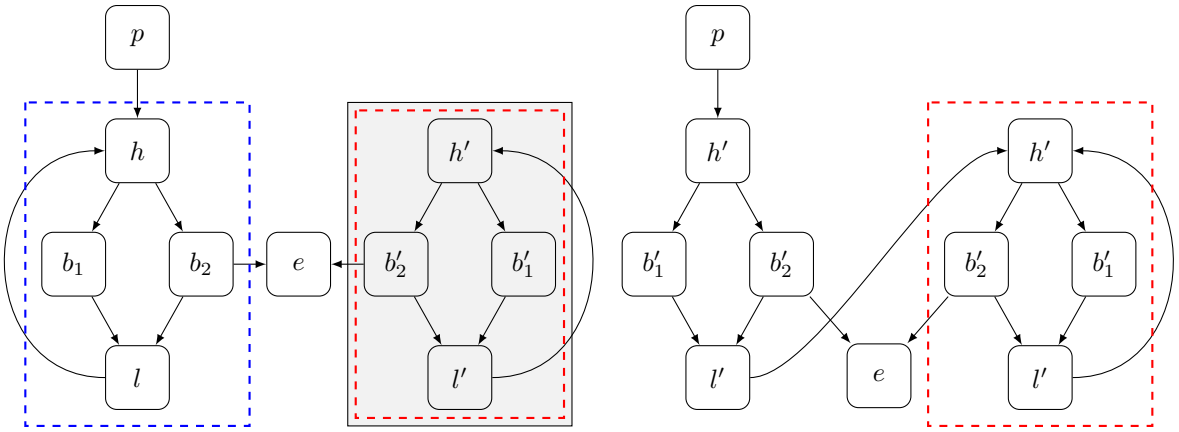


Figure 9.72: Peeling the loop by duplicating the body and placing it between the preheader and the backedge.

original loop is replaced by an edge terminating at the cloned header, (l, h'). Note that this destroys the original loop while making the blocks of the newly cloned subgraph reachable.

This transformation is formally described by Figure 9.73. Again, the transformation starts by duplicating the loop body. The resulting function is constructed by replacing the backedge reference in the original latch with a reference to the cloned header block. Unlike in loop unrolling, we do not perform the symmetric operation in the cloned backedge. The dominator tree is constructed by taking dominator forest produced by cloning the loop, replacing the original backedge with an edge from the old latch to the new header, and repairing the dominators that were broken by duplicating the predecessors of exit nodes (notice that these are not necessarily the exits themselves). The loop nesting forest produced from cloning the loop is repaired by attaching the root of the cloned nesting forest as well as loops immediately nested in l as a sibling of l. As this operation destroys the original loop, we then remove all references of l in the loop nesting relation, body sets, and exit sets. All exits of loop l' necessarily undedicated, as they each have a predecessor in both the original body of l and the current body of l', and are re-dedicated. In order to keep the graph in SSA form, each register defined by the cloned loop is rewritten with a fresh register and its uses within the cloned loop are updated.

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l]{\text{duplicate}} (f', D', H', L', X'), (l', \sigma_L) \qquad b_h = header(l)$$

$$b'_h = header(l') \qquad \ell_{b_h} = lab(b_h) \qquad \ell_{b'_h} = lab(b'_h) \qquad b_{h_p} = header(p_F(l))$$

$$b_l = latch(l)[\ell_{b_h}/\ell_{b'_h}] \qquad f' = (\ell, B, t, \ell_0) \qquad B' = B \setminus \{latch(l)\} \cup \{b_l\}$$

$$f'' = (\ell, B', t, \ell_0) \qquad B_D = \{b' \mid idom(b') \in body_f(l_d) \wedge b' \notin body_f(l_d) \wedge l_d \in F_f\langle l \rangle\}$$

$$D'' = D'[b'_h \mapsto b_l][b' \mapsto nca_{D'}(pred(b')) \mid b' \in B_D]$$

$$H'' = H'[b'_h \mapsto b_{h_p}][header(c) \mapsto b_{h_p} \mid c \in child_F(l)][b_h \mapsto \varnothing]$$

$$L'' = L' \setminus \{(\ell, \ell_{b_h}) \in L'\} \qquad X'' = X' \setminus \{(\ell, \ell_{b_h}) \in X'\}$$

$$(f'', D'', H'', L'', X'') \xrightarrow[l']{\text{dedicate preheader}} \cdot \xrightarrow[\{(l',e) \mid e \in exit(l')\}]{\text{dedicate exit}}{}^{*} (f''', D_{out}, H_{out}, L_{out}, X_{out})$$

$$f''' \xrightarrow[defs(l')]{\text{rewrite uses}}{}^{*} (f_{out}, \_)$$

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l]{\text{peel}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Figure 9.73: Peeling a single iteration from the loop to be performed prior to entering the loop boundary.

Appendix

Here we provide the proofs of maintenance properties of optimization transformations presented in Chapter 9.

9.A Straightening

In this section, we refer the *straighten* transformation (notated below). For specific details (including names of intermediate components), refer to Figure 9.11.

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[a,b]{\text{straighten}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Theorem 9.A.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. We prove by filling in the slots of Template 6.A.1.

[**Slot → Refinement**] We refine the *trans* relation by replacing clones of blocks a and b with a clone of block a' and appending the register-mapped stream of block b to the streams of clones of a. For brevity, we decompose the stream s_t into (s'_t, T_t).

$$trans(\langle f_t, b_t \rangle) = \begin{cases} a'[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(a) \vee b_t \in \mathcal{C}(b) \\ b_t & otherwise \end{cases}$$

$$trans(\langle f_t, b_t, (s_t, T_t, \hat{s}_t) \rangle) = \begin{cases} s'_t \cup (s' \cup stream(b))[\overrightarrow{\sigma_{f_t}}], trans(\hat{s}_t) & b_t \in \mathcal{C}(a) \\ s_t, trans(\hat{s}_t) & otherwise \end{cases}$$

We add the following additional relationship between register contexts.

$$\gamma_1 = \gamma_2$$

[Slot → Invokability] If $b_{t_1} \in \mathcal{C}(a)$, then $b_{t_2} = a'[\overrightarrow{\sigma_{f_{t_1}}}]$. Otherwise, $b_{t_2} = b_{t_1}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[Slot → Asymmetric Evaluation] By the refined *trans* relation, all temporarily diverging steps of evaluation occur within blocks a and b.

Collapsed Edge First, we cover the case where the $(n+1)$-th step of evaluation of f transfers control from block b_t to block $b_{t_1} \in \mathcal{C}(b)$ by application of rule E-Switch. Here, $b_t \in \mathcal{C}(a)$ as $pred_f(b) = \{a\}$ and $dom(param((b_{t_1})) = \langle r_1, \ldots, r_k \rangle$.

$$\begin{aligned}(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \rightarrow^n\ & (\langle p, f_{t_1}, b_t \rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{branch}\ \ell_b(\overline{e_i}), \hat{s}_{t_1}) \\ \rightarrow\ & (\langle p, f_{t_1}, b_{t_1} \rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_1}), \hat{s}_{t_1})\end{aligned}$$

By our strengthened inductive hypothesis, the n'-th step of evaluation of f_{out} stops in function $f_{t_2} = trans(f_{t_1})$ in block $b_{t_2} = trans(\langle f_{t_1}, b_t \rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. Here, $b_{t_2} \in \mathcal{C}(a')$ and $s'_t = s'$. The next k steps

of evaluation of f_{out} proceed from repeated application of rule E-Move. For brevity, we let $s_{t_2} = stream(b)[\overrightarrow{\sigma_{f_{t_1}}}]$.

$$(p[^{f}/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \to^{n'}$$

$$(\langle p[^{f}/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ s'[\overrightarrow{\sigma_{f_{t_2}}}] \cup s_{t_2}, trans(\hat{s}_{t_1}) \to$$

$$(\langle p[^{f}/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r_j \mapsto \gamma_2(e_j)}^{(j<i)}] \mid \mu' \mid \Psi';\ \langle s_i, \ldots\rangle[\overrightarrow{\sigma_{f_{t_2}}}] \cup s_{t_2}, trans(\hat{s}_{t_1})) \to^*$$

$$(\langle p[^{f}/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ s_{t_2}, trans(\hat{s}_{t_1}))$$

Streams s_{t_1} and s_{t_2} are consistent with the strengthened induction, and as register contexts γ_1 and γ_2 are consistent with the strengthened induction, they remain consistent after the parallel addition of registers in the last step of evaluation.

Collapsed Block Next, we cover the cases where the n-th step of evaluation of f is within a clone of block a or a clone of block b. First, we consider when the $(n+1)$-th step of evaluation of f evaluates an instruction. By our strengthened inductive hypothesis, the n'-th step of evaluation of f_{out} stops at a state with a symmetric function, block, stream, and register context. Here, both streams have identical instruction parts (only differing at the terminator of a clone of block a). By application of Lemma 6.B.1, the resulting streams are the input streams with their head elements removed, in which case the tails of both streams remain consistent with the strengthened induction by construction, or both streams are ϵ, also consistent with the strengthened induction. Additionally, the resulting register contexts are extended with the same register-value pair, which does not violate the strengthened induction. Now, consider when the $(n+1)$-th step of evaluation of f evaluates the terminator of a clone of block b (the other terminator was covered above). By construction, the terminators of each stream are identical. By application of Lemma 6.B.3, the resulting blocks are identical, as are their unmodified streams. Additionally, the resulting

register contexts are extended with the same set of register-value pairs, which does not violate the strengthened induction.

[**Slot ← Refinement**] We refine the translation of blocks by mapping clones of block a' to clones of block a or block b, depending on how far along execution is within the block. We refine the translation of streams by replacing the tail-end of the stream with the original terminator of block a while evaluation remains in the first segment of a clone of block a'. Notice that if execution is already within the second segment, the instruction stream is already in the correct symmetric form. In the following, we use s_t to mean the instruction stream (without a return context) in the *same step of evaluation* of function f_{out} as when the $trans^{-1}$ relation is applied. For brevity, let $S_s = |stream(b)|$, $S_p = |param(b)|$, and $S_{p+s} = S_p + S_s$. Also let $S_{\langle \ldots,i \rangle}$ denote the first i elements of sequence S in the following.

$$trans^{-1}(\langle f_t, b_t \rangle) = \begin{cases} a[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(a') \wedge |s_t| > S_s \\ b[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(a') \wedge |s_t| \leq S_s \\ b_t & otherwise \end{cases}$$

$$trans^{-1}(\langle f_t, b_t, (s_t, \hat{s}_t) \rangle) = \begin{cases} s_{t\langle \ldots,|s_t|-S_{p+s}) \rangle} \cup term(a)[\overrightarrow{\sigma_{f_t}}], trans^{-1}(\hat{s}_t) & b_t \in \mathcal{C}(a') \wedge |s_t| > S_s \\ s_t, trans^{-1}(\hat{s}_t) & otherwise \end{cases}$$

Now, we define the following relationship stating that registers occurring in γ_2 but not γ_1 must be a parameter to a clone of block b. This relation allows the evaluation of f_{out} to sequentially assign values to registers that are assigned all at once in the parallel evaluation.

$$\gamma_1 \subseteq \gamma_2 \qquad \gamma_2 \setminus \gamma_1 \subseteq \bigcup_{b_i \in \mathcal{C}(b)} dom(param(b_i))$$

[**Slot ← Invokability**] If $b_{t_2} \in \mathcal{C}(a')$, then $b_{t_1} = a[\overrightarrow{\sigma_{f_{t_2}}}]$. Otherwise $b_{t_1} = b_{t_2}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[**Slot ← Asymmetric Evaluation**] By the refined $trans^{-1}$ relation, all temporarily diverging steps of evaluation occur within block a'.

First Segment Here, we cover the case where the n'-th and $(n'+1)$-th steps of evaluation of f_{out} are in a block in $\mathcal{C}(a')$ such that $|s_t| > S_s$.

$$(p[^{f}/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \rightarrow^{n'}$$

$$(\langle p[^{f}/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ I, s_{t_2}, \hat{s}_{t_2}$$

By our strengthened inductive hypothesis, the n-th step of evaluation of f stops in function $f_{t_1} = trans^{-1}(f_{t_2})$ in block $trans^{-1}(\langle f_{t_2}, b_t\rangle) \in \mathcal{C}(a)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. Here, the leading instructions of each stream are identical (up to the terminator of the clone of block a). First, we consider the case where the leading instruction is not a function call. By application of Lemma 6.B.1, the resulting streams are either the input streams with their head elements removed or both ϵ. In the former case, the tails of both streams still have an identical leading instruction (and this case will apply again in the subsequent steps of evaluation), or the leading element of the stream in the evaluation of f is a terminator and the leading elements of the stream in the evaluation of f_{out} is a sequence of move instructions simulating the collapsed edge (and the following case will apply on the subsequent step of evaluation). In the latter case, the ϵ streams are trivially consistent with the strengthened induction. Now, we consider the case where the leading instruction is a function call. By application of Lemma 6.B.2, the resulting functions are either identical, or are instances of functions f and f_{out}. In the former case, the resulting

blocks are identical. In the latter case, the resulting blocks are either identical, or clones of blocks a and a'. In all cases, the blocks and their unmodified streams are consistent with the strengthened induction.

Collapsed Edge Here, we cover the case where the n'-th and $(n'+1)$-th steps of evaluation of f_{out} are in a block in $\mathcal{C}(a')$ such that $S_{p+s} \geq |s_t| > S_s$. In this case, evaluation of f_{out} is somewhere within the sequence of `move` instructions that simulates the collapsed edge. Let s_{t_2} have the form $\langle s'_i, s'_{i+1}, \dots \rangle \cup \mathit{stream}(b)[\overrightarrow{\sigma_{f_{t_2}}}]$ in the following.

$$(p[f/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{v_{t_i}})) \rightarrow^{n'}$$

$$(\langle p[f/_{f_{out}}], f_{t_2}, b_{t_2} \rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ s_{t_2}, \hat{s}_{t_2} \rightarrow$$

$$(\langle p[f/_{f_{out}}], f_{t_2}, b_{t_2} \rangle \mid \gamma_2[r_i \mapsto \gamma_2(e_i)] \mid \mu' \mid \Psi';\ \langle s'_{i+1}, \dots \rangle \cup \mathit{stream}(b)[\overrightarrow{\sigma_{f_{t_2}}}], \hat{s}_{t_2})$$

By our strengthened inductive hypothesis, the n-th step of evaluation of f stops in function $f_{t_1} = \mathit{trans}^{-1}(f_{t_2})$ in block $\mathit{trans}^{-1}(\langle f_{t_2}, b_t \rangle) \in \mathcal{C}(a)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \rightarrow^{n}$$

$$(\langle p, f_{t_1}, \mathit{trans}^{-1}(\langle f_{t_2}, b_t \rangle) \rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ \mathit{trans}^{-1}(\langle f_{t_2}, b_{t_2}, s_{t_2} \rangle), \mathit{trans}^{-1}(\hat{s}_{t_2}))$$

Here, $\mathit{trans}^{-1}(\langle f_{t_2}, b_{t_2}, s_{t_2} \rangle) = \mathit{term}(a)[\overrightarrow{\sigma_{f_{t_2}}}]$. No additional steps of evaluation are necessary as the blocks and their streams are consistent with the strengthened induction, and γ_2 asymmetrically assigned new values (which does not violate the register context superset relation). It is important to note, in order to bridge the gap between this case and the next, that the set of registers assigned by this sequence of instructions are equivalent to the parameters of the subsequent block in the evaluation of f.

Second Segment Here, we cover the case where the n'-th step of evaluation of f_{out} is in a block in $\mathcal{C}(a')$ such that $|s_t| \leq |\mathit{stream}(b)|$. The block of the parallel evaluation is in $\mathcal{C}(b)$

and the streams of both evaluations are identical. This case can then proceed with the rest of the symmetric evaluation cases. □

Corollary 9.A.2. The set of paths over block labels from ℓ_0 to $\ell_{b'}$ in $G_{f_{out}}$ for some block $b' \in B$ where $lab(b') = \ell_{b'}$ can be constructed by replacing the subpath $\langle \ell_a, \ell_b, lab(c) \rangle$ with the edge $(\ell_a, lab(c))$ for every block $c \in succ_f(b)$ in the set of such paths in G_f.

Theorem 9.A.3. If f is in canonical form, then f_{out} is in canonical form.

Proof. Let F' be the loop nesting forest reconstructed from $(H_{out}, L_{out}, X_{out})$ and let $l \in F$ and $l' \in F'$ be loops that share a header label. By construction of F', both loops have the same header, latch, and exits. As the edge cannot cross a loop boundary, both loops also have the same preheader. Then, any violation in l' must also be present in l. □

Theorem 9.A.4. If f is in LCSSA form, then f_{out} is in LCSSA form.

Proof. Block a' defines and uses the same set of registers as block a and block b. As neither block a nor b contains any uses outside of an ancestor of $loop_f(a)$ or defines any registers used outside of $loop_f(a)$, construction of a' introduces no violations to function f_{out}. □

Theorem 9.A.5. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f_{out}$ is well-typed.

Proof. By definition, $nonlocal_{f_{out}}(a')$ is equivalent to

$$implicit(a) \cup \boxed{(implicit(b) \ominus def(a)) \bigcup_{s \in succ_{f_{out}}(a')} nonlocal_{f_{out}}(s)} \ominus def(a')$$

where the highlighted term is equivalent to $nonlocal_f(b)$. Thus, $nonlocal(a) = nonlocal(a')$. Now, we show that the stream of a' is well-typed. Let $\Gamma_0 = param(a), nonlocal_f(a)$ and $\Gamma_0' = param(b), nonlocal_f(b)$. By inversion of rule T-Block, $(p\ f \mid \Gamma_0) \vdash stream(a)$ and $(p\ f \mid \Gamma_0') \vdash stream(b)$. Let Γ_k be the register typing context used in rule T-Ref in the former derivation. Here, $nonlocal_f(b) \subseteq \Gamma_k$ and $\Gamma_k \vdash e_i : t_i$ for every $(r_i : t_i) \in param(b)$.

Now, we show that $(p\ f_{out} \mid \Gamma_0) \vdash \mathit{stream}(a')$. The derivation begins in identical fashion as the first k steps of $\mathit{stream}(a)$ as both streams have identical leading k instructions. Next, the derivation $(p\ f_{out} \mid \Gamma_k) \vdash s' \cup \mathit{stream}(b)$ proceeds by with zero or more applications of rule T-Move. Here, each instruction $r_i \leftarrow \texttt{move}(e_i)$ adds the pair (r_i, t_i) to the register typing context as each expression $\Gamma_k \vdash e_i : t_i$ as shown above. This yields the expanded register typing context $\Gamma_k, \mathit{param}(b)$. Lastly, the derivation $(p\ f_{out} \mid \Gamma_k, \mathit{param}(b)) \vdash \mathit{stream}(b)$ proceeds in identical fashion to the derivation of $\mathit{stream}(b)$ above as $\mathit{nonlocal}(b) \subset \Gamma_k$, also shown above. □

Theorem 9.A.6. The unique dominator tree of $G_{f_{out}}$ is D_{out}.

Proof. Let b' be a block such that $p_D(b') = b$. The rightmost common subpath in the set of paths from the entry block to b' in G_f is suffixed by $\langle a, b \rangle$. The rightmost common subpath in such paths in $G_{f_{out}}$ is suffixed by $\langle a' \rangle$ by Corollary 9.A.2. Therefore, $\mathit{idom}_{f_{out}}(B') = a'$. It also follows from Corollary 9.A.2 that no other immediate dominator is modified. □

Theorem 9.A.7. If f is in caonical form, then F_{out} reconstructed from $(H_{out}, L_{out}, X_{out})$ is the unique loop nesting forest of $G_{f_{out}}$.

