Guaranteed Optimization: Proving Nullspace Properties of Compilers

Todd L. Veldhuizen and Andrew Lumsdaine

Indiana University, Bloomington IN 47401 USA
tveldhui@acm.org lums@osl.iu.edu

Abstract. Writing performance-critical programs can be frustrating because optimizing compilers for imperative languages tend to be unpredictable. For a subset of optimizations – those that simplify rather than reorder code – it would be useful to prove that a compiler reliably performs optimizations. We show that adopting