Proof. Let $C \subseteq B$ be a strongly connected component of G_f. It follows from Corollary 9.A.2 that C is also a strongly connected component of $G_{f_{out}}$ when $a \notin C$ and $C \setminus \{a, b\} \cup \{a'\}$ is a strongly connected component of $G_{f_{out}}$ otherwise. The block b can neither be a header nor an exit of a loop as $\mathit{loop}_f(a) = \mathit{loop}_f(b)$ and $|\mathit{pred}_f(b)| = 1$ (and a canonical form header has exactly two predecessors). Then, $F_{f_{out}}$ can be constructed simply by removing references to block b from the set of loop bodies. This exactly mirrors the construction of the triple $(H_{out}, L_{out}, X_{out})$. □

9.B If Simplification

In this section, we refer to the *if simplify* transformation (notated below). For specific details (including names of intermediate components), refer to Figure 9.21.

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[\langle f,b \rangle]{\text{if simplify}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Our first step to proving that the evaluation of functions f and f_{out} are symmetric is to show that evaluation of function f and the intermediate function f' (the state of the function directly before the removal of the edges from b) are symmetric.

Lemma 9.B.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f' can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. We prove by filling in the slots of Template 6.A.1.

[**Slot → Refinement**] We refine the translation of blocks by mapping clones of block b to clones of block b'. We refine the translation of streams by replacing the terminator of clones of block b with a terminator that unconditionally switches to the chosen default block. For brevity, we decompose the stream s_t into $(s'_t, \hat{s}_t)$.

$$trans(\langle f_t, b_t \rangle) = \begin{cases} b'[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(b) \\ b_t & otherwise \end{cases}$$

$$trans(\langle f_t, b_t, (s_t, \hat{s}_t) \rangle) = \begin{cases} (s'_t, T[\overrightarrow{\sigma_{f_t}}]), trans(\hat{s}_t) & b_t \in \mathcal{C}(b) \\ s_t, trans(\hat{s}_t) & otherwise \end{cases}$$

We add the following additional relationship between register contexts.

$$\gamma_1 = \gamma_2$$

[Slot → Invokability] If $b_{t_1} \in \mathcal{C}(b)$, then $b_{t_2} = b'[\overrightarrow{\sigma_{f_{t_1}}}]$. Otherwise, $b_{t_2} = b_{t_1}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[Slot → Asymmetric Evaluation] By the refined *trans* relation, all temporarily diverging steps of evaluation occur at the terminator of a clone of block b.

Simplified Switch In the asymmetric case, the $(n+1)$-th step of evaluation of f transfers control away from block $b_t \in \mathcal{C}(b)$ by application of rule E-Switch. By the conditions of the transformation, control is transferred to block b_{t_1} over all execution traces (including this one). Here, $dom(param(b_{t_1})) = \langle \overline{r_i} \rangle$ and $ref = \ell_{b'}(\overline{e_i})$ is the block reference taken by this step of evaluation.

$$\begin{aligned}(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \rightarrow^n\ & (\langle p, f_{t_1}, b_t \rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{switch}\ v\ \overline{cv_i \mapsto ref_i}\ ref_d, \hat{s}_{t_1}) \\ \rightarrow\ & (\langle p, f_{t_1}, b_{t_1} \rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_1}), \hat{s}_{t_1})\end{aligned}$$

By our strengthened inductive hypothesis, the n'-th step of evaluation of f' stops in function $f_{t_2} = trans(f_{t_1})$ on a terminator that transfers control away from $trans(\langle f_{t_1}, b_t \rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. Then, the $(n'+1)$-th step of evaluation of f' proceeds from application of rule E-Switch by taking the default switch case (equivalent to *ref* by construction). Here, $b_{t_2} = trans(\langle f_{t_1}, b_{t_1} \rangle)$ and, as no block

parameters are modified by the transformation, $dom(param(b_{t_2})) = \langle \overline{r_i} \rangle$.

$$(p[f/f'] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f'(\overline{v_{t_i}})) \rightarrow^{n'}$$

$$(\langle p[f/f'], f_{t_2}, trans(\langle f_{t_1}, b_t \rangle) \rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ T[\overrightarrow{\sigma_{f_{t_2}}}], trans(\hat{s}_{t_1})) \rightarrow$$

$$(\langle p[f/f'], f_{t_2}, b_{t_2} \rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_2}), trans(\hat{s}_{t_1}))$$

First, suppose that $b_{t_1} \in \mathcal{C}(b)$. In this case, $b_{t_1} = b_t$ and $b_{t_2} = trans(\langle f_{t_1}, b_t \rangle)$. Additionally, $stream(b_{t_2}) = stream(b)[\overrightarrow{\sigma_{f_{t_1}}}]$, and $stream(b_{t_2}) = stream(b')[\overrightarrow{\sigma_{f_{t_2}}}]$. These streams differ only by the terminator by construction and are consistent with the strengthened induction. In all other cases, the blocks and streams are identical in both evaluations and both pairs of components are consistent with the strengthened induction. In either case, as register contexts γ_1 and γ_2 are consistent with the strengthened induction, they remain consistent after the parallel addition of registers in the last step of evaluation.

Simplified Block Next, we cover the cases where the n-th step of evaluation of f is earlier within block b (not yet at the terminator). This case is trivial by application of Lemma 6.C.1.

[**Slot ← Refinement**] We refine the translation of blocks by mapping clones of block b' to clones of block b. We refine the translation of streams by replacing the terminator of clones of block b' with the original terminator of block b – this simply adds extra switch cases which are unmatchable during any execution. For brevity, we decompose the stream s_t into $(s'_t, \hat{s}_t)$.

$$trans(\langle f_t, b_t \rangle) = \begin{cases} b[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(b') \\ b_t & otherwise \end{cases}$$

$$trans(\langle f_t, b_t, (s_t, \hat{s}_t) \rangle) = \begin{cases} (s'_t, term(b)[\overrightarrow{\sigma_{f_t}}]), trans(\hat{s}_t) & b_t \in \mathcal{C}(b') \\ s_t, trans(\hat{s}_t) & otherwise \end{cases}$$

We add the following additional relationship between register contexts.

$$\gamma_1 = \gamma_2$$

[**Slot ← Invokability**] If $b_{t_2} \in \mathcal{C}(b')$, then $b_{t_1} = b[\overrightarrow{\sigma_{f_{t_2}}}]$. Otherwise $b_{t_1} = b_{t_2}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[**Slot ← Asymmetric Evaluation**] By the refined $trans^{-1}$ relation, all temporarily diverging steps of evaluation occur at the terminator of a clone of block b'.

Simplified Switch In the asymmetric case, the $(n'+1)$-th step of evaluation of f' transfers control away from block $b_t \in \mathcal{C}(b')$ by application of rule E-Switch. The terminator T switches on the `unit` value, thus the default switch case is always taken. Here, $dom(param(b_{t_2})) = \langle \overline{r_i} \rangle$ and $ref = \ell_{b'}(\overline{e_i})$ is the block reference taken by this step of evaluation.

$$(p[f/f'] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f'(\overline{v_{t_i}})) \to^{n'}$$

$$(\langle p[f/f'], f_{t_2}, b_t \rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ T[\overrightarrow{\sigma_{f_{t_2}}}], \hat{s}_{t_2}) \to$$

$$(\langle p[f/f'], f_{t_2}, b_{t_2} \rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_2}), \hat{s}_{t_2})$$

By our strengthened inductive hypothesis, the n-th step of evaluation of f stops in function $f_{t_1} = trans^{-1}(f_{t_2})$ on a terminator that transfers control away from $trans^{-1}(\langle f_{t_2}, b_t \rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. Then, the $(n+1)$-th step of evaluation of f proceeds from application of rule E-Switch by taking the block reference equivalent to *ref*. Here, $b_{t_1} = trans(\langle f_{t_2}, b_{t_2} \rangle)$ and, as no block parameters

are modified by the transformation, $dom(param(b_{t_2})) = \langle \overline{r_i} \rangle$.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n$$

$$(\langle p, f_{t_1}, trans^{-1}(\langle f_{t_2}, b_t \rangle) \rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{switch}\ v\ \overline{cv_i \mapsto ref_i}\ ref_d, trans^{-1}(\hat{s}_{t_2})) \to$$

$$(\langle p, f_{t_1}, b_{t_1} \rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_1}), trans^{-1}(\hat{s}_{t_2}))$$

First, suppose that $b_{t_2} \in \mathcal{C}(b')$. In this case, $b_{t_2} = b_t$ and $b_{t_1} = trans^{-1}(\langle f_{t_2}, b_t \rangle)$. Additionally, $stream(b_{t_2}) = stream(b')[\overrightarrow{\sigma_{f_{t_1}}}]$, and $stream(b_{t_1}) = stream(b)[\overrightarrow{\sigma_{f_{t_1}}}]$. These streams differ only by the terminator by construction and are consistent with the strengthened induction. In all other cases, the blocks and streams are identical in both evaluations and both pairs of components are consistent with the strengthened induction. In either case, as register contexts γ_1 and γ_2 are consistent with the strengthened induction, they remain consistent after the parallel addition of registers in the last step of evaluation.

Simplified Block Next, we cover the cases where the n'-th step of evaluation of f' is within block b'. This case is trivial by application of Lemma 6.C.1. □

Theorem 9.B.2. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. By application of Lemma 9.B.1 and Theorem 8.B.8. □

Corollary 9.B.3. The set of paths over block labels from ℓ_0 to $\ell_{b'}$ in $G_{f'}$ for some block $b' \in B$ where $lab(b') = \ell_{b'}$ is identical to the set of such paths in G_f.

Theorem 9.B.4. If f is in canonical form, then f_{out} is in canonical form.

Proof. By Corollary 9.B.3, f' is also in canonical form. The rest then follows by application of Theorem 8.B.10. □

Theorem 9.B.5. If f is in LCSSA form, then f_{out} is in LCSSA form.

Proof. If f is in LCSSA form, then the intermediate function f' is also in LCSSA form as f' is constructed by removing a single use from f. The rest then follows by application of Theorem 8.B.11. □

Theorem 9.B.6. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f_{out}$ is well-typed.

Proof. Let $\Gamma_0 = \mathit{param}(b), \mathit{nonlocal}(b)$. As $p \mid f$ is well-typed, $(p\ f \mid \Gamma_0) \vdash \mathit{stream}(b)$ by inversion of rule T-Block. Let Γ_k be the register typing environment used in rule T-Switch. By construction of block b', its parameters, nonlocal parameters, and instruction stream (up to the terminator) are identical to that of block b. Derivation of $(p\ f' \mid \Gamma_0) \vdash \mathit{stream}(b')$ begins in identical fashion as the first k steps of *streamb*) as both streams have identical leading k instructions. Now, $(p\ f' \mid \Gamma_k) \vdash T$ by application of rule T-Switch as each reference was shown to be well-typed with respect to the same register typing environment above. Then, $p[f/f'] \mid f'$ is well-typed as each block is well-typed and, as no implicit parameters were added to any block, $\mathit{nonlocal}_{f_{out}}(\mathit{entry}(f_{out})) = \emptyset$. The rest then follows by application of Theorem 8.B.12. □

Theorem 9.B.7. The unique dominator tree of $G_{f_{out}}$ is D_{out}.

Proof. By Corollary 9.B.3, D_f is the unique dominator tree of $G_{f'}$. The rest then follows by application of Theorem 8.B.13. □

Theorem 9.B.8. If f is in canonical form, then F_{out} reconstructed from $(H_{out}, L_{out}, X_{out})$ is the unique loop nesting forest of $G_{f_{out}}$.

Proof. By Corollary 9.B.3, F_f is the unique loop nesting forest of $G_{f'}$. The rest then follows by application of Theorem 8.B.14. □

9.C Jump Simplification

In this section, we refer to the *jump simplify* transformation (notated below). For specific details (including names of intermediate components), refer to Figure 9.31.

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[p,b]{\text{jump simplify}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Our first step to proving that the evaluation of functions f and f_{out} are symmetric is to show that evaluation of function f and the intermediate function f' (the state the function directly before SSA reconstruction) are symmetric.

Lemma 9.C.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f' can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. We prove by filling in the slots of Template 6.A.1.

[**Slot → Refinement**] We define the predicate $from_p$, as follows, that determines whether or not the current state of evaluation has entered a clone of block b via a clone of block p. This predicate will be necessary to distinguish whether or not symmetric evaluation of function f' has entered a clone block b or a clone of block b'. This predicate is true whenever the evaluation path ρ contains an element referencing the terminator of a clone of p (at which point a clone of block b has been entered) and is not followed by the terminator of a clone of block b (at which point a clone of block b has been exited).

$$from_p(\rho) = \begin{cases} true & \rho = \langle \ldots, (\langle f'', p'', term(p'')\rangle, b''), \overline{(c_i, b_i)}\rangle \\ & \text{where } p'' \in \mathcal{C}(p) \wedge \forall i \cdot c_i \neq term(b[\overrightarrow{\sigma_{f''}}]) \\ false & otherwise \end{cases}$$

We refine the translation of blocks by mapping clones of block b to clones of block b *or* block b', as described above. We refine the translation of streams by rewriting the terminator of p. As b and b' have identical streams in function f', no stream translation is necessary when $b_t \in \mathcal{C}(b)$. For brevity, we decompose the stream s_t into (s'_t, T_t).

$$trans_\rho(\langle f_t, b_t \rangle) = \begin{cases} p'[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(p) \\ b'[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(b) \wedge from_p(\rho) \\ b_t & otherwise \end{cases}$$

$$trans_\rho(\langle f_t, b_t, (s_t, \hat{s}_t) \rangle) = \begin{cases} (s'_t, T[\overrightarrow{\sigma_{f_t}}]), trans_\rho(\hat{s}_t) & b_t \in \mathcal{C}(p) \\ s_t, trans_\rho(\hat{s}_t) & otherwise \end{cases}$$

We add the following additional relationship between register contexts.

$$\gamma_1 = \gamma_2$$

[Slot → Invokability] Application of this transformation requires that b has multiple forward-predecessors, thus $b_{t_1} = b$. If $b_{t_1} = p$, then $b_{t_2} = p'[\overrightarrow{\sigma_{f_{t_1}}}]$. Otherwise, $b_{t_2} = b_{t_1}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[Slot → Asymmetric Evaluation] By the refined *trans* relation, all temporarily diverging steps of evaluation occur at the terminator of a clone of p, or within a clone of block b' when entered from a clone of block p.

Altered Jump First, we cover the case where the $(n+1)$-th step of evaluation of f transfers control from block $b_t \in \mathcal{C}(p)$ to block $b_{t_1} \in \mathcal{C}(b)$ by application of rule E-Switch.

$$\begin{aligned}(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n_{\rho_1}\ & (\langle p, f_{t_1}, b_t\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{branch}\ \ell_b(\overline{e_i}), \hat{s}_{t_1}) \\ \to_{\rho_2}\ & (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_1}), \hat{s}_{t_1})\end{aligned}$$

By our strengthened inductive hypothesis, the n'-th step of evaluation of f' stops in function $f_{t_2} = \mathit{trans}(f_{t_1})$ on a terminator that transfers control from block $\mathit{trans}(\langle f_{t_1}, b_t\rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. Then, the $(n'+1)$-th step of evaluation of f' proceeds by application of rule E-Switch. Here, $b_{t_2} \in \mathcal{C}(b')$. It is worth nothing that the terminating element of ρ_2 is the terminator of b_t (and as there is yet no subsequent terminator of b_{t_1}), $\mathit{from}_p(\rho_2)$ is true in the subsequent step of evaluation.

$$\begin{aligned}&(p[{}^{f}/_{f_{out}}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f'(\overline{v_{t_i}})) \to^{n'} \\ &(\langle p[{}^{f}/_{f_{out}}], f_{t_2}, \mathit{trans}(\langle f_{t_1}, b_t\rangle)\rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ \texttt{branch}\ \ell_{b'}(\overline{e_i}), \mathit{trans}(\hat{s}_{t_1})) \to \\ &(\langle p[{}^{f}/_{f_{out}}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r'_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_2}), \mathit{trans}(\hat{s}_{t_1}))\end{aligned}$$

Fresh Block Next, we cover the cases where the n-th step of evaluation of f is within block b and the predicate $\mathit{from}_p(\rho)$ is true, where ρ is the evaluation path at the n-th step of evaluation of f. If evaluation of f' is not yet at a terminator, then this case is trivial by application of Lemma 6.C.1. Now, consider when the $(n+1)$-th step of evaluation of f evaluates the terminator of a clone of block b. By construction, the terminators of each stream are identical. By application of Lemma 6.B.3, the resulting blocks are identical – it cannot be the case that evaluation of f' transfers control to block p' and evaluation of f transfers control to block p, as block b can neither be a header nor a latch of a loop. Then, the unmodified streams of blocks p and p' in the same function are consistent with the

strengthened induction by construction. The resulting register contexts are extended with the same register-value pair from both lemmas, which does not violate the strengthened induction.

[Slot ← Refinement] We refine the translation of blocks by mapping clones of blocks p' and b' to clones of block p and b, respectively. For brevity, we decompose the stream s_t into (s'_t, T_t).

$$trans(\langle f_t, b_t \rangle) = \begin{cases} p[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(p') \\ b[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(b') \\ b_t & otherwise \end{cases}$$

$$trans(\langle f_t, b_t, (s_t, \hat{s}_t) \rangle) = \begin{cases} (s'_t, term(p)[\overrightarrow{\sigma_{f_t}}]), trans(\hat{s}_t) & b_t \in \mathcal{C}(p) \\ s_t, trans(\hat{s}_t) & otherwise \end{cases}$$

We add the following additional relationship between register contexts.

$$\gamma_1 = \gamma_2$$

[Slot ← Invokability] If $b_{t_2} = p'$, then $b_{t_1} = p[\overrightarrow{\sigma_{f_{t_1}}}]$. Otherwise, $b_{t_1} = b_{t_2}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[Slot ← Asymmetric Evaluation] By the refined $trans^{-1}$ relation, all temporarily diverging steps of evaluation occur at the terminator of a clone of p', or within a clone of block b'.

Altered Jump First, we cover the case where the $(n'+1)$-th step of evaluation of f' transfers control from block $b_t \in \mathcal{C}(p')$ to block $b_{t_2} \in \mathcal{C}(b')$ by application of rule E-Switch.

$$(p[f/f'] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f'(\overline{v_{t_i}})) \to^{n'}$$

$$(\langle p[f/f'], f_{t_2}, b_t\rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ \texttt{branch}\ \ell_{b'}(\overline{e_i}), \hat{s}_{t_1}) \to$$

$$(\langle p[f/f'], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r'_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_2}), \hat{s}_{t_2})$$

By our strengthened inductive hypothesis, the n-th step of evaluation of f stops in function $f_{t_1} = \mathit{trans}^{-1}(f_{t_2})$ on a terminator that transfers control from block $\mathit{trans}^{-1}(\langle f_{t_1}, b_t\rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. Then, the $(n+1)$-th step of evaluation of f proceeds by application of rule E-Switch. Here, $b_{t_1} \in \mathcal{C}(b)$.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^{n}$$

$$(\langle p, f_{t_1}, \mathit{trans}^{-1}(\langle f_{t_2}, b_t\rangle)\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{branch}\ \ell_b(\overline{e_i}), \mathit{trans}(\hat{s}_{t_2})) \to$$

$$(\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_1}), \mathit{trans}(\hat{s}_{t_2}))$$

Fresh Block Next, we cover the cases where the n'-th step of evaluation of f' is within block b'. If evaluation of f' is not yet at a terminator, then this case is trivial by application of Lemma 6.C.1. Now, consider when the $(n'+1)$-th step of evaluation of f' evaluates the terminator of a clone of block b'. By construction, the terminators of each stream are identical. By application of Lemma 6.B.3, the resulting blocks are again identical. Then, the unmodified streams of both blocks are identical and the resulting register contexts are extended with the same register-value pair from both lemmas, which does not violate the strengthened induction. □

Theorem 9.C.2. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be

evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. By application of Lemma 9.C.1 and Theorem 7.A.1. □

Corollary 9.C.3. The set of paths over block labels from ℓ_0 to $\ell_{b'}$ in $G_{f_{out}}$ for some block $b' \in B$ where $lab(b') = \ell_{b'}$ is the set of such paths in G_f unioned with the set of such paths in G_F with (all combinations of) ℓ_b replaced with $\ell_{b'}$.

Theorem 9.C.4. If f is in canonical form, then f_{out} is in canonical form.

Proof. Block b cannot be a preheader of a loop as it must shave multiple successors and cannot be a header of a loop by assumption. Then, by Corollary 9.C.3 and Theorem 9.C.9, the only changes to loop structures occurs when b' is added to an exit set of some loop l. However, as block b was dedicated to loop l, then block b' just also be dedicated to loop l as block p', its sole predecessor, is a member of l. □

Theorem 9.C.5. If f is in LCSSA form, then f_{out} is in LCSSA form.

Proof. The only uses introduced into the intermediate function f' are those in block b'. However, these uses are identical to the uses of block b and both blocks are placed in the same loop by Theorem 9.C.9. Then, the rest follows by application of Theorem 7.A.7. □

Lemma 9.C.6. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f'$ is well-typed.

Proof. As blocks b and b' have the same set of successors, have the same set of implicit parameters, and define the same registers, they must also have the same set of nonlocal parameters. Then, blocks b and b' have identical typing derivations. References to block b' in block p' are well-typed by application of rule T-Ref as a similar block reference was well-typed in block p and blocks b and b' have identical block parameters and nonlocal block parameters. It then follows that blocks p also have p' identical nonlocal block parameters. □

Theorem 9.C.7. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f_{out}$ is well-typed.

Proof. By application of Lemma 9.C.6 and Theorem 7.A.2. □

Theorem 9.C.8. The unique dominator tree of $G_{f_{out}}$ is D_{out}.

Proof. By Corollary 9.C.3, the blocks that change dominators are block b' (not present in function f), block b (as it is not directly reachable via block p'), and the set of blocks that were previously immediately dominated by block b (as there is now a path reaching it that now travels through block b'). As $pred_{f_{out}}(b') = \{p'\}$, p' immediately dominates block b. It follows that the remaining predecessors of block b have the correct dominator and the immediate dominator of b can be calculated from the nearest common ancestor in the old dominator tree as shown by Alstrup [8]. The same logic can then be repeated to calculate the immediate dominators for the blocks in set $child_{D_f}(b)$. □

Theorem 9.C.9. If f is in canonical form, then F_{out} reconstructed from $(H_{out}, L_{out}, X_{out})$ is the unique loop nesting forest of $G_{f_{out}}$.

Proof. By Corollary 9.C.3, block b' can reach the same blocks as block b and can be reached by the same blocks that can reach block b. Then, blocks b and b' belong in the same strongly connected component and must belong in the same loop. If b is an exit of some loop l, then b' must also be an exit of that loop as the latch of l cannot be reached from block b' while staying in the body set of l as block b cannot either. □

9.D Function Inlining

In this section, we refer to the *inline* transformation (notated below). For specific details (including names of intermediate components), refer to Figure 9.41.

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[\langle f,b,r \leftarrow \texttt{call}(r_c,\overline{e_i})\rangle]{\text{inline}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Lemma 9.D.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f'' can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. We prove by filling in the slots of Template 6.A.1.

[**Slot → Refinement**] We define the predicate *inlined*, as follows, that determines if the current state of evaluation is within a clone of the function later inlined by this transformation. This predicate is true when the element of the evaluation path ρ that transfers control into function f_t, a clone of the target function, comes from the inlined callsite. Because each invocation of a function creates a fresh clone, there is at most one such element.

$$\mathit{inlined}(f_t, \rho) = \begin{cases} \mathit{true} & (\langle b', I_k[\overrightarrow{\sigma_{f_t'}}]\rangle, \mathit{entry}(f_t)) \in \rho \wedge b' \in \mathcal{C}(b) \\ \mathit{false} & \mathit{otherwise} \end{cases}$$

We also define the predicate *beforecall*, as follows, that determines whether or not evaluation has progressed past the inlined call instruction of a given block b_t.

$$\mathit{beforecall}(f_t, b_t, \rho) = (\langle f_t, b_t, I_k[\overrightarrow{\sigma_{f_t}}]\rangle, \_) \in \rho \wedge b_t \in \mathcal{C}(b)$$

We refine the translation of functions by mapping clones of function f and clones of the inlined function to the symmetric clone of function f. All other functions are identical, as it is in the unrefined case.

$$\mathit{trans}_\rho(f_t) = \begin{cases} f''[\overrightarrow{\sigma_{f_t}}] & f_t \in \mathcal{C}(f) \vee \mathit{inlined}(f_t, \rho) \\ f_t & \mathit{otherwise} \end{cases}$$

We refine the translation of blocks by mapping clones of block b to either a clone of block b_1 or a clone of block b_2, depending on how far into the block evaluation has moved. The

terminators of the exit blocks of clones of function $\ddot{f}$ are replaced to branch to block the function's clone of b_2, but only when evaluation is currently within the inlined call.

$$trans_\rho(\langle f_t, b_t \rangle) = \begin{cases} b_1[\overrightarrow{\sigma_{f_t}}] & beforecall(f_t, b_t, \rho) \\ b_2[\overrightarrow{\sigma_{f_t}}] & \neg beforecall(f_t, b_t, \rho) \\ b_t[\texttt{return } e_i/\texttt{branch } \ell_{b'(e_i)}] & b_t \in \mathcal{C}(exit(\ddot{f})) \wedge inlined(f_t, \rho) \\ b_t & otherwise \end{cases}$$

We refine the translation of streams by mapping streams originating from block b to a portion of the stream translated from either s_1 or s_2, depending on how far into the block evaluation has moved. We replace the terminator of clones of the exit blocks of function $\ddot{f}$ with branches back to block b_2, but only when evaluation is currently within the inlined call. For brevity, we decompose the stream s_t into (s'_t, T_t) in the following.

$$trans_\rho(\langle f_t, b_t, (s_t, \hat{s}_t) \rangle) = \begin{cases} \langle s_{1_i}, \ldots \rangle[\overrightarrow{\sigma_{f_t}}], trans_\rho(\hat{s}_t) & b_t \in \mathcal{C}(b) \wedge s' = \langle I_i, \ldots, T \rangle \wedge i \leq k \\ \langle s_{2_{i-k}}, \ldots \rangle[\overrightarrow{\sigma_{f_t}}], trans_\rho(\hat{s}_t) & b_t \in \mathcal{C}(b) \wedge s' = \langle I_i, \ldots, T \rangle \wedge i \not\leq k \\ (s'_t, \texttt{branch } \ell_{b'}(e)), trans_\rho(\hat{s}_t) & inlined(f_t, \rho) \wedge T_t = \texttt{return } e \\ s_t, trans_\rho(\hat{s}_t) & otherwise \end{cases}$$

We refine the translation of return contexts. Whenever the return context's evaluation path ends with an element referencing the inlined instruction, denoted by (b', I'_k) such that $b' \in \mathcal{C}(b)$ and I'_k is the inlined call instruction under register substitution, we simply translate the inner return context and ignore the function, block, register, and remaining instruction stream. In effect, this removes the inlined call from the evaluation history but

leaves everything else alone. Again, for brevity, we decompose the stream s into $(s', \hat{s}_t)$.

$$trans_\rho(\langle f_t, b_t, r_t, s\rangle_{\rho'}) = \begin{cases} trans_{\rho'}(\hat{s}_t) & (c, \_) \in \rho' \\ \langle trans_{\rho'}(f_t), trans_{\rho'}(b_t), r_t, trans_{\rho'}(\langle f_t, b_t, s\rangle)\rangle & otherwise \end{cases}$$

$$\text{where } c = \langle f_t, b_t, I_k[\overrightarrow{\sigma_{f_t}}]\rangle \wedge b_t \in \mathcal{C}(b)$$

We add the following additional relationship between register contexts. Unfortunately, showing register symmetry is not as straightforward as the other transformations in this work as the registers of the inlined function and the source function are not equivalent. Let $\sigma_{\ddot{f}}$ be the substitution set that relates the registers of function $\ddot{f}$ to its clone f' during the transformation (this substitution is unique). We define the following substitution that translates a register from an instance of the non-inlined function into the symmetric register in the inlined function. This is done by first translating the register defined during evaluation of function f into the symmetric register defined in the prototype of $\ddot{f}$, mapping this register into the symmetric register defined in the prototype of f', and finally mapping this prototype register into the symmetric register defined during evaluation of function f''. We use the notation $\langle \ldots, \rho_i\rangle$ to mean the first i elements of ρ so that we can take the proper evaluation path prefix to find the correct function in the parallel evaluation of f''.

$$\sigma = \bigcup_{(\langle f_{t_i}, c\rangle, \_) \in \rho} [\overrightarrow{\sigma_{f_{t_i}}}^{-1} \cdot \sigma_{\ddot{f}} \cdot \overrightarrow{\sigma_{trans_{\langle \ldots, \rho_i\rangle}(f_{t_i})}} \colon f_{t_i} \in \mathcal{C}(\ddot{f}) \wedge inlined(\langle \ldots, \rho_i\rangle)]$$

Now, we define the following relationship stating that registers occurring in γ_1 but not γ_2 must be mappable to a register defined in an instance of the inlined function.

$$(\gamma_1 \cap \gamma_2) \cup (\gamma_1 \setminus \gamma_2)[\sigma] = \gamma_2$$

[Slot → Invokability] If $b_{t_1} \in \mathcal{C}(b)$, then $b_{t_2} = b_1[\overrightarrow{\sigma_{f_{t_1}}}]$. Otherwise, $b_{t_2} = b_{t_1}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[Slot → Asymmetric Evaluation] By the refined *trans* relation, all temporarily diverging steps of evaluation occur in block b or within the function call which is inlined by the transformation.

Inlined Call First, we cover the case where the $(n+1)$-th step of evaluation of f invokes the function to be inlined by application of rule E-Call. Here, $f_{t_1} \in \mathcal{C}(f')$, $b_{t_1} = \mathit{entry}(f_{t_1})$, and $\mathit{dom}(\mathit{param}(b_{t_1})) = \langle \overline{r_i} \rangle$.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n_{\rho_1}$$

$$(\langle p, f_t, b_t \rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ r' \leftarrow \texttt{call}(r_c, \overline{e_i}), s_t, \hat{s}_t) \to_{\rho_2}$$

$$(\langle p, f_{t_1}, b_{t_1} \rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_1}), \langle f_t, b_t, r', (s_t, \hat{s}_t) \rangle)$$

By our strengthened inductive hypothesis, the n'-th step of evaluation of f'' stops in function $f_{t_2} = \mathit{trans}(f_{t_1}) \in \mathcal{C}(f'')$ on a terminator that transfers control from block $\mathit{trans}(\langle f_{t_1}, b_t \rangle)$ to block b_{t_2} by application of rule E-Switch. The predicate $\mathit{inlined}(\rho_2)$ is true as the $(n+1)$-th step of evaluation of f inserted an element including the inlined call instruction onto the evaluation path. Here, $f_{t_2} = \mathit{trans}_{\rho_1}(f_{t_1}) = f''[\overrightarrow{\sigma_{f_{t_1}}}]$, $b_{t_2} = \mathit{trans}_{\rho_2}(\langle f_{t_1}, b_{t_1} \rangle) = b_{t_1}[\overrightarrow{\sigma_{f_{t_1}}}]$, and $\mathit{dom}(\mathit{param}(b_{t_2})) = \langle \overline{r'_i} \rangle$.

$$(p[^f/_{f''}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f''(\overline{v_{t_i}})) \to^{n'}$$

$$(\langle p[^f/_{f''}], f_{t_2}, \mathit{trans}_{\rho_1}(\langle f_{t_1}, b_t \rangle) \rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ \texttt{branch}\ \ell_b(\overline{e_i}), \mathit{trans}_{\rho_1}(\hat{s}_t)) \to$$

$$(\langle p[^f/_{f''}], f_{t_2}, b_{t_2} \rangle \mid \gamma_2[\overline{r'_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_2}), \mathit{trans}_{\rho_1}(\hat{s}_t))$$

The return context $\langle f_t, b_t, r, (s_t, \hat{s}_t) \rangle$ is constructed with the evaluation path ρ_2. This evaluation path ends with the call instruction taken in the $(n+1)$-th step of evaluation of f. Thus, the translation of this return context yields $\mathit{trans}(\hat{s}_t)$ and the resulting return contexts, as well as the instruction streams of blocks b_{t_1} and b_{t_2}, are consistent with the strengthened

induction. Additionally, the following relationships hold with respect to the register substitutions uniquely defining the register spaces of functions f_{t_1} and f_{t_2}.

$$\overrightarrow{\sigma_{f_{t_1}}} \subseteq \{p_i \mapsto r_i \mid p_i \in dom(param(b_{t_1}))\} \qquad \overrightarrow{\sigma_{f_{t_2}}} \subseteq \{p_i \mapsto r'_i \mid p_i \in dom(param(b_{t_2}))\}$$

Then, for each register $\forall r_i \in \{\overline{r_i}\} \cdot \sigma(r_i) = r'_i$ and the register contexts $\gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}]$ and $\gamma_2[\overline{r'_i \mapsto \gamma_2(e_i)}]$ are also consistent with the strengthened induction.

Inlined Return Next, we cover the case where the $(n+1)$-th step of evaluation of f returns from function $f_t \in \mathcal{C}(\ddot{f})$ by rule E-Return. For this case we assume that the call originated from the callsite to be inlined. Here, $f_{t_1} \in \mathcal{C}(f)$, $b_{t_1} = entry(f_{t_1})$, and $dom(param(b_{t_1})) = \langle \overline{r_i} \rangle$.

$$\begin{aligned}(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n_{\rho_1}\ & (\langle p, f_t, b_t \rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{return}\ e, \langle f_{t_1}, b_{t_1}, r', (s_{t_1}, \hat{s}_t) \rangle_{\rho_2}) \\ \to\ & (\langle p, f_{t_1}, b_{t_1} \rangle \mid \gamma_1[r' \leftarrow \gamma_1(e)] \mid \mu' \mid \nu' \mid \Psi';\ s_{t_1}, \hat{s}_t)\end{aligned}$$

By our strengthened inductive hypothesis, the n'-th step of evaluation of f'' stops in function $f_{t_2} = trans(f_{t_1})$ on a terminator that branches to block $b_{t_2} \in \mathcal{C}(b_2)$ by application of rule E-Switch. Here, $f_{t_2} \in \mathcal{C}(f'')$, $b_{t_2} \in \mathcal{C}(b_2)$ and $dom(param(b_{t_2})) = \langle r' \rangle$.

$$\begin{gathered}(p[f/f''] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f''(\overline{v_{t_i}})) \to^{n'} \\ (\langle p[f/f''], f_{t_2}, trans_\rho(\langle f_{t_1}, b_t \rangle) \rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ \texttt{branch}\ \ell_{b'}(e), trans_{\rho_2}(\hat{s}_t)) \to \\ (\langle p[f/f''], f_{t_2}, b_{t_2} \rangle \mid \gamma_2[r \mapsto \gamma_2(e)] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_2}), trans_{\rho_2}(\hat{s}_t))\end{gathered}$$

By assumption, $inlined(\rho_1)$, thus s_{t_1} consists of the instructions occurring after instruction I'_k such that I'_k is the inlined call instruction in block b_{t_1}. Also by assumption, ρ_2 is terminated by an element referencing I'_k, therefore $trans_{\rho_1}(\langle f_{t_1}, b_{t_1}, r', (s_{t_1}, \hat{s}_t) \rangle_{\rho_2}) = trans_{\rho_2}(\hat{s}_t)$. Additionally, as register contexts γ_1 and γ_2 are consistent with the strengthened induction, they remain consistent after the parallel addition of register r' in the last step of evaluation.

$\cdots$ The remaining cases are only described here. The remaining evaluation within a clone of block b or within an inlined block can be proven by application of the lemmas described in Section 6.B. The latter cases requires an symmetric but alternate case analysis due to the translation of registers in the inlined region.

[**Slot $\leftarrow$ Refinement**] We define the predicate *inlined*, as follows, that determines if the given block exists in f'' due to the transformation. This predicate is true when the block label does not exist in the set of block labels that originated from function f (including the labels of blocks b_1 and b_2).

$$\mathit{inlined}(f_t, b_t) = f_t \in \mathcal{C}(f'') \wedge \mathit{lab}(b_t) \notin \{\mathit{lab}(b) \mid b \in \mathit{body}(f)\} \cup \{\ell_{b'}\}$$

We also define the partial function *choose*, as follows, that returns the clone of the currently inlined function in the parallel evaluation of f for a given evaluation context. The result of this function is undefined when the current state of evaluation does not transfer control to or keep control within the set of inlined blocks. In the other cases, control is either transferred to an inlined block (in which case this function returns the target function of the last element of ρ), or is transferred away from an inlined block via a function call (in which case this function returns the source function of the last element of ρ).

$$\mathit{choose}(\rho) = \begin{cases} \mathit{trans}^{-1}(f'_{t_2}) & \mathit{last}(\rho) = (\langle f'_{t_1}, b'_{t_1}, c' \rangle, \langle f'_{t_2}, b'_{t_2} \rangle) \wedge \mathit{inlined}(f'_{t_2}, b'_{t_2}) \\ \mathit{trans}^{-1}(f'_{t_1}) & \mathit{last}(\rho) = (\langle f'_{t_1}, b'_{t_1}, c' \rangle, \langle f'_{t_2}, b'_{t_2} \rangle) \wedge \mathit{inlined}(f'_{t_1}, b'_{t_1}) \end{cases}$$

We refine the translation of functions by mapping clones of function f'' to either a clone of function f or a clone of function $\ddot{f}$, depending on whether or not evaluation is currently within an inlined block. In the following, we use b_t to mean the block in the *same step of*

evaluation of function f'' as when the $trans^{-1}$ relation is applied.

$$trans^{-1}(f_t) = \begin{cases} f[\overrightarrow{\sigma_{f_t}}] & f_t \in \mathcal{C}(f'') \wedge \neg inlined(f_t, b_t) \\ \ddot{f}[\overrightarrow{\sigma_{f_t}}] & inlined(f_t, b_t) \\ f_t & otherwise \end{cases}$$

We refine the translation of blocks by mapping clones of blocks b_1 and b_2 back to a clone of block b. Additionally, the terminators of inlined blocks that branch back to block b_2 are replaced by a return of the same value.

$$trans^{-1}(\langle f_t, b_t \rangle) = \begin{cases} b[\overrightarrow{\sigma_{f_t}}] & \in \mathcal{C}(b_1) \cup \mathcal{C}(b_2) \\ b_t\big[\texttt{branch } \ell_{b'}(e)/\texttt{return } e\big] & inlined(f_t, b_t) \\ b_t & otherwise \end{cases}$$

We refine the translation of streams by replacing the terminator of clones of block b_1 with the inlined call instruction and append the stream of the clone of block b_2 in the same function. Additionally, we replace the terminator of clones of inlined blocks that branch back to block b_2 with a return of the same value. For brevity, we decompose the stream s_t into (s'_t, T_t) in the following.

$$trans_\rho^{-1}(\langle f_t, b_t, (s_t, \hat{s}_t) \rangle) = \begin{cases} s'_t \cup \langle I_k, I_{k+1}, T \rangle[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(b_1) \\ (s'_t, \texttt{return } e, trans_\rho^{-1}(\hat{s}_t) & inlined(f_t, b_t) \wedge T_t = \texttt{branch } \ell_{b'}(e)) \\ s_t, trans_\rho^{-1}(\hat{s}_t) & otherwise \end{cases}$$

We refine the translation of return contexts by adding an additional layer when the state of evaluation is within the inlined region. This layer points to the instruction stream that occurs after the inlined callsite in a clone of block b. In effect, this adds a stack frame entry of the inlined region. For legibility, we define the $trans'$ relation, which translates return

contexts and the empty return context in the default way.

$$trans_{\rho}^{-1}(\hat{s}) = \begin{cases} \langle f'_t, block_{f'_t}(\ell_b), r[\overrightarrow{\sigma_{f'_t}}], (s_2[\overrightarrow{\sigma_{f'_t}}], trans'(\hat{s}))\rangle & f_t = trans^{-1}(choose(\rho)) \\ trans'(\hat{s}) & otherwise \end{cases}$$

$$trans'(\hat{s}) = \begin{cases} \langle trans^{-1}(f_t), trans^{-1}(b_t), r_t, trans_{\rho'}^{-1}(\langle f_t, b_t, s\rangle)\rangle & \hat{s} = \langle f_t, b_t, r_t, s\rangle_{\rho'} \\ \epsilon & \hat{s} = \epsilon \end{cases}$$

We use the same relationship for register contexts as the forward case (with the same abuses of notation).

[**Slot ← Invokability**] If $b_{t_2} \in \mathcal{C}(b_1)$, then $b_{t_1} = b[\overrightarrow{\sigma_{f_{t_1}}}]$. Otherwise, $b_{t_2} = b_{t_1}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[**Slot ← Asymmetric Evaluation**] By the refined $trans^{-1}$ relation, all temporarily diverging steps of evaluation occur in blocks b_1 and b_2 or within the function call which is inlined by the transformation.

Inlined Call First, we cover the case where the $(n'+1)$-th step of evaluation of f'' transfers control into the inlined region by application of rule E-Switch. Here, $b_{t_2} \in \mathcal{C}(entry(f'))$, and $dom(param(b_{t_2})) = \langle \overline{r_i} \rangle$.

$$(p[f/f''] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f''(\overline{v_{t_i}})) \rightarrow_{\rho'_1}^{n'}$$

$$(\langle p[f/f''], f_{t_2}, b_t)\rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ \texttt{branch}\ \ell_b(\overline{e_i}), \hat{s}_t) \rightarrow_{\rho'_2}$$

$$(\langle p[f/f''], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_2}), \hat{s}_t)$$

By our strengthened inductive hypothesis, the n-th step of evaluation of f stops in function $f_t = trans^{-1}(f_{t_2})$ on the inlined callsite of block $trans^{-1}(b_t) \in \mathcal{C}(b)$. Evaluation proceeds by

application of rule E-Call. Here, $f_{t_1} \in \mathcal{C}(f')$, $b_{t_1} = \mathit{entry}(f_{t_1})$, and $\mathit{dom}(\mathit{param}(b_{t_1})) = \langle \overline{r'_i} \rangle$.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n$$

$$(\langle p, f_t, \mathit{trans}^{-1}(b_t) \rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ r' \leftarrow \texttt{call}(r_c, \overline{e_i}), s_t, \mathit{trans}^{-1}_{\rho'_1}(\hat{s}_t)) \to$$

$$(\langle p, f_{t_1}, b_{t_1} \rangle \mid \gamma_1[\overline{r'_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_1}), \langle f_t, b_t, r', (s_t, \mathit{trans}^{-1}_{\rho'_1}(\hat{s}_t)) \rangle)$$

The predicate $\mathit{inlined}(f_{t_1}, f_{t_2})$ is true and the blocks and their unmodified streams are consistent with the strengthened induction. Because the last element of ρ'_2 is a switch into the inlined block, the translation of the resulting return context must add a layer that will assign the register symmetric to r and resume the instruction stream symmetric to s_2 – these are precisely r' and s_t with respect to function f_t. Additionally, register contexts γ_1 and γ_2 are consistent with the strengthened induction, and the following relationships hold with respect to the register substitutions uniquely defining the register spaces of functions f_{t_1} and f_{t_2}.

$$\overrightarrow{\sigma_{f_{t_1}}} \subseteq \{p_i \mapsto r'_i \mid p_i \in \mathit{dom}(\mathit{param}(b_{t_1}))\} \qquad \overrightarrow{\sigma_{f_{t_2}}} \subseteq \{p_i \mapsto r_i \mid p_i \in \mathit{dom}(\mathit{param}(b_{t_2}))\}$$

Then, for each register $\forall r_i \in \{\overline{r_i}\} \cdot \sigma(r_i) = r'_i$ and the register contexts $\gamma_1[\overline{r'_i \mapsto \gamma_1(e_i)}]$ and $\gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}]$ are also consistent with the strengthened induction.

Inlined Return Next, we cover the case where the $(n'+1)$-th step of evaluation of f'' transfers control back from the inlined region by rule E-Switch. Here, $f_{t_1} \in \mathcal{C}(f'')$, $b_{t_2} \in \mathcal{C}(b_2)$, and $\mathit{dom}(\mathit{param}(b_{t_2})) = \langle r' \rangle$. Notice that the last element of ρ'_1 must be a switch or a return back to block b_t.

$$(p[f/f''] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f''(\overline{v_{t_i}})) \to^{n'}_{\rho'_1}$$

$$(\langle p[f/f''], f_{t_2}, b_t \rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ \texttt{branch}\ \ell_{b'}(e), \hat{s}_{t_2}) \to_{\rho'_2}$$

$$(\langle p[f/f''], f_{t_2}, b_{t_2} \rangle \mid \gamma_2[r' \mapsto \gamma_2(e)] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_2}), \hat{s}_{t_2})$$

By our strengthened inductive hypothesis, the n-th step of evaluation of f stops in function $f_{t_1} = \mathit{trans}^{-1}(f_{t_2})$ on a return back to the inlined callsite by application of rule E-Return. Then, as the predicate $\mathit{inlined}(f_{t_2}, b_t)$ is true, $\mathit{trans}^{-1}_{\rho_1'}(\hat{s}_{t_2}) = (f_{t_1}, b[\overrightarrow{\sigma_{f_{t_1}}}], r', s_2[\overrightarrow{\sigma_{f_{t_1}}}], \hat{s}_{t_1})$.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n$$

$$(\langle p, f_t, \mathit{trans}^{-1}(\langle f_{t_2}, b_t\rangle)\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{return}\ e, \mathit{trans}^{-1}_{\rho_1'}(\hat{s}_{t_2})) \to$$

$$(\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1[r' \leftarrow \gamma_1(e)] \mid \mu' \mid \nu' \mid \Psi';\ s_2[\overrightarrow{\sigma_{f_{t_1}}}], \hat{s}_{t_1})$$

The return block is a clone of block b and the instruction stream consists of the instruction that occur in block b after the call (this is precisely the stream s_2). The resulting return contexts are trivially consistent with the strengthened induction with respect to the evaluation path ρ_2', as $\hat{s}_{t_1}$ is the exact translation of $\hat{s}_{t_2}$ under the previous evaluation context. Additionally, as register contexts γ_1 and γ_2 are consistent with the strengthened induction, they remain consistent after the parallel addition of register r' in the last step of evaluation.

$\cdots$ As in the forward case, we omit the remaining evaluation within a clone of block b_1, block b_2, or within the inlined region. □

Theorem 9.D.2. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. By Lemma 9.D.1 and Theorem 8.B.8. □

Corollary 9.D.3. If there is at least one exit block of function f', then the set of paths over block labels from ℓ_0 to $\ell_{b'}$ in $G_{f_{out}}$ for some block $b' \in B$ where $\mathit{lab}(b') = \ell_{b'}$ can be constructed by replacing subpaths in the set of such paths in G_f as follows: replace all edges matching the form $(\ell_b, \mathit{lab}(c))$ where $c \in \mathit{succ}_f(b)$ with a subpath from the family of

subpaths $\langle \ell_b, P_{f'}, \ell_{b'}, lab(c) \rangle$. Here, $P_{f'}$ expands to all complete paths through function f' from the entry block to some exit block of f'. If there are no exit blocks of function f', then the set of paths over block labels from ℓ_0 to ℓ_b in $G_{f_{out}}$ can be extended by the set of infinite paths of f' from the entry block to any reachable block in f'.

Before we can state the maintenance properties of the *inline* procedure, we need to define several lemmas to show that the approximate loop nesting forest constructed directly before the *eject* procedure is in the form assumed by ejection. First, we show that the approximate loop nesting forest contains the correct number of loops and each loop has the correct header. In the following lemmas, we refer to function f_i and the deconstructed loop triple (H_i, L_i, X_i) as the inputs of the *eject* procedure. These are equivalent to function f' and the triple (H', L', X') in Figure 8.23 but are renamed to reduce ambiguity with the names of Figure 9.42.

Lemma 9.D.4. If f and f' are in canonical form, then $dom(H_i) = \{lab(l') \mid l' \in F_{f_i}\}$.

Proof. The blocks and edges introduced to function f include the entire structure of function f' as well as the forward edges (b_1, b') and (e, b_2) for each exit e of f'. The forest $F_{f'}$ contains a loop for every backedge introduced to the graph. The remainder of the proof then proceeds identically to Lemma 8.B.3. □

Now we show that the approximate loop nesting forest forms a subset lattice (the nesting structure of loops is somewhat correct – an inner loop with too many blocks in its body set may be nested under a descendant of its true parent).

Lemma 9.D.5. The loop nesting forest reconstructed from (H_i, L_i, X_i) forms a subset lattice.

Proof. All blocks of f' are added to the body set of loop l as well as the body set of its ancestors. As loop nesting forests L_{F_f} and $L_{F_{f'}}$ form subset lattices by definition, the

addition of blocks to the union to a set and all of its supersets do not break this property. The remainder of the proof then proceeds identically to Lemma 8.B.4. □

Now we show that every loop in the approximate loop nesting forest has a body set containing a superset of the correct blocks. This is stated formally in Lemma 9.D.6. This property ensures that blocks must only be *ejected* from the loop to correct the body set, and no additional blocks can be brought in from the outside.

Lemma 9.D.6. If f and f' are in canonical form, then $body_{f_i}(l') \subseteq S$ for each $l' \in F_i$ where S is the set of blocks composing the body of the loop with header $header_{f_i}(l')$ reconstructed from H_i and L_i.

Proof. First, no internal edges of function f' are altered and the duplicated loop structure must encode the correct body sets. Second, no block introduced to a graph can belong to a strongly connected component that does not also contain block b_1 as each such block is dominated by b_1. The remainder of the proof then proceeds identically to Lemma 8.B.5. □

Now we show that the exits of each loop are correct with respect to the claimed body set. Naturally, this implies the exit sets for loops with an accurate body set are correct.

Lemma 9.D.7. If f and f' are in canonical form, then the exit set of a loop l' is consistent with its body set for every loop l' in the loop nesting forest reconstructed from (H_i, L_i, X_i).

Proof. All blocks of f' are added to the body set of loop l as well as the body set of its ancestors. Any new edge leaving a block of f' targets block b_2, which is also contained in loop l. Then, no new edges are introduced that cross a boundary of a claimed loop body. The remainder of the proof then proceeds identically to Lemma 8.B.6. □

Now we show that all inaccurate loops can be found on a single path in the claimed loop nesting forest.

Lemma 9.D.8. If f and f' are in canonical form, then all loops with an inaccurate body set exists on the same ancestor path in the loop nesting forest reconstructed from (H_i, L_i, X_i).

Proof. By Corollary 9.D.3, all strongly connected components altered by the introduction of the subgraph f' also contain block b. Then, the only loops that contain superfluous blocks in their body are ancestor of $loop_f(b)$. The remainder of the proof then proceeds identically to Lemma 9.D.8. □

Finally, we can state the maintenance theorems for the *inline* procedure.

Theorem 9.D.9. If f and f' are in canonical form, then f_{out} is in canonical form.

Proof. All loops of function f' are in canonical form and the introduction of the subgraph f' cannot undedicate the preheader or exits of the loops containing block b. Then, the remainder of the proof then proceeds identically to Lemma 8.B.10. □

Theorem 9.D.10. If f and f' are in LCSSA form, then f_{out} is in LCSSA form.

Proof. The registers of functions f and f' are disjoint, thus there are no uses of any register defined in f' are used outside of the inlined subgraph. All movement of uses are then caused by an application of the *eject* procedure, and the remainder of the proof then proceeds identically to Lemma 8.B.11. □

Theorem 9.D.11. If $p \mid f$ and $p \mid f'$ are well-typed and f and f' are in SSA form, then $p \mid f_{out}$ is well-typed.

Proof. The implicit block parameter and nonlocal block parameter sets of the entry of function f' are empty. Then, if block b_2 remains reachable after the introduction of the subgraph f', then $nonlocal_{f_{out}}(b_1) = nonlocal_f(b)$ and $nonlocal_{f_{out}}(b_1) = param(b_1)$ otherwise. In the former case, nonlocal parameters are only removed from the set and cannot adversely affect

previously well-typed derivations. In either case, the definitions on the path to b_1 have not been affected and the nonlocal parameter set for the entry of f remains empty. Additionally, it is worth nothing that if block b_2 remains reachable, every block of f' gains the nonlocals of block b_2 as no definition of f' can kill a use required by a block of f.

Now, we show that the altered blocks b_1, b_2, and the exits of f' are also well-typed. Let $\Gamma_0^0 = \mathit{param}(b_1), \mathit{nonlocal}_f(b_1)$. First, we show that $(p\ f_{out} \mid \Gamma_0^0) \vdash \mathit{stream}(b_1)$. Here, $\mathit{stream}(b_1) = s_1$ and the derivation begins in identical fashion as the first k steps of $\mathit{stream}(b)$. Next, we show that $(p\ f_{out} \mid \Gamma_k^0) \vdash \texttt{branch}\ \ell_{b'}(\overline{e_i})$. By inversion of rule T-Call on the stream of b, $\Gamma_k^0 \vdash e_i \colon t_i$. Here, each type t_i matches a symmetric parameter of block b'. As stated above, block b' has no nonlocal block parameters. Then, derivation proceeds by application of rules T-Switch and T-Ref and block b_1 is well-typed.

Let $\Gamma_0^1 = \mathit{param}(b_2), \mathit{nonlocal}_f(b_2)$. Next, we show that $(p\ f_{out} \mid \Gamma_0^1) \vdash \mathit{stream}(b_2)$. Here, $\mathit{param}(b_2) = \langle r \colon t' \rangle$, which is the same register and yielded by the call instruction of b, $\mathit{nonlocal}_{f_{out}}(b_2) \setminus \mathit{def}(b) = \mathit{nonlocal}_f(b)$, and $\mathit{stream}(b_2) = s_2$. Then, Γ_0^1 is equivalent to the register typing context at the k-th step of evaluation of $\mathit{stream}(b)$ and the derivation of both streams proceed in identical fashion.

Lastly, we show that each modified exit of f' is well-typed. Let block e be such an exit. The initial typing derivation of the original and modified blocks proceed identically until the terminator. In the derivation of the original block, $(p\ f_{out} \mid \Gamma_2) \vdash \texttt{return}\ e_i$ is well-typed by inversion of rule T-Return such that $\Gamma^2 \vdash e_i \colon t'$. Here, t' is the target type of block b_2's only parameter. As stated above, the set of nonlocals of each block in b is a superset of the nonlocals of block b_2. Then, $(p\ f_{out} \mid \Gamma_2) \vdash \texttt{branch}\ \ell_{b_2}(e_i)$ is well-typed by application of rule E-Switch. □

Theorem 9.D.12. The unique dominator tree of $G_{f_{out}}$ is D'.

Proof. As there is no path to any block originating from function f' that does not pass

through $entry(f')$, no immediate dominator encoded by the subtree $D_{f'}$ changes with respect to function f_{out}. By Corollary 9.D.3, the blocks that change dominators are blocks b_1 and b_2 (not present in function f), and the set of blocks that were previously immediately dominated by block b (as their new predecessor is now block b_2). As block b_1 takes takes the place of block b in path through the graph, it also takes it place in the dominator tree. As block b_1 and block b have the same label, no mapping is necessary (and this relationship is already encoded in D_f). Next, all paths to the inlined call function must pass through block b_1 thus, block b_1 immediately dominates block $entry(f')$ (and, transitively, dominates all blocks in f'). Lastly, we show the domination relationships around block b_2. First suppose that the inlined function contains at least one exit block. By Corollary 9.D.3, the set of all paths to block b_2 must travel through an exit of the inlined function f' and the immediate dominator of b_2 is the nearest common ancestor of all such exits in $D_{f'}$. Additionally, block $c \in child_{D_f}(b)$ must pass through block b_2 (after passing through its original dominator) by Corollary 9.D.3 and block b_2 dominates block c. Now, suppose that the inlined function contains no return instructions (and consists entirely of a series of unbreakable loops). Then, block b_2 is not reachable, nor are any of the blocks dominated by block b. These blocks are removed from the function, as well as D_{out}, by application of the *post delete edge* procedure. □

Theorem 9.D.13. If f is in canonical form, then F_{out} reconstructed from $(H_{out}, L_{out}, X_{out})$ is the unique loop nesting forest of $G_{f_{out}}$.

Proof. By application of Lemma 9.D.4 through Lemma 9.D.8 and Theorem 8.A.9. □

9.E Loop Unswitching

In this section, we refer to the *unswitch* transformation (notated below). For specific details (including names of intermediate components), refer to Figure 9.52.

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l,\langle f,b\rangle]{\text{unswitch}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Lemma 9.E.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f' can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. We prove by filling in the slots of Template 6.A.1.

[**Slot → Refinement**] We define the predicate *version*, as follows, that selects the version of the loop l that would be entered based on the value of $r \in \gamma_1$. This relates directly to the branching behavior of the new preheader p'. In the following, we use γ_1 to mean the register context in the *same step of evaluation* of function f as predicate is applied.

$$version(\gamma_1) = \begin{cases} l'_i & \gamma_1(r) = cv_i \\ l & otherwise \end{cases}$$

We refine the translation of blocks by mapping clones of a block in l to the symmetric clone of a block in the duplicate loop. We refine the translation of streams by replacing the terminator of clones of block p with the terminator that unconditionally switches to the

chosen default block. For brevity, we decompose the stream s_t into (s'_t, T_t).

$$trans(\langle f_t, b_t \rangle) = \begin{cases} p'[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(p) \\ b'[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(b) \wedge version(\gamma_1) = l \\ b'_i[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(b) \wedge version(\gamma_1) = l'_i \\ block_{f'_k}(\sigma_{L_i}(lab(b_t))[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(body_f(l)) \wedge version(\gamma_1) = l'_i \\ b_t & otherwise \end{cases}$$

$$trans(\langle f_t, b_t, (s_t, \hat{s}_t) \rangle) = \begin{cases} (s'_t, T'[\overrightarrow{\sigma_{f_t}}]), trans(\hat{s}_t) & b_t \in \mathcal{C}(p) \\ (s'_t, T'_i[\overrightarrow{\sigma_{f_t}}]), trans(\hat{s}_t) & b_t \in \mathcal{C}(b) \wedge version(\gamma_1) = l'_i \\ (s'_t, T_t[\sigma_{L_i}]), trans(\hat{s}_t) & b_t \in \mathcal{C}(body_f(l) \setminus \{b\}) \wedge version(\gamma_1) = l'_i \\ s_t, trans(\hat{s}_t) & otherwise \end{cases}$$

We add the following additional relationship between register contexts.

$$\gamma_1 = \gamma_2$$

[Slot → Invokability] The entry block of f cannot be in loop l as the loop is assumed to have a dedicated preheader. If $b_{t_1} = p$, then $b_{t_2} = p'[\overrightarrow{\sigma_{f_{t_1}}}]$. Otherwise, $b_{t_2} = b_{t_1}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[Slot → Asymmetric Evaluation] By the refined *trans* relation, all temporarily diverging steps of evaluation occur at the terminator of a clone of p, a clone of block b or its duplicates, or within the body of l when $\gamma_1(r) \neq \texttt{unit}$.

Preheader First, we cover the case where the $(n+1)$-th step of evaluation of f transfers control from block $b_t \in \mathcal{C}(p)$ by application of rule E-Switch. Here, $b_{t_1} \in \mathcal{C}(\mathit{header}(l))$.

$$\begin{aligned}(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n\ & (\langle p, f_{t_1}, b_t\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{branch}\ \ell_h(\overline{e_i}), \hat{s}_{t_1}) \\ \to\ & (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_1}), \hat{s}_{t_1})\end{aligned}$$

By our strengthened inductive hypothesis, the n'-th step of evaluation of f' stops in function $f_{t_2} = \mathit{trans}(f_{t_1})$ on a terminator that transfers control from block $\mathit{trans}(\langle f_{t_1}, b_t\rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. The $(n'+1)$-th step of evaluation of f' proceeds by application of rule E-Switch. Here, b_{t_2} corresponds to a clone of the header of the version of loop l targeted by the switch with the constant value $\gamma_1(r)$.

$$\begin{gathered}(p[f/f'] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f'(\overline{v_{t_i}})) \to^{n'} \\ (\langle p[f/f'], f_{t_2}, \mathit{trans}(\langle f_{t_1}, b_t\rangle)\rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ T'[\overrightarrow{\sigma_{f_{t_1}}}], \mathit{trans}(\hat{s}_{t_1})) \to \\ (\langle p[f/f'], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_2}), \mathit{trans}(\hat{s}_{t_1}))\end{gathered}$$

Switched Block Next, we cover the case where the $(n+1)$-th step of evaluation of f transfers control from block $b_t \in \mathcal{C}(b)$ by application of rule E-Switch.

$$\begin{aligned}(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n\ & (\langle p, f_{t_1}, b_t\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{switch}\ r\ \overline{cv_i \mapsto \mathit{ref}_i}\ \mathit{ref}_d, \hat{s}_{t_1}) \\ \to\ & (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_1}), \hat{s}_{t_1})\end{aligned}$$

By our strengthened inductive hypothesis, the n'-th step of evaluation of f' stops in function $f_{t_2} = \mathit{trans}(f_{t_1})$ on a terminator that transfers control from block $\mathit{trans}(\langle f_{t_1}, b_t\rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. The $(n'+1)$-th step of evaluation of f' proceeds by application of rule E-Switch. First, suppose that ref_d was the block reference taken duration evaluation of f. Then, $\gamma_1(r) \neq cv_i$ for any switch

case in the terminator of b, $b_t \in \mathcal{C}(b')$, and $b_{t_2} = \mathcal{C}(b_{t_1})$.

$$(p[f/f'] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f'(\overline{v_{t_i}})) \to^{n'}$$
$$(\langle p[f/f'], f_{t_2}, \mathit{trans}(\langle f_{t_1}, b_t\rangle)\rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ \texttt{branch}\ \overline{cv_i \mapsto \mathit{ref}_i}\ \mathit{ref}_d, \mathit{trans}(\hat{s}_{t_1})) \to$$
$$(\langle p[f/f'], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_2}), \mathit{trans}(\hat{s}_{t_1}))$$

Now, suppose that ref_j was the block reference taken duration evaluation of f. Then, $\gamma_1(r) = cv_j$, $b_t \in \mathcal{C}(b'_j)$, and $b_{t_2} \in \mathcal{C}(\mathit{block}_{f'_k}(\sigma_{L_j}(\mathit{lab}(b_{t_1}))))$.

$$(p[f/f'] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f'(\overline{v_{t_i}})) \to^{n'}$$
$$(\langle p[f/f'], f_{t_2}, \mathit{trans}(\langle f_{t_1}, b_t\rangle)\rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ \texttt{switch}\ cv_j\ \overline{cv_i \mapsto \mathit{ref}_i}\ \mathit{ref}_d, \mathit{trans}(\hat{s}_{t_1})) \to$$
$$(\langle p[f/f'], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_2}), \mathit{trans}(\hat{s}_{t_1}))$$

Loop Body Next, we cover the cases where the n-th step of evaluation of f is within loop l and $\mathit{version}(\gamma_1) = l_i$, where γ_1 is the register context at step n of evaluation of f. If evaluation of f is not yet at a terminator, then this case is trivial by application of Lemma 6.C.1. Now, consider when the $(n+1)$-th step of evaluation of f evaluates a terminator (distinct from the switched terminator in a clone of block b). This terminator is necessarily a `switch` as evaluation is currently within a loop. By our strengthened inductive hypothesis, the n'-th step of evaluation of f' stops at a state with a symmetric function, block, terminator, and register context. Let T_1 and T_2 be the terminator in the evaluation of functions f and f', respectively. Here, $T_2 = T_1[\sigma_{L_i}]$. By application of Lemma 6.B.3, the resulting blocks are either the same or mappable through σ_{L_i}. The blocks and their unmodified streams are consistent with the strengthened induction. Additionally, the resulting register contexts are extended with the same set of register-value pairs, which does not violate the strengthened induction.

[Slot ← Refinement] We refine the translation of blocks and streams by collapsing all blocks (and block references) that belong to the duplicated loop to the symmetric block (and reference to the symmetric block) in the original loop. Let $\sigma_L^{-1} = \bigcup \sigma_{L_i}^{-1}$, which maps a label of a duplicated block back to its source (notice that the codomain of each σ_{L_i} is unique). Let $B_D = (\mathcal{C}(body_{f'}(l')) \cup \mathcal{C}(latch_{f'}(l))) \setminus \mathcal{C}(\{b', \overline{b'_j}\})$. This denotes the set of blocks that reference a duplicated block in function f'. For brevity, we decompose the stream s_t into $(s'_t, \hat{s}_t)$.

$$trans(\langle f_t, b_t \rangle) = \begin{cases} p[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(p') \\ b[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(b') \cup \mathcal{C}(b'_j) \\ block_f(\sigma_L^{-1}(lab(b_t))[\overrightarrow{\sigma_{f_t}}] & b_t \in B_D \\ b_t & otherwise \end{cases}$$

$$trans(\langle f_t, b_t, (s_t, \hat{s}_t) \rangle) = \begin{cases} (s'_t, latch(p)[\overrightarrow{\sigma_{f_t}}]), trans(\hat{s}_t) & b_t \in \mathcal{C}(p') \\ (s'_t, latch(b)[\overrightarrow{\sigma_{f_t}}]), trans(\hat{s}_t) & b_t \in \mathcal{C}(b') \\ (s'_t, T_t[\sigma_L^{-1}]), trans(\hat{s}_t) & b_t \in B_D \\ s_t, trans(\hat{s}_t) & otherwise \end{cases}$$

We add the following additional relationship between register contexts.

$$\gamma_2 = \gamma_1$$

[Slot ← Invokability] If $b_{t_2} = p'$, then $b_{t_1} = p[\overrightarrow{\sigma_{f_{t_2}}}]$. Otherwise, $b_{t_1} = b_{t_2}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[Slot ← Asymmetric Evaluation] By the refined $trans^{-1}$ relation, all temporarily diverging steps of evaluation occur at the terminator of a clone of p, a clone of block b', a clone

of some block b'_j, or within the body of some loop l_i.

Preheader First, we cover the case where the $(n'+1)$-th step of evaluation of f' transfers control from block $b_t \in \mathcal{C}(p')$ by application of rule E-Switch.

$$(p[f/f'] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f'(\overline{v_{t_i}})) \to^{n'}$$

$$(\langle p[f/f'], f_{t_2}, b_t\rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ T'[\overrightarrow{\sigma_{f_{t_1}}}], \hat{s}_{t_2}) \to$$

$$(\langle p[f/f'], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_2}), \hat{s}_{t_2})$$

By our strengthened inductive hypothesis, the n-th step of evaluation of f stops in function $f_{t_1} = \mathit{trans}^{-1}(f_{t_2})$ on a terminator that transfers control from block $\mathit{trans}^{-1}(\langle f_{t_2}, b_t\rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. The $(n+1)$-th step of evaluation of f proceeds by application of rule E-Switch such that $\mathit{trans}^{-1}(\langle f_{t_2}, b_t\rangle) \in \mathcal{C}(p)$ and $\sigma_L^{-1}(\mathit{lab}(b_{t_2})) = \ell_h$ by translation.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^{n}$$

$$(\langle p, f_{t_1}, \mathit{trans}^{-1}(\langle f_{t_2}, b_t\rangle)\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{branch}\ \ell_h(\overline{e_i}), \mathit{trans}^{-1}(\hat{s}_{t_2})) \to$$

$$(\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_1}), \mathit{trans}^{-1}(\hat{s}_{t_2}))$$

Switched Block Next, we cover the case where the $(n'+1)$-th step of evaluation of f transfers control from block $b_t \in \mathcal{C}(b') \cup \mathcal{C}(b'_j)$ by application of rule E-Switch. Notice that $cv = \texttt{unit}$ when $b_t \in \mathcal{C}(b')$.

$$(p[f/f'] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f'(\overline{v_{t_i}})) \to^{n'}$$

$$(\langle p[f/f'], f_{t_2}, b_t\rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ \texttt{switch}\ cv\ \overline{cv_i \mapsto \mathit{ref}_i}\ \mathit{ref}_d, \hat{s}_{t_2}) \to$$

$$(\langle p[f/f'], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_2}), \hat{s}_{t_2})$$

By our strengthened inductive hypothesis, the n-th step of evaluation of f stops in function $f_{t_1} = \mathit{trans}^{-1}(f_{t_2})$ on a terminator that transfers control from block $\mathit{trans}^{-1}(\langle f_{t_2}, b_t\rangle) \in \mathcal{C}(b)$,

as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. The $(n+1)$-th step of evaluation of f proceeds by application of rule E-SWITCH.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n$$

$$(\langle p, f_{t_1}, \mathit{trans}^{-1}(\langle f_{t_2}, b_t\rangle)\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ \texttt{switch}\ r\ \overline{cv_i \mapsto \mathit{ref}_i}\ \mathit{ref}_d, \mathit{trans}^{-1}(\hat{s}_{t_2})) \to$$

$$(\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_1}), \mathit{trans}^{-1}(\hat{s}_{t_2}))$$

First, we consider the case where b_t is a member of loop l. As all paths into this loop must pass through the modified preheader and r is not assigned within loop l, it must be the case that $\gamma_1(r) = \gamma_2(r) = \texttt{unit}$. Then, the switch value and switch cases are the same and symmetric block references are taken. Next, we consider the case where b_t is a member of some loop l'_j. Again, by the path through the modified preheader, it must be the case that $\gamma_1(r) = \gamma_2(r)$ and a symmetric block reference is taken. Then, the resulting blocks are either the same or mappable through σ_L^{-1}. The blocks and their unmodified streams are consistent with the strengthened induction.

Loop Body Next, we cover the cases where the n'-th step of evaluation of f' is within loop l_i. If evaluation of f' is not yet at a terminator, then this case is trivial by application of Lemma 6.C.1. Now, consider when the $(n'+1)$-th step of evaluation of f' evaluates a terminator (distinct from the switched terminator in a clone of block b). This terminator is necessarily a `switch` as evaluation is currently within a loop. By our strengthened inductive hypothesis, the n-th step of evaluation of f stops at a state with a symmetric function, block, terminator, and register context. Let T_1 and T_2 be the terminator in the evaluation of functions f and f', respectively. Here, $T_1 = T_2[\sigma_{L_i}^{-1}]$. By application of Lemma 6.B.3, the resulting blocks are either the same or mappable through $\sigma_{L_i}^{-1}$. The blocks and their unmodified streams are consistent with the strengthened induction. Additionally, the resulting register contexts are extended with the same set of register-value pairs, which does

not violate the strengthened induction. □

Theorem 9.E.2. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. By application of 9.E.1, Theorem 7.E.1, Theorem 7.F.1, and Theorem 7.A.1. □

Corollary 9.E.3. The set of paths over block labels from ℓ_0 to $\ell_{b'}$ in $G_{f'}$ for some block $b' \in B$ where $lab(b') = \ell_{b'}$ can be constructed by replacing each minimal subpath of the form $p = \langle lab(header(l)), \ldots, \ell_e \rangle$ where $\ell_e \in exit(l)$ by the path $p[\sigma_{L_i}]$ for some σ_{L_i} in the set of such paths in G_f.

Theorem 9.E.4. If f is in canonical form, then f_{out} is in canonical form.

Proof. By Corollary 9.E.3, all additional paths that can break canonical form occur at the boundaries of the duplicated loops. The preheaders and exits of the original loop and each duplicated loop are subsequently dedicated. Therefore, no canonical form violations can remain. □

Theorem 9.E.5. If f is in LCSSA form, then f_{out} is in LCSSA form.

Proof. As loop l contains all uses of any definition in l, each duplication of l has the same property. The application of *rewrite uses* then effectively maps the set of registers defined by $def(l)$ to a set of fresh registers for each duplicated loop. This is obvious as the reaching definition of each such use is the unique definition of the used register. Clearly, no additional violating uses are introduced to the function. □

Lemma 9.E.6. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f'$ is well-typed.

Proof. Each block $b \in body_f(l)$ and its duplicates have the same set of register uses. By Corollary 9.E.3, the set of register uses reachable from block b and the set of register uses reachable from one of its duplicates are equivalent. Further, each block b and its duplicates also define the same registers and thus have the same implicit block parameters and nonlocal block parameters. Clearly, blocks b and its duplicates have identical typing derivations. Let *ref* be the block reference in block p. As *ref* is well-typed, all references in block p' are similarly well-typed as each reference has identical arguments, and each target block has the same set of parameters and nonlocal implicit parameters. As all predecessors of block p' have the same set of nonlocal block parameters, it follows that block p and p' have identical nonlocal block parameters. □

Theorem 9.E.7. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f_{out}$ is well-typed.

Proof. By application of 9.E.6, Theorem 7.E.4, Theorem 7.F.3, and Theorem 7.A.2. □

In order for this transformation to maintain domination, we add the additional requirement that the exits of the input function are dedicated. While this is not an explicit requirement for maintenance in the other transformations, it is an implicit assumption in the design of the optimizer described in this work.

Lemma 9.E.8. If all exits of f are dedicated, then the unique dominator tree of $G_{f'}$ is D''.

Proof. As there is no path to any block originating from a duplicate loop l'_k that does not pass through block $header(l')$, no immediate dominator encoded by the subtree D'_k changes with respect to function f'. By Corollary 9.E.3, the blocks that change dominators are the headers of the duplicate loops (not present in f) and the set of blocks not in l but dominated by a block of l (each such block now has an additional set of paths that travel through a duplicate loop). Each such header is dominated by p', as each such header has exactly two

predecessors: block p' and its own latch. The remaining blocks are also dominated by p' as p' is the last dominating block on all paths until one of the $k+1$ loops are entered. □

Theorem 9.E.9. If all exits of f are dedicated, then unique dominator tree of $G_{f_{out}}$ is D_{out}.

Proof. By application of 9.E.8, Theorem 7.E.5 and Theorem 7.F.4. □

Lemma 9.E.10. If f is in canonical form, then F' reconstructed from (H', L'_k, X'_k) is the unique loop nesting forest of $G_{f'}$.

Proof. Each backedge introduced to the graph was done so by application of the *duplicate* procedure, which also duplicates the structure of l in the loop nesting forest. Clearly, the loop bodies and exit sets of the duplicated loops are correct. Additionally, as no edge was removed from the graph, each loop of $G_{f'}$ is represented one-to-one by an entry in F'. Let $b \notin body_f(l)$ be a block reachable from the body of l. By Corollary 9.E.3,b is also reachable from the body of each duplicated loop l'_i via a set of symmetric paths. Thus, l'_i must be immediately nested within $p_{F_f}(l)$. □

Theorem 9.E.11. If f is in canonical form, then F_{out} reconstructed from $(H_{out}, L_{out}, X_{out})$ is the unique loop nesting forest of $G_{f_{out}}$.

Proof. By application of Lemma 9.E.10, Theorem 7.E.6 and Theorem 7.F.5. □

9.F Loop Unrolling

In this section, we refer to the *unroll* transformation (notated below). For specific details (including names of intermediate components), refer to Figure 9.62.

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l]{\text{unroll}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Lemma 9.F.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f'' can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. We prove by filling in the slots of Template 6.A.1.

[**Slot → Refinement**] We define the predicate *oddIter*, as follows, that determines whether or not the evaluation is within an *odd* number iteration of the loop unrolled by the transformation (assuming evaluation is within the loop at all). This will be necessary as every other iteration of the loop will cause the parallel evaluation to enter the duplicated loop instead of the original. This predicate is true if the *last* occurrence of the preheader terminator occurs in the evaluation path is followed by an odd number of latch terminators.

$$oddIter(\rho, f_t) = \begin{cases} true & \langle \ldots, (p, \_), \overline{(c_i, b_i)} \rangle \\ & \text{where } (\forall c_i \cdot c_i \neq p) \wedge |\{c_i \mid c_i = T_l[\overrightarrow{\sigma_{f_t}}]\}| \text{ is odd} \\ false & otherwise \end{cases}$$

$$\text{where } p = \langle f_t, b_p, term(b_p) \rangle,\ b_p = preheader(l)[\overrightarrow{\sigma_{f_t}}],\ \text{and } T_l = term(latch(l))$$

We refine the translation of blocks by mapping clones of a block in l to the symmetric clone of a block in the duplicate loop. We refine the translation of streams by replacing the terminator of clones of the latch of l with terminators that target the relevant clone of the

duplicate loop header. For brevity, we decompose the stream s_t into $(s'_t, \hat{s}_t)$.

$$trans(\langle f_t, b_t \rangle) = \begin{cases} b_l[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(latch_f(l)) \wedge \neg odditer(\rho) \\ block_{f''}(\sigma_L(lab(b_t))[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(body_f(l)) \wedge odditer(\rho) \\ b_t & otherwise \end{cases}$$

$$trans(\langle f_t, b_t, (s_t, \hat{s}_t) \rangle) = \begin{cases} (s'_t, T_t[\ell_h/\ell_{h'}]), trans_\rho(\hat{s}_t) & b_t \in \mathcal{C}(latch_f(l)) \wedge \neg odditer(\rho) \\ (s'_t, T_t[\sigma_L][\ell_{h'}/\ell_h]), trans_\rho(\hat{s}_t) & b_t \in \mathcal{C}(latch_f(l)) \wedge odditer(\rho) \\ (s'_t, T_t[\sigma_L]), trans_\rho(\hat{s}_t) & b_t \in \mathcal{C}(body_f(l)) \wedge odditer(\rho) \\ s_t, trans_\rho(\hat{s}_t) & otherwise \end{cases}$$

We add the following additional relationship between register contexts.

$$\gamma_1 = \gamma_2$$

[Slot → Invokability] If the entry block of f is the latch of loop l (which can occur when canonical form is broken and the entry block forms a self-loop), then $b_{t_2} = b_l[\overrightarrow{\sigma_{f_{t_1}}}]$. Otherwise, the predicate $odditer(\rho)$ is false on entry and $b_{t_2} = b_{t_1}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[Slot → Asymmetric Evaluation] By the refined *trans* relation, all temporarily diverging steps of evaluation occur at the terminator of a clone of $latch(l)$ or within the body of l on odd iterations.

Latch Next, we cover the case where the $(n+1)$-th step of evaluation of f transfers control from block $b_t \in \mathcal{C}(latch(l))$. Let $T = \texttt{switch}\ v\ \overline{cv_i \mapsto ref_i}\ ref_d$ and let $ref = \ell_b(\overline{e_i})$ be the

block reference taken by this step of evaluation.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^n_{\rho_1} (\langle p, f_{t_1}, b_t\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ T, \hat{s}_{t_1})$$
$$\to_{\rho_2} (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_1}), \hat{s}_{t_1})$$

By our strengthened inductive hypothesis, the n'-th step of evaluation of f'' stops in function $f_{t_2} = \mathit{trans}_{\rho_1}(f_{t_1})$ on a terminator that transfers control from block $\mathit{trans}_{\rho_1}(\langle f_{t_1}, b_t\rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. The $(n'+1)$-th step of evaluation of f'' proceeds by application of rule E-Switch by taking the symmetric switch case.

$$(p[^f/_{f''}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f''(\overline{v_{t_i}})) \to^{n'}$$
$$(\langle p[^f/_{f''}], f_{t_2}, \mathit{trans}_{\rho_1}(\langle f_{t_1}, b_t\rangle)\rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ T', \mathit{trans}_{\rho_1}(\hat{s}_{t_1})) \to$$
$$(\langle p[^f/_{f''}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ \mathit{stream}(b_{t_2}), \mathit{trans}_{\rho_1}(\hat{s}_{t_1}))$$

First, consider the case where $b_{t_1} \in \mathcal{C}(b_h)$. Then, $\ell_b = \ell_{b_h}$. If if the predicate $\mathit{odditer}(\rho_1)$ is true, then $\mathit{trans}(\langle f_{t_1}, b_{t_1}\rangle) \in \mathcal{C}(b'_l)$ and $b_{t_2} \in \mathcal{C}(b_h)$. Either $b_{t_2} = b_{t_1}$ or, in the case that b_{t_1} forms a self-loop, $b_h = \mathit{latch}(l)$ and $b_{t_2} = b_l[\overrightarrow{\sigma_{f_{t_1}}}]$. Next, consider the case where $b_{t_1} \in \mathcal{C}(\mathit{body}_f(l))$. Then, $\ell_b \in \mathit{dom}(\sigma_L)$ and the blocks are either the same (when the predicate *odditer* is false) or mappable through σ_L (when the predicate *odditer* is true). Lastly, consider the case where $b_{t_1} \notin \mathcal{C}(\mathit{body}_f(l))$. Then, $\ell_b \notin \mathit{dom}(\sigma_L)$ and $b_{t_2} = b_{t_1}$. In all cases, the blocks and their unmodified streams are consistent with the strengthened induction.

Loop Body Next, we cover the cases where the n-th step of evaluation of f is within loop l and the predicate *odditer* is true. If evaluation of f is not yet at a terminator, then this case is trivial by application of Lemma 6.C.1. Now, consider when the $(n+1)$-th step of evaluation of f evaluates a terminator (distinct from a clone of the latch). This terminator is necessarily a `switch` as evaluation is currently within a loop. By our strengthened inductive

hypothesis, the n'-th step of evaluation of f'' stops at a state with a symmetric function, block, terminator, and register context. Let T_1 and T_2 be the terminator in the evaluation of functions f and f'', respectively. Here, $T_2 = T_1[\sigma_L]$. By application of Lemma 6.B.3, the resulting blocks are either the same or mappable through σ_L. The blocks and their unmodified streams are consistent with the strengthened induction. Additionally, the resulting register contexts are extended with the same set of register-value pairs, which does not violate the strengthened induction.

[**Slot ← Refinement**] We refine the translation of blocks and streams by collapsing all blocks and block references that belong to the duplicated loop to the symmetric block (or reference to the symmetric block) in the original loop. For brevity, we decompose the stream s_t into $(s'_t, \hat{s}_t)$.

$$trans^{-1}(\langle f_t, b_t \rangle) = \begin{cases} block_f(\sigma_L^{-1}(lab(b_t))[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(body_{f''}(l') \cup \mathcal{C}(latch_{f''}(l)) \\ b_t & otherwise \end{cases}$$

$$trans^{-1}(\langle f_t, b_t, (s_t, \hat{s}_t) \rangle) = \begin{cases} (s'_t, T_t[\sigma_L^{-1}]), trans^{-1}(\hat{s}_t) & b_t \in \mathcal{C}(body_{f''}(l') \cup \mathcal{C}(latch_{f''}(l)) \\ s_t, trans^{-1}(\hat{s}_t) & otherwise \end{cases}$$

We add the following additional relationship between register contexts.

$$\gamma_2 = \gamma_1$$

[**Slot ← Invokability**] It cannot be the cases that $b_{t_2} \in \mathcal{C}(body_{f''}(l'))$. If the entry block of f'' is the latch of loop l, then $b_{t_1} = latch_f(l)[\overrightarrow{\sigma_{f_{t_2}}}]$. Otherwise, $b_{t_1} = b_{t_2}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[**Slot ← Asymmetric Evaluation**] By the refined $trans^{-1}$ relation, all temporarily diverging steps of evaluation occur at the terminator of a clone of $latch(l)$ or within the body of l'.

Original Latch First, we cover the case where the $(n'+1)$-th step of evaluation of f'' transfers control from block $b_t \in \mathcal{C}(latch(l))$. Let $T = \texttt{switch}\ v\ \overline{cv_i \mapsto ref_i}\ ref_d$ and let $ref = \ell_b(\overline{e_i})$ be the block reference taken by this step of evaluation.

$$(p[^f/_{f''}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f''(\overline{v_{t_i}})) \to^{n'}$$
$$(\langle p[^f/_{f''}], f_{t_2}, b_t)\rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ T[^{\ell_{b_h}}/_{\ell_{b'_h}}], trans^{-1}(\hat{s}_{t_1})) \to$$
$$(\langle p[^f/_{f''}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_2}), \hat{s}_{t_2})$$

By our strengthened inductive hypothesis, the n-th step of evaluation of f stops in function $f_{t_1} = trans^{-1}(f_{t_2})$ on a terminator that transfers control from block $trans^{-1}(\langle f_{t_2}, b_t\rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. The $(n+1)$-th step of evaluation of f proceeds by application of rule E-Switch by taking the symmetric switch case.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^{n}$$
$$(\langle p, f_{t_1}, trans^{-1}(\langle f_{t_1}, b_t\rangle\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ T, trans^{-1}(\hat{s}_{t_2})) \to$$
$$(\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_1}), trans^{-1}(\hat{s}_{t_2}))$$

First, consider the case where $b_{t_2} \in \mathcal{C}(b'_h)$. Then, $\ell_b = \ell_{b'_h}$ and $b_{t_1} \in \mathcal{C}(b_h)$ by translation. In the other cases, $b_{t_2} \notin \mathcal{C}(body_{f''}(l')$, thus $b_{t_2} = b_{t_1}$. In all cases, the blocks and their unmodified streams are consistent with the strengthened induction.

Loop Body Next, we cover the cases where the n'-th step of evaluation of f'' is within loop l'. If evaluation of f'' is not yet at a terminator, then this case is trivial by application

of Lemma 6.C.1. Now, consider when the $(n'+1)$-th step of evaluation of f'' evaluates a terminator (distinct from a clone of the latch). This terminator is necessarily a `switch` as evaluation is currently within a loop. By our strengthened inductive hypothesis, the n-th step of evaluation of f stops at a state with a symmetric function, block, terminator, and register context. Let T_1 and T_2 be the terminator in the evaluation of functions f and f'', respectively. Here, $T_1 = T_2[\sigma_L^{-1}]$. By application of Lemma 6.B.3, the resulting blocks are either the same or mappable through σ_L^{-1}. The blocks and their unmodified streams are consistent with the strengthened induction. Additionally, the resulting register contexts are extended with the same set of register-value pairs, which does not violate the strengthened induction. □

Theorem 9.F.2. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. By application of Lemma9.F.1, Theorem 7.F.1, and Theorem 7.A.1. □

Corollary 9.F.3. The set of paths over block labels from ℓ_0 to $\ell_{b'}$ in $G_{f'}$ for some block $b' \in B$ where $lab(b') = \ell_{b'}$ can be constructed by replacing each path $p_1 \cup p_2$ where p_1 and p_2 are the minimal subpaths of the forms $\langle lab(header(l)), \ldots, lab(latch(l))\rangle$ and $lab(header(l)), \ldots, \ell_e\rangle$, respectively, and $\ell_e \in exit(l) \cup latch(l)$ by the path $p_1 \cup p_2[\sigma_L]$ in the set of such paths in G_f.

Theorem 9.F.4. If f is in canonical form, then f_{out} is in canonical form.

Proof. As each exit of l has dedicated exits in f, and loop l' shares the same exits, the exits of the union of loops l and l' are also dedicated. Let l_c be a child of l and let l'_c be the loop duplicated from l_c. These loops have the same exit sets but are siblings in the loop nesting forest, thus each exit has necessarily become undedicated. These are precisely the

loops whose exits are rededicated by the transformation. No other paths outside of loop l change by Corollary 9.F.3. □

Theorem 9.F.5. If f is in LCSSA form, then f_{out} is in LCSSA form.

Proof. All definitions and uses introduced into the intermediate function f' are those in loop l'. Loops l and l' merge to become one strongly connected component, so any additional use in l' is introduced to a loop that already used that register and thus is not a violation. Similarly, let l_c be a child of l and let l'_c be the loop duplicated from l_c. Loop l_c and l'_c are siblings in the resulting loop nesting forest, and each use in l'_c is then either defined by a parent loop, or is defined by a definition also duplicated from l_c. Then, the rest follows by application of Theorem 7.A.7. □

Theorem 9.F.6. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f''$ is well-typed.

Proof. Each block $b \in body_f(l)$ and its duplicates have the same set of register uses. By Corollary 9.F.3, the set of register uses reachable from block b and the set of register uses reachable from one of its duplicates are equivalent. Further, each block b and its duplicates also define the same registers and thus have the same implicit block parameters and nonlocal block parameters. Both latches are modified to point to the opposite header. Both headers have the same set of block parameters, implicit block parameters, and stream. Clearly, each block b and its duplicates have identical typing derivations. □

Theorem 9.F.7. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f_{out}$ is well-typed.

Proof. By application of Lemma 9.F.6, Theorem 7.F.3, and Theorem 7.A.2. □

Lemma 9.F.8. The unique dominator tree of $G_{f''}$ is D''.

Proof. Block h' is immediately dominated by $latch(l)$ as it is its sole predecessor. All other blocks inserted by application of the *duplicate* procedure has the correct immediate dominator, as there is no direct path to any block that does not first travel through h'. Let b be

a block that change immediate dominator. By Corollary 9.F.3, this block must have been previously dominated by a block in l for which there is no an additional path that instead passes through the additional blocks. These set of such violating blocks are described completely by the set B_D and each dominator is be calculated from the nearest common ancestor in the old dominator tree as shown by Alstrup [8]. □

Theorem 9.F.9. The unique dominator tree of $G_{f_{out}}$ is D_{out}.

Proof. By application of Lemma 9.F.8 and Theorem 7.F.4. □

Theorem 9.F.10. If f is in canonical form, then F_{out} reconstructed from (H'', L'', X'') is the unique loop nesting forest of $G_{f''}$.

Proof. All blocks in l have a path to $latch(l)$ in f. Similarly, all blocks in l' have a path to $lath(l')$ in f'. Once the backedges are flipped, all blocks in l have a path to $header(l')$, therefore all blocks in l', and all blocks in l have a path to $header(l)$ through the combined loop's backedge. Let l_c be a child of l and let l'_c be the loop duplicated from l_c. Loop l_c has a preheader that previously belonged to loop l and loop l'_c has a preheader that previously belonged to loop l'. Both loops also share the same set of exits. As l_c is a child loop of l, one exit must exist on the path to the latch of its parent. Therefore, loop l'_c can reach the same latch and must be a sibling of l_c. □

Theorem 9.F.11. If f is in canonical form, then F_{out} reconstructed from $(H_{out}, L_{out}, X_{out})$ is the unique loop nesting forest of $G_{f_{out}}$.

Proof. By application of Lemma 9.F.11 and Theorem 7.F.5. □

9.G Loop Peeling

In this section, we refer to the *peel* transformation (notated below). For specific details (including names of intermediate components), refer to Figure 9.72.

$$(f, D_f, H_{F_f}, L_{F_f}, X_{F_f}) \xrightarrow[l]{\text{peel}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Lemma 9.G.1. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f'' can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. We prove by filling in the slots of Template 6.A.1.

[**Slot → Refinement**] s We define the predicate *firstiter*, as follows, that determines, assuming evaluation is somewhere within the loop, whether or not evaluation is in the first iteration of the loop peeled by the transformation. This will be necessary as every iteration of the loop after the first causes the parallel evaluation to enter the duplicated loop rather than the original. This predicate is true if the backedge terminator occurs after the *last* occurrence of the preheader terminator in the evaluation path.

$$\mathit{firstiter}(\rho, f_t) = \begin{cases} \mathit{true} & \langle \ldots, (p, b''), \overline{(c_i, b_i)} \rangle \\ & \text{where } (\forall c_i \cdot c_i \neq p) \wedge |\{c_i \mid c_i = T_l[\overrightarrow{\sigma_{f_t}}]\}| \geq 0 \\ \mathit{false} & \mathit{otherwise} \end{cases}$$

$$\text{where } p = \langle f_t, b_p, \mathit{term}(b_p) \rangle,\ b_p = \mathit{preheader}(l)[\overrightarrow{\sigma_{f_t}}],\ \text{and } T_l = \mathit{term}(\mathit{latch}(l))$$

We refine the translation of blocks by mapping clones of a block in l to the symmetric clone of a block in the duplicate loop. We refine the translation of streams by replacing the

terminator of clones of the latch of l with a terminators that targets the relevant clone of the duplicate loop header. For brevity, we decompose the stream s_t into $(s'_t, \hat{s}_t)$.

$$trans_\rho(\langle f_t, b_t \rangle) = \begin{cases} b_l[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(latch_f(l)) \wedge firstiter(\rho) \\ block_{f''}(\sigma_L(lab(b_t))[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(body_f(l)) \wedge \neg firstiter(\rho) \\ b_t & otherwise \end{cases}$$

$$trans_\rho(\langle f_t, b_t, (s_t, \hat{s}_t) \rangle) = \begin{cases} (s'_t, T_t[\ell_{b_h}/\ell_{b'_h}]), trans_\rho(\hat{s}_t) & b_t \in \mathcal{C}(latch_f(l)) \wedge firstiter(\rho) \\ (s'_t, T_t[\sigma_L]), trans_\rho(\hat{s}_t) & b_t \in \mathcal{C}(body_f(l)) \wedge \neg firstiter(\rho) \\ s_t, trans_\rho(\hat{s}_t) & otherwise \end{cases}$$

We add the following additional relationship between register contexts.

$$\gamma_1 = \gamma_2$$

[**Slot → Invokability**] If the entry block of f is the latch of loop l (which can occur when canonical form is broken and the entry block forms a self-loop), then $b_{t_2} = b_l[\overrightarrow{\sigma_{f_{t_1}}}]$. Otherwise, the predicate $firstiter(\rho)$ is true on entry and $b_{t_2} = b_{t_1}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[**Slot → Asymmetric Evaluation**] By the refined *trans* relation, all temporarily diverging steps of evaluation occur at the terminator of a clone of $latch(l)$ or within the body of loop l.

Latch First, we cover the case where the $(n+1)$-th step of evaluation of f transfers control from block $b_t \in \mathcal{C}(latch(l))$ and the predicate $firstiter(\rho_1)$ is true. For legibility, we

let $T = \texttt{switch}\ v\ \overline{cv_i \mapsto ref_i}\ ref_d$. Additionally, let $ref = \ell_b(\overline{e_i})$ be the block reference taken by this step of evaluation.

$$\begin{aligned}(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) &\to^n_{\rho_1} (\langle p, f_{t_1}, b_t\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ T, \hat{s}_{t_1}) \\ &\to_{\rho_2} (\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_1}), \hat{s}_{t_1})\end{aligned}$$

By our strengthened inductive hypothesis, the n'-th step of evaluation of f'' stops in function $f_{t_2} = trans_{\rho_1}(f_{t_1})$ on a terminator that transfers control from block $trans_{\rho_1}(\langle f_{t_1}, b_t\rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. The $(n'+1)$-th step of evaluation of f'' proceeds by application of rule E-Switch by taking the symmetric switch case.

$$\begin{gathered}(p[^f/_{f''}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f''(\overline{v_{t_i}})) \to^{n'} \\ (\langle p[^f/_{f''}], f_{t_2}, trans_{\rho_1}(\langle f_{t_1}, b_t\rangle)\rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ T[^{\ell_{b_h}}/_{\ell_{b'_h}}], trans_{\rho_1}(\hat{s}_{t_1})) \to \\ (\langle p[^f/_{f''}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_2}), trans_{\rho_1}(\hat{s}_{t_1}))\end{gathered}$$

First, consider the case where $b_{t_1} \in \mathcal{C}(b_h)$. Then, $\ell_b = \ell_{b_h}$ and $b_{t_2} \in \mathcal{C}(b'_h)$ by translation. In the other case, the predicate $firstiter(\rho_2)$ remains true in the subsequent step of evaluation as the backedge is not taken and $b_{t_2} = trans(\langle f_{t_1}, b_{t_1}\rangle)$.

Loop Body Next, we cover the cases where the n-th step of evaluation of f is within loop l and the predicate $firstiter(\rho)$ is false. If evaluation of f is not yet at a terminator, then this case is trivial by application of Lemma 6.C.1. Now, consider when the $(n+1)$-th step of evaluation of f evaluates a terminator (distinct from a clone of the latch). This terminator is necessarily a `switch` as evaluation is currently within a loop. By our strengthened inductive hypothesis, the n'-th step of evaluation of f'' stops at a state with a symmetric function, block, terminator, and register context. Let T_1 and T_2 be the terminator in the evaluation of

functions f and f'', respectively. Here, $T_2 = T_1[\sigma_L]$. By application of Lemma 6.B.3, the resulting blocks are either the same or mappable through σ_L. The blocks and their unmodified streams are consistent with the strengthened induction. Additionally, the resulting register contexts are extended with the same set of register-value pairs, which does not violate the strengthened induction.

[**Slot ← Refinement**] We refine the translation of blocks and streams by collapsing all blocks and block references that belong to the duplicated loop to the symmetric block (or reference to the symmetric block) in the original loop. For brevity, we decompose the stream s_t into $(s'_t, \hat{s}_t)$.

$$trans^{-1}(\langle f_t, b_t \rangle) = \begin{cases} latch(l)[\overrightarrow{\sigma_{f_{t_2}}}] & b_t \in \mathcal{C}(latch_{f''}(l)) \\ block_f(\sigma_L^{-1}(lab(b_t))[\overrightarrow{\sigma_{f_t}}] & b_t \in \mathcal{C}(body_{f''}(l') \\ b_t & otherwise \end{cases}$$

$$trans^{-1}(\langle f_t, b_t, (s_t, \hat{s}_t) \rangle) = \begin{cases} (s'_t, T_t[\sigma_L^{-1}]), trans^{-1}(\hat{s}_t) & b_t \in \mathcal{C}(body_{f''}(l') \cup \mathcal{C}(latch_{f''}(l)) \\ s_t, trans^{-1}(\hat{s}_t) & otherwise \end{cases}$$

We add the following additional relationship between register contexts.

$$\gamma_2 = \gamma_1$$

[**Slot ← Invokability**] It cannot be the case that $b_{t_2} \in \mathcal{C}(body_{f''}(l'))$. If the entry block of f'' is the latch of loop l, then $b_{t_1} = latch_f(l)[\overrightarrow{\sigma_{f_{t_2}}}]$. Otherwise, $b_{t_1} = b_{t_2}$. In either case, the unmodified streams and register contexts are consistent with the strengthened induction without an additional step of evaluation.

[**Slot ← Asymmetric Evaluation**] By the refined $trans^{-1}$ relation, all temporarily diverging steps of evaluation occur at the terminator of a clone of $latch(l)$ or within the body of l'.

Original Latch First, we cover the case where the $(n'+1)$-th step of evaluation of f'' transfers control from block $b_t \in \mathcal{C}(latch(l))$. Let $T = \texttt{switch}\ v\ \overline{cv_i \mapsto ref_i}\ ref_d$ and let $ref = \ell_b(\overline{e_i})$ be the block reference taken by this step of evaluation.

$$(p[^f/_{f''}] \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f''(\overline{v_{t_i}})) \to^{n'}$$
$$(\langle p[^f/_{f''}], f_{t_2}, b_t)\rangle \mid \gamma_2 \mid \mu' \mid \Psi';\ T[^{\ell_{b_h}}/_{\ell_{b'_h}}], trans^{-1}(\hat{s}_{t_1})) \to$$
$$(\langle p[^f/_{f''}], f_{t_2}, b_{t_2}\rangle \mid \gamma_2[\overline{r_i \mapsto \gamma_2(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_2}), \hat{s}_{t_2})$$

By our strengthened inductive hypothesis, the n-th step of evaluation of f stops in function $f_{t_1} = trans^{-1}(f_{t_2})$ on a terminator that transfers control from block $trans^{-1}(\langle f_{t_2}, b_t\rangle)$, as shown below, such that γ_1 and γ_2 are consistent with the strengthened induction. The $(n+1)$-th step of evaluation of f proceeds by application of rule E-Switch by taking the symmetric switch case.

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{v_{t_i}})) \to^{n}$$
$$(\langle p, f_{t_1}, trans^{-1}(\langle f_{t_1}, b_t\rangle\rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ T, trans^{-1}(\hat{s}_{t_2})) \to$$
$$(\langle p, f_{t_1}, b_{t_1}\rangle \mid \gamma_1[\overline{r_i \mapsto \gamma_1(e_i)}] \mid \mu' \mid \nu' \mid \Psi';\ stream(b_{t_1}), trans^{-1}(\hat{s}_{t_2}))$$

First, consider the case where $b_{t_2} \in \mathcal{C}(b'_h)$. Then, $\ell_b = \ell_{b'_h}$ and $b_{t_2} \in \mathcal{C}(b_h)$ by translation. Otherwise, $b_{t_1} \notin \mathcal{C}(body_{f''}(l'))$ and $b_{t_2} = b_{t_1}$.

Loop Body Next, we cover the cases where the n'-th step of evaluation of f'' is within loop l'. If evaluation of f'' is not yet at a terminator, then this case is trivial by application of Lemma 6.C.1. Now, consider when the $(n'+1)$-th step of evaluation of f'' evaluates a

terminator (distinct from a clone of the latch). This terminator is necessarily a `switch` as evaluation is currently within a loop. By our strengthened inductive hypothesis, the n-th step of evaluation of f stops at a state with a symmetric function, block, terminator, and register context. Let T_1 and T_2 be the terminator in the evaluation of functions f and f'', respectively. Here, $T_1 = T_2[\sigma_L^{-1}]$. By application of Lemma 6.B.3, the resulting blocks are either the same or mappable through σ_L^{-1}. The blocks and their unmodified streams are consistent with the strengthened induction. Additionally, the resulting register contexts are extended with the same set of register-value pairs, which does not violate the strengthened induction. □

Theorem 9.G.2. Assume $p \mid f$ is well-typed. If f can be evaluated n steps with some register context, memory context, nondeterminism state, and effects list, then f_{out} can be evaluated for n' steps with the same initial context and arguments and reach a state with the same memory context, nondeterminism state, and a similar effects list (and vice versa).

Proof. By application of Lemma 9.G.1, Theorem 7.E.1, Theorem 7.F.1, and Theorem 7.A.1. □

Corollary 9.G.3. The set of paths over block labels from ℓ_0 to $\ell_{b'}$ in $G_{f'}$ for some block $b' \in B$ where $lab(b') = \ell_{b'}$ can be constructed by replacing each path $p_1 \cup \langle \overline{p_i} \rangle$ where p_1 and p_i are the minimal subpaths of the forms $\langle lab(header(l)), \ldots, lab(latch(l)) \rangle$ and $lab(header(l)), \langle \overline{b_i} \rangle$ where all $b_i \in block(l)$, respectively, by the path $p_1[\sigma_L] \cup \langle \overline{p_i} \rangle$ in the set of such paths in G_f.

Theorem 9.G.4. If f is in canonical form, then f_{out} is in canonical form.

Proof. If block b_l contains a branch to an exit block of loop l, then b_l would not be dedicated to loop l'. In this circumstance, the preheader of loop l' is rededicated. The exit set of loop l' contains blocks that are reachable from the blocks that originally belonged to l. Let l_c be a child of l and let l'_c be the loop duplicated from l_c. Distinct loops l_c and l'_c also share

the same set of exits and are necessarily undedicated. These are precisely the loops whose exits are rededicated by the transformation. No other paths outside of loop l change by Corollary 9.F.3. □

Theorem 9.G.5. If f is in LCSSA form, then f_{out} is in LCSSA form.

Proof. The application of *rewrite uses* effectively maps the set of registers defined by $\mathit{def}(l)$ to a set of fresh registers in l'. This is obvious as the reaching definition of each such use is the unique definition of the used register. Then, the set of definitions introduced by loop l' are fresh registers, and each use is also contained within l'. When loop l is destroyed, all definitions and uses are moved to the parent of l. As each use of a definition of l is also contained in l, then each use of a definition in the parent of l is also contained in the parent of l. Lastly, let l_c be a child of l and let l'_c be the loop duplicated from l_c. The nearest ancestor of l_c and l'_c in the resulting loop nesting forest is the parent of l, and each use in l'_c is then either defined by a parent loop, or is defined by a definition also duplicated from l_c. □

Lemma 9.G.6. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f''$ is well-typed.

Proof. Each block $b \in \mathit{body}_f(l)$ and its duplicates have the same set of register uses. By Corollary 9.G.3, every block of l can reach every block of l'. Additionally, every block of l and l' can reach the same set of blocks outside of loops l and l'. The set of register uses reachable from block b and the set of register uses reachable from one of its duplicates are then equivalent. Further, each block b and its duplicates also define the same registers and thus have the same implicit block parameters and nonlocal block parameters. The block $\mathit{latch}(l)$ is modified to point to the duplicated loop header. Both l and l' have the same set of block parameters, implicit block parameters, and stream. Clearly, each block b and its duplicates have identical typing derivations. □

Theorem 9.G.7. If $p \mid f$ is well-typed and f is in SSA form, then $p \mid f_{out}$ is well-typed.

Proof. By application of 9.G.6, Theorem 7.E.4, Theorem 7.F.3, and Theorem 7.A.2. □

Lemma 9.G.8. The unique dominator tree of $G_{f''}$ is D''.

Proof. Block $header(l')$ is immediately dominated by $latch(l)$ as it is its sole forward-edge predecessor. All other blocks inserted by application of the *duplicate* procedure has the correct immediate dominator, as there is no direct path to any block that does not first travel through $header(l')$. Let b be a block that change immediate dominator. By Corollary 9.G.3, this block must have been previously dominated by a block in l for which there is no an additional path that instead passes through the additional blocks. These set of such violating blocks are described completely by the set B_D and each dominator is be calculated from the nearest common ancestor in the old dominator tree as shown by Alstrup [8]. □

Theorem 9.G.9. The unique dominator tree of $G_{f_{out}}$ is D_{out}.

Proof. By application of 9.G.8, Theorem 7.E.5 and Theorem 7.F.4. □

Lemma 9.G.10. If f is in canonical form, then F_{out} reconstructed from (H'', L'', X'') is the unique loop nesting forest of $G_{f''}$.

Proof. Once the backedge of loop l is modified, each block in l no longer has a path back to $header(l)$ without first traveling through the preheader which, by definition, is not within the loop body. Then, the original loop l is destroyed and all references to it in H'', L'', and X'' are removed. The children of l are not modified, but their placement in the loop nesting hierarchy *goes up* one level – each child is still nested within the parent of l. As loop l' has the same exit blocks as loop l, it also shares a path to the latch of its parent and therefore is also nested under the parent of l. □

Theorem 9.G.11. If f is in canonical form, then F_{out} reconstructed from $(H_{out}, L_{out}, X_{out})$ is the unique loop nesting forest of $G_{f_{out}}$.

Proof. By application of 9.G.10, Theorem 7.E.6 and Theorem 7.F.5. □

10 Evaluation

In this chapter we present evaluation results using a proof-of-concept implementation of an optimizer using the algorithms discussed in this work. This implementation is written in Scala and supports the following optimization passes:

1. Constant Propagation
2. Copy Propagation
3. Dead Code Elimination
4. If Simplify (IS)
5. Function Inline (FI)
6. Jump Simplify (JS)
7. Jump Thread (JT)
8. Straighten (S)
9. Loop Unroll (UNR)
10. Loop Unswitch (UNS)

The first three passes do not have an effect on control flow (dead code elimination is applied over instructions, not blocks) and do not play a role in this evaluation. The remainder of this chapter discusses the source programs used for the evaluation, the evaluation methodology, and a discussion of the results.

10.1 Source Programs

The flow graphs of the Waddle IR used for this evaluation were generated from compiled LLVM IR. The resulting flow graphs are (mostly) isomorphic in control flow to the LLVM IR, but each instruction in the generated graph is chosen randomly. Exception flow, indirect function calls, and unreachability, for which there are currently no direct representations in the Waddle IR, were approximated as closely as possible. An instruction in a basic block of

the source IR and the generated instruction in the symmetric basic block of the generated IR use the same register names.

The resulting generated graphs have the same control flow structure and live register intervals of a real-world program. This provides programs expressed as valid Waddle IR that have opportunities for jump, loop, and inline optimizations that would occur in real programs. This is more than sufficient for this benchmark, as the *exact* semantics of the program are inconsequential.

The six programs used to generate the LLVM IR are single-file C++11 implementations of graph algorithms taken from the GAP Benchmark Suite [11], described below. These sources were chosen because they utilize a wide variety of iterative structures that were likely to make realistic yet interesting loop nesting forests. The original source (git commit `f166dc4`) is available on Github[1]. Each compilation unit performs an operation over a graph whose edges are stored as compressed sparse rows.

bc.cc

Calculate betweenness centrality scores for each vertex

bfs.cc

Perform a breadth-first traversal and create a mapping from a vertex to its predecessor in the traversal order.

cc.cc

Label each vertex with an identifier of connected component to which it belongs.

pr.cc

Calculate PageRank scores for each vertex. This program terminates once the total change is less than some (supplied) epsilon.

[1] `https://github.com/sbeamer/gapbs`

sssp.cc

Calculate the shortest path distance from a source vertex to all other vertices.

tc.cc

Count the number of triangles (cliques of size 3) in an undirected graph where neighborhoods are sorted by vertex labels.

To give a brief sense of the scale of the Waddle IR used here: each compilation unit contains between 363 and 445 total function definitions, between 71 and 101 of which are interesting (neither trivial, a single node, nor loop-less). There are between 2 and 237 blocks per function (with an average of 21 blocks per function) and between 2 and 354 edges per function (with an average of 30 edge per function). Each interesting function contains between 1 and 10 loops with depths ranging between 1 (top-level) and 4. The size of loop body sets range between 1 and 99 blocks, and the size of loop exit sets range between 0 and 10 blocks. Many of the loops with empty exit sets come from Waddle's equivalent of an *unreachable* LLVM instruction, in which a single trivial block branches back to itself.

10.2 Methodology

We refer to an *optimization pass* as a unit of work over a single function that first detects opportunities for optimizations, then builds a worklist, then applies a transformation to the function for each value in the worklist. A single pass performs only one type of transformation, but may apply that transformation many times. For instance, an if simplification pass may detect multiple branches targets of which are based on constant values. A single pass would reduce all of these multiple-target switches into unconditional jumps. It may also be the case that an item in the worklist is no longer a candidate for the transformation at time the transformation is applied. In the example above, the deletion of an edge may

make an item not yet processed in the worklist unreachable. Such items are simply skipped as the worklist is processed. There may also be a circumstance where performing a transformation uncovers a further opportunity for optimization that was not found when building the original worklist. Running an optimization pass a second time will detect such missed opportunities.

Optimization passes will also limit the resulting code size by following a handful of basic heuristics. A function will not be inlined if it consists of more than 100 blocks. Similarly, a loop will neither be unswitched nor unrolled if the body set consists of more than 100 blocks. Additionally, there may be at most 15 applications of loop unrolling and at most 3 applications of loop unswitching per pass. Finally, a branch with more than 8 successors is never eligible for loop unswitching.

We present results in two parts. First, we compare a single optimization pass that incrementally repairs canonical form against an alternative implementation that performs the least amount of work during the transformation (not caring to repair dominator or loop information) then rebuilds canonical form from scratch at the end of optimization. Both versions of the passes maintain static single assignment, implicit block parameters, and set of register uses. These properties are handled implicitly by Waddle's IR API, so there is no extra effort required for the 'dumb' versions of the transformation.

Second, we compare a sequence of optimization passes that incrementally repairs canonical form against a sequence of optimization passes that rebuild only the necessary properties of the IR between passes, again performing the least amount of work necessary to effectively perform the optimization. This is similar to how the pass manager in LLVM behaves and gives a good preliminary idea of how this methodology performs against realistic workloads. The if simplification, jump simplification, jump threading, and straightening passes require

domination information. This is used to build the worklist in a deterministic fashion (a post-order walk over the dominator tree) so that the resulting functions between both versions are the same. The function inliner requires loop information. As the maximum amount of work for this pass is limited, the loop nesting structure is used in order to populate the worklist with the inline opportunities that will have the most impact on the resulting function. As folk wisdom tells us, the (likely) beneficial callsites to inline are located in the deepest loops. These loops are likely to be evaluated more frequently and thus saves on stack manipulation and function prologue/epilogue overhead. Loop unrolling requires loop information (unsurprisingly) to determine the size and cost of optimization opportunities as well as identify the bodies which must be duplicated. Loop unswitching requires canonical form in addition to basic loop information. Part of this optimization attempts to make uses of registers in loop bodies invariant by hoisting instructions into the loop's dedicated preheader. The latter two passes are written so that the required properties are rebuilt between successful loop operations. This is required as subsequent application of a loop operation using stale data may malform the IR or, even worse, change the semantics of the program under optimization. Additionally, canonical form is rebuilt from scratch after all optimization passes have run over a particular function. This is necessary as the target of inline operations are expected to be in canonical form.

10.3 Single Pass

Here, we present the results for individual optimization passes. For each pass, the following range of operations were performed: if simplification was performed between 1013 and 1242 times; function inlining was performed 207 to 256 times; jump simplification was performed 4 to 26 times; jump threading was performed 4 to 28 times; straightening was performed 255

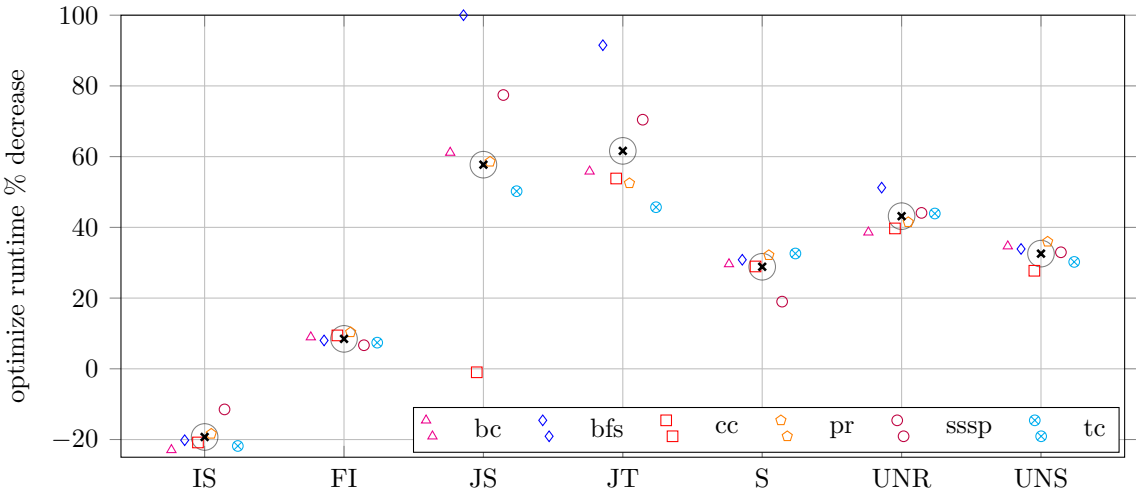


Figure 10.31: Runtime Decrease comparisons between incremental maintenance of properties and complete repair of properties for a single transformation pass application.

to 321 times; loop unrolling was performed 124 to 175 times; loop unswitching was performed 106 to 130 times.

For every optimization except for IS, the incremental strategy outperforms the strategy that rebuilds structures at the end of the pass. This is likely due to the large number of operations performed during this pass (a factor of four over the other passes). Due to the way the program IR is generated, switch values that have no direct translation from LLVM IR are replaced by a randomized constant. This creates a large number of optimization opportunities for jump simplification, which would not occur in practice for the same program. If fewer opportunities were presented so that a smaller batch of work was performed per pass, then the incremental approach would likely see an increased benefit with respect to the recomputation approach.

Figure 10.31 combines the results of each program to show the percentage decrease in runtime compared to the recomputation strategy. Here, a 50% decrease in runtime is the same as running in half the time, and a -20% decrease in runtime is the same as taking an additional 20%. The average over all benchmarks for each optimization pass are presented

as a circled cross. Again, the performance of the incremental IS pass does not compare well against the dumb bulk repair strategy. Function inlining gives a modest decrease in runtime. Straightening and loop optimizations perform solidly, giving between a 28% and a 43% decrease in runtime. Jump optimizations have the most striking decrease in runtime, hovering around 60%. The same results by the program under optimization are given in Figure 10.32. Strangely, **bfs.ir** did not find any opportunities for jump optimizations (and was omitted from the above lists).

These results are encouraging, especially paired with a theory for the poor performance of the IS optimization pass. As we will discuss in Chapter 11, validating this theory and optimizing the performance of the incremental approach is a good candidate for future work.

10.4 Pass Sequence

Next, we present the results for a single, fixed sequence of optimizations. This sequence applies the following transformation passes over each function one at a time: FI, JT, JS, UNS, IS, S, UNS, UNR, S. Straightening is applied after unrolling and if simplification, as these operations tend to create non-critical edges. If simplification is applied directly after the first instance of loop unswitching,s as each cloned loop can immediately be reduced due to the unconditional branch created in the loop body. This reduction can sometimes be drastic. Figure 10.41 partitions runtime data by program. The programs **bc.ir** and **cc.ir** show a solid reduction in optimization runtime. The remaining programs hover within 5% of the original runtime, which is not a large enough jump to attribute any significant results in either the positive or negative direction. Figure 10.42 additionally gives the number of operations performed for each program for this optimization sequence.

By the nature of phase ordering, such constructed tests are very difficult to evaluate. Each

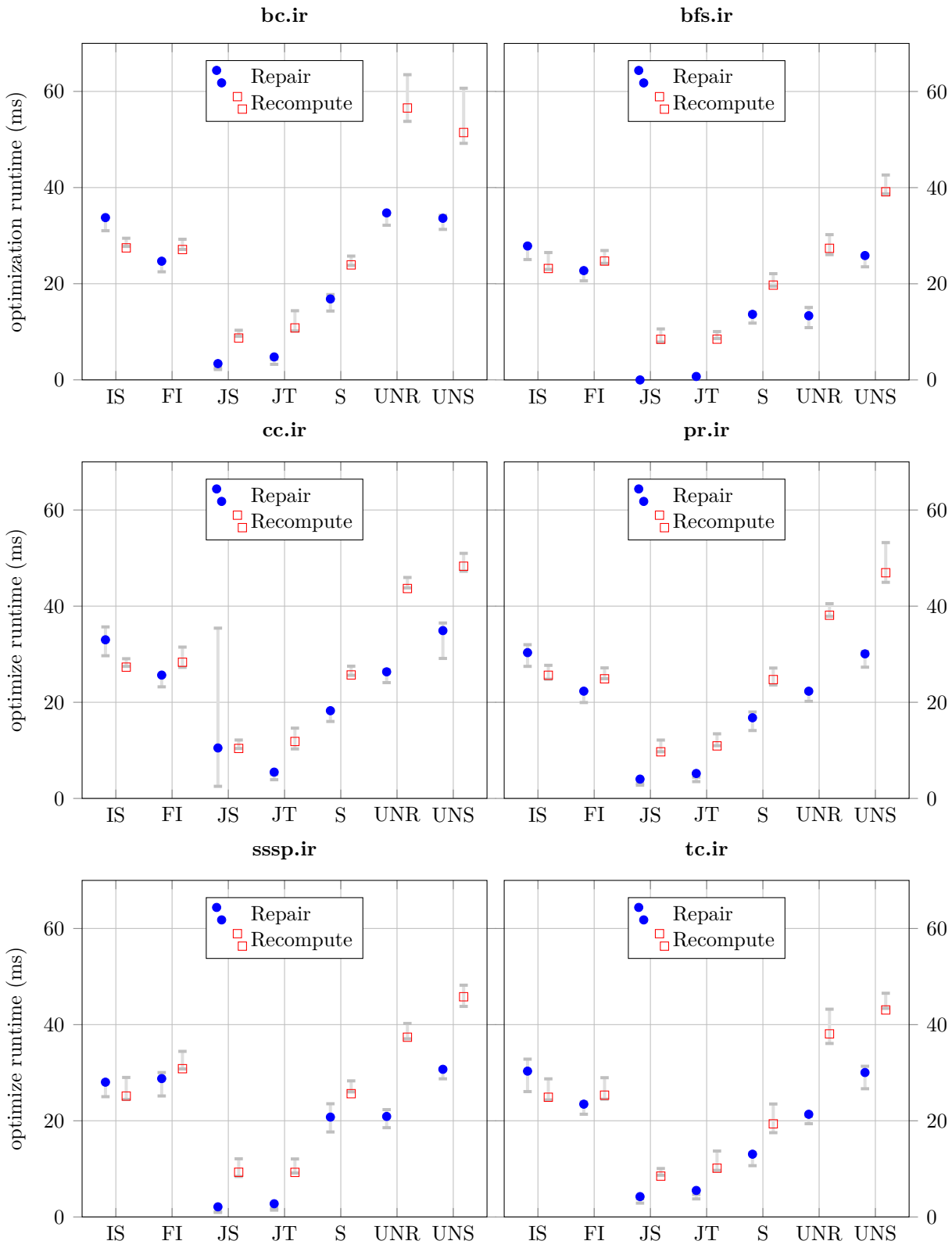


Figure 10.32: Runtime comparisons between incremental maintenance of properties and complete repair of properties for a single transformation pass application.

Program	Incremental Repair (ms)	Rebuild (ms)	Runtime % Decrease
bc.ir	2929 (±071)	3572 (±612)	+18.00
bfs.ir	2215 (±079)	2210 (±045)	−0.23
cc.ir	2652 (±023)	2960 (±131)	+10.41
pr.ir	2384 (±066)	2407 (±067)	+0.96
sssp.ir	2357 (±073)	2413 (±042)	+2.32
tc.ir	2403 (±110)	2303 (±088)	−4.34

Figure 10.41: Runtime results for a single, fixed sequence of optimizations.

	bc.ir	bfs.ir	cc.ir	pr.ir	sssp.ir	tc.ir
IS	2218	1903	2064	1923	1958	1838
FI	235	228	212	208	254	204
JS	10	0	8	6	6	6
JT	39	4	42	38	4	39
S	1640	1528	1404	1424	1439	1379
UNR	97	78	75	72	88	68
UNS	152	128	136	128	142	122

Figure 10.42: Number of operations performed during optimization.

program will react radically differently to distinct orders of optimizations. This evaluation gives only a single order, chosen arbitrarily with respect to performance. Other programs with different branching and register live range structures, other orders of optimization passes, and different values for opportunity heuristics may produce radically different results, affecting both the optimized program as well as the runtime required to produce it.

The combined results for this *particular* optimization pass, while not outstanding, is a necessary sanity check to ensure that the performance of incremental repair is not detrimental to the runtime of an optimizer. As this proof of concept implementation has moderately exceeded its baseline, it is now worth the effort to explore the future directions of such incremental methodologies, further discussed in Chapter 11.

11 Future Directions

In this chapter we briefly present ideas of how this research can be applied and extended in the future.

Evaluation We have shown that the incremental approach to maintaining canonical form is *correct* and have also shown that each individual pass is *performant*. The multiple-pass evaluation presented here shows that the incremental approach does not backstep with respect to performance, but shows this only for a very specific, very small sequence of optimizations. This is partly because the proof of concept implementation only has a complete implementation for the optimization passes described here. Future work should further evaluate the incremental approach in the context of multiple passes with differing order of phases.

In a blog post by John Regehr [55], the complete optimization of the following function by Clang/LLVM 6.0.1 is illustrated.

```
bool is_sorted(int *a, int n) {
    for (int i = 0; i < n - 1; i++) {
        if (a[i] > a[i + 1]) return false;
    }

    return true;
}
```

Towards the end of optimization, he shows that things get a *tiny bit weird*, and application of jump threading, CFG simplification, and some other unnamed passes keep *flipping* the IR between the exact same canonicalized and uncanonicalized versions seven times. Reinserting and subsequently threading the same dedicated exit wastes only compilation time and has no benefit on the ultimate output of the program. And this behavior occurs on a simple function

with a single simple loop. It is not hard to imagine that such waste increases with program size and complexity. This is exactly the behavior an incremental approach to optimization attempts to solve.

The list of transformations applied by LLVM 7.0.0 is given in Figure 11.01. This list contains nearly 300 optimization passes applied to every function of a program, 51 of which are passes dedicated to restoring some property of the IR for a future pass (highlighted in red). In future evaluations targeting multiple passes, this sequence of optimizations should be used as the baseline. This will provide a very common real-world phase order, and any decrease in optimization runtime would hint at a true savings for countless workloads.

Application Going further, it would be worth the experimental effort to modify transformation passes of LLVM itself in order to add preservation of LCSSA form, dominators, loops, loop-simplify to the passes described here. If this were to be done in such a way that entire repair passes were no longer necessary after a given transformation, this could be a big win. This was not attempted in the span of this work as LLVM is an *absolute beast*. The Waddle IR was designed as a simplified version of the LLVM IR, and therefore missing some features. These features would need to be carefully considered in order to build a correct incremental version of LLVM passes (e.g. landing pads for exception flow, stack and heap allocation, and arrays and object shapes).

Phaseless Optimizers The problem of ordering optimization phases is a well-studied one without a clear solution [5, 40–44, 63, 64]. Matthieu Queva [52] developed a compiler framework in 2007 that, instead of relying on a phase order that is fixed for all input programs, uses dataflow analysis as a heuristic for what phase to apply next to a given function or program. The idea is to greedily find the phase order for a particular program that will work better than any fixed phase order that was chosen orthogonally to the contents and

targetlibinfo	targetlibinfo	libcalls-shrinkwrap	memdep	basiccg	demanded-bits
tti	tti	loops	lazy-branch-prob	rpo-functionattrs	lazy-branch-prob
tbaa	tbaa	branch-prob	lazy-block-freq	globalopt	lazy-block-freq
scoped-noalias	scoped-noalias	block-freq	opt-remark-emitter	globaldce	opt-remark-emitter
assumption-cache-tracker	assumption-cache-tracker	lazy-branch-prob	gvn	basiccg	slp-vectorizer
profile-summary-info	profile-summary-info	lazy-block-freq	phi-values	globals-aa	opt-remark-emitter
forceattrs	forceattrs	opt-remark-emitter	basicaa	float2int	instcombine
inferattrs	inferattrs	pgo-memop-opt	aa	domtree	loop-simplify
ipsccp	ipsccp	basicaa	memdep	loops	lcssa-verification
called-value-propagation	called-value-propagation	aa	memcpyopt	loop-simplify	lcssa
globalopt	globalopt	loops	sccp	lcssa-verification	scalar-evolution
domtree	domtree	lazy-branch-prob	demanded-bits	lcssa	loop-unroll
mem2reg	mem2reg	lazy-block-freq	bdce	basicaa	lazy-branch-prob
deadargelim	deadargelim	opt-remark-emitter	basicaa	aa	lazy-block-freq
domtree	domtree	tailcallelim	aa	scalar-evolution	opt-remark-emitter
basicaa	basicaa	simplifycfg	loops	loop-rotate	instcombine
aa	aa	reassociate	lazy-branch-prob	loop-accesses	loop-simplify
loops	loops	domtree	lazy-block-freq	lazy-branch-prob	lcssa-verification
lazy-branch-prob	lazy-branch-prob	loops	opt-remark-emitter	lazy-block-freq	lcssa
lazy-block-freq	lazy-block-freq	loop-simplify	instcombine	opt-remark-emitter	scalar-evolution
opt-remark-emitter	opt-remark-emitter	lcssa-verification	lazy-value-info	loop-distribute	licm
instcombine	instcombine	lcssa	jump-threading	branch-prob	alignment-from-assumptions
simplifycfg	simplifycfg	basicaa	correlated-propagation	block-freq	strip-dead-prototypes
basiccg	basiccg	aa	basicaa	scalar-evolution	globaldce
globals-aa	globals-aa	scalar-evolution	aa	basicaa	constmerge
prune-eh	prune-eh	loop-rotate	phi-values	aa	domtree
inline	inline	licm	memdep	loop-accesses	loops
functionattrs	functionattrs	loop-unswitch	dse	demanded-bits	branch-prob
domtree	domtree	simplifycfg	loops	lazy-branch-prob	block-freq
sroa	sroa	domtree	loop-simplify	lazy-block-freq	loop-simplify
basicaa	basicaa	basicaa	lcssa-verification	opt-remark-emitter	lcssa-verification
aa	aa	aa	lcssa	loop-vectorize	lcssa
memoryssa	memoryssa	loops	basicaa	loop-simplify	basicaa
early-cse-memssa	early-cse-memssa	lazy-branch-prob	aa	scalar-evolution	aa
speculative-execution	speculative-execution	lazy-block-freq	scalar-evolution	aa	scalar-evolution
basicaa	basicaa	opt-remark-emitter	licm	loop-accesses	branch-prob
aa	aa	instcombine	postdomtree	loop-load-elim	block-freq
lazy-value-info	lazy-value-info	loop-simplify	adce	basicaa	loop-sink
jump-threading	jump-threading	lcssa-verification	simplifycfg	aa	lazy-branch-prob
correlated-propagation	correlated-propagation	lcssa	domtree	lazy-branch-prob	lazy-block-freq
simplifycfg	simplifycfg	scalar-evolution	basicaa	lazy-block-freq	opt-remark-emitter
domtree	domtree	indvars	aa	opt-remark-emitter	instsimplify
basicaa	basicaa	loop-idiom	loops	instcombine	div-rem-pairs
aa	aa	loop-deletion	lazy-branch-prob	simplifycfg	simplifycfg
loops	loops	loop-unroll	lazy-block-freq	domtree	verify
lazy-branch-prob	lazy-branch-prob	mldst-motion	opt-remark-emitter	loops	
lazy-block-freq	lazy-block-freq	phi-values	instcombine	scalar-evolution	
opt-remark-emitter	opt-remark-emitter	basicaa	barrier	basicaa	
instcombine	instcombine	aa	elim-avail-extern	aa	

Figure 11.01: The standard sequence of LLVM 7.0.0 -O2 passes (read by columns).

properties of the program.

Waddle is a solid first step in moving towards such an idea. If Waddle can be extended to preserve or repair a certain class of analysis results as well (e.g. loop trip count, branch probabilities, alias analysis, possible register value ranges), even approximately, then the heuristics described by this framework could be applied to choose the next optimization pass to perform dynamically.

This idea can be taken even further and can provide a very granular solution to the phase order problem. Instead of structuring an optimizer into passes that must first determine *where* in the program they are applicable (e.g. searching the program for loops containing an invariant switch value, comparing target function sizes against a max inline size), it can be structured as a single *generic* pass. This generic pass chooses the next optimization from a global priority list of transformations that have been found applicable, along with a score. This score is a heuristically chosen value that denotes the impact of the transformation on the function and takes into account things like code size, trip counts, reduction in number of instructions or branches, and how deep in a loop the transformation occurs. During each transformation, transformation *listeners* look at the changed parts of the graph and determine if there are new opportunities that can be added to the queue. This pass would eventually halt once the queue is empty, the maximum score of the queue is lower than some threshold, or a time limit is reached.

The methodology used to construct Waddle would also be key to implementing such an architecture. If each operation cannot guarantee that the internal representation is in canonical form, then each listener would need to re-canonicalize the program (inserting a canonicalization step in between every transformation - **not pass**) or risk failing to recognize a subsequent optimization opportunity.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 253–265, New York, NY, USA, 1973. ACM.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] F. E. Allen and J. Cocke. Graph theoretic constructs for program control flow analysis. Technical Report IBM Res. Rep. RC 3923, IBM T.J. Watson Research Center, 1972.

[4] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[5] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '04, pages 231–239, New York, NY, USA, 2004. ACM.

[6] Bowen Alpern, Roger Hoover, Barry K Rosen, Peter F Sweeney, and F Kenneth Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 32–42. Society for Industrial and Applied Mathematics, 1990.

[7] Bowen Alpern, Mark N Wegman, and F Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11. ACM, 1988.

[8] Stephen Alstrup, Dov Harel, Peter W Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–2132, 1999.

[9] Stephen Alstrup and Peter W. Lauridsen. A simple and optimal algorithm for finding immediate dominators in reducible graphs, 1996.

[10] C Scott Ananian. *The static single information form.* PhD thesis, Massachusetts Institute of Technology, 1999.

[11] S. Beamer, K. Asanović, and D. Patterson. The GAP Benchmark Suite. *ArXiv e-prints*, August 2015.

[12] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. Simple and efficient construction of static single assignment form. In *Compiler Construction*, pages 102–122. Springer, 2013.

[13] Preston Briggs, Keith D Cooper, Timothy J Harvey, L Taylor Simpson, et al. Practical improvements to the construction and destruction of static single assignment form. *Software-Practice and experience*, 28(8):859–882, 1998.

[14] Adam L Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R Westbrook. Linear-time pointer-machine algorithms for least common ancestors, mst verification, and dominators.

[15] Adam L Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(6):1265–1296, 1998.

[16] Adam L Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R Westbrook. Corrigendum: a new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):383–387, 2005.

[17] M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute update. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 274–284, New York, NY, USA, 1988. ACM.

[18] Larry Carter, Jeanne Ferrante, and Clark Thomborson. Folklore confirmed: reducible flow graphs are exponentially larger. In *ACM SIGPLAN Notices*, volume 38, pages 106–114. ACM, 2003.

[19] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 55–66, New York, NY, USA, 1991. ACM.

[20] Fred Chow, Sun Chan, Shin Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in ssa form. In *Compiler Construction*, pages 253–267. Springer, 1996.

[21] Richard Cole and Ramesh Hariharan. Dynamic lca queries on trees. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, pages 235–244, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[22] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4:1–10, 2001.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[24] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[25] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, June 1994.

[26] Harold N Gabow. *Data structures for weighted matching and nearest common ancestors with linking*.

[27] Loukas Georgiadis. *Linear-time algorithms for dominators and related problems*. 2005.

[28] Loukas Georgiadis, Giuseppe F Italiano, Luigi Laura, and Federico Santaroni. An experimental study of dynamic dominators. In *Algorithms–ESA 2012*, pages 491–502. Springer, 2012.

[29] Loukas Georgiadis and Robert E Tarjan. Finding dominators revisited. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 869–878. Society for Industrial and Applied Mathematics, 2004.

[30] Loukas Georgiadis, Renato F Werneck, Robert E Tarjan, Spyridon Triantafyllis, and David I August. Finding dominators in practice. In *Algorithms–ESA 2004*, pages 677–688. Springer, 2004.

[31] Joe Groff and Chris Lattner. Swift intermediate language: A high level ir to complement llvm. `https://llvm.org/devmtg/2015-10/slides/GroffLattner-SILHighLevelIR.pdf`, 2015.

[32] Sebastian Hack. *Interference graphs of programs in SSA-form*. Citeseer.

[33] D Harel. A linear algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 185–194, New York, NY, USA, 1985. ACM.

[34] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4):557–567, 1997.

[35] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977.

[36] Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Program. Lang. Syst.*, 19(6):1031–1052, November 1997.

[37] John B Kam and Jeffrey D Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, 1976.

[38] Kathleen Knobe and Vivek Sarkar. Array ssa form and its use in parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 107–120, New York, NY, USA, 1998. ACM.

[39] Donald E Knuth. An empirical study of fortran programs. *Software: Practice and Experience*, 1(2):105–133, 1971.

[40] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. *SIGPLAN Not.*, 38(7):12–23, June 2003.

[41] Prasad A Kulkarni, Stephen R Hines, David B Whalley, Jason D Hiser, Jack W Davidson, and Douglas L Jones. Fast and efficient searches for effective optimization-phase sequences. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(2):165–198, 2005.

[42] Prasad A Kulkarni, David Whalley, and Gary Tyson. Evaluating heuristic optimization phase order search algorithms. In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pages 157–169. IEEE, 2007.

[43] Prasad A. Kulkarni, David B. Whalley, Gary S. Tyson, and Jack W. Davidson. Exhaustive optimization phase order space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 306–318, Washington, DC, USA, 2006. IEEE Computer Society.

[44] Prasad A. Kulkarni, David B. Whalley, Gary S. Tyson, and Jack W. Davidson. In search of near-optimal optimization phase orderings. *SIGPLAN Not.*, 41(7):83–92, June 2006.

[45] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[46] Niko Matsakiss. Rust pull request #1211. `https://github.com/rust-lang/rfcs/blob/master/text/1211-mir.md`, 2015.

[47] Steven S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[48] Nikos Parotsidis and Loukas Georgiadis. Dominators in directed graphs: a survey of recent results, applications, and open problems. 2013.

[49] Konstantinos Patakakis, Loukas Georgiadis, and Vasileios A Tatsis. Dynamic dominators in practice. In *2011 Panhellenic Conference on Informatics*, pages 100–104. IEEE, 2011.

[50] Fernando Magno Quintao Pereira and Jens Palsberg. Ssa elimination after register allocation. In *International Conference on Compiler Construction*, pages 158–173. Springer, 2009.

[51] Paul W. Purdom, Jr. and Edward F. Moore. Immediate predominators in a directed graph [h]. *Commun. ACM*, 15(8):777–778, August 1972.

[52] Matthieu Stéphane Benoit Queva. *Phase-ordering in optimizing compilers*. PhD thesis, Technical University of Denmark, DTU, DK-2800 Kgs. Lyngby, Denmark, 2007.

[53] G Ramalingam and Thomas Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 287–296. ACM, 1994.

[54] Ganesan Ramalingam. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):175–188, 1999.

[55] John Regehr. How llvm optimizes a function. `https://blog.regehr.org/archives/1603`, 2018.

[56] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM, 1988.

[57] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. An efficient incremental algorithm for maintaining dominator trees and its application to ϕ-nodes update. ACAPS Technical Memo 77, 1994.

[58] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Incremental computation of dominator trees. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, pages 1–12, New York, NY, USA, 1995. ACM.

[59] Vugranam C Sreedhar, Guang R Gao, and Yong-Fong Lee. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(6):649–658, 1996.

[60] Bjarne Steensgaard. Sequentializing program dependence graphs for irreducible programs. 1993.

[61] Arthur Stoutchinin and Francois De Ferriere. Efficient static single assignment form for predication. In *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, pages 172–181. IEEE, 2001.

[62] Robert Tarjan. Testing flow graph reducibility. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 96–107. ACM, 1973.

[63] Sid-Ahmed-Ali Touati and Denis Barthou. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, pages 147–156, New York, NY, USA, 2006. ACM.

[64] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 204–215, Washington, DC, USA, 2003. IEEE Computer Society.

[65] Sebastian Unger and Frank Mueller. *Handling irreducible loops: Optimized node splitting vs. DJ-graphs*. Springer, 2001.

[66] Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

[67] Jianzhou Zhao. *Formalizing the SSA-based Compiler for Verified Advanced Program Transformations*. PhD thesis, University of Pennsylvania, 2013.

[68] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 427–440, New York, NY, USA, 2012. ACM.

CURRICULUM VITAE

Eric Fritz

Education

2018 - Ph.D. in Engineering, University of Wisconsin - Milwaukee.
Dissertation: *Waddle – Always-Canonical Intermediate Representation*
Supervised by Dr. John Boyland

2013 - M.S. in Computer Science, University of Wisconsin - Milwaukee.
Capstone: *Optimizing the RedPrairie Distance Cache*
Supervised by Dr. Christine Cheng

2011 - B.F.A. in Film, University of Wisconsin - Milwaukee.

Publications

2018 - Eric Fritz. *Maintaining Canonical Form After Edge Deletion.* Presented at ICOOOLPS 2018 (co-located with ECOOP 2018).

2017 - Eric Fritz and Tian Zhao. *Typing and Semantics of Asynchronous Arrows in JavaScript.* Science of Computer Programming; Volume 141 Issue C, 2017.

2016 - Eric Fritz, Jose Antony, and Tian Zhao. *Arrows in Commercial Web Applications.* Presented at HotWeb 2016.

2015 - Eric Fritz and Tian Zhao. *Type Inference of Asynchronous Arrows in JavaScript.* Presented at REBLS 2015 (co-located with SPLASH 2015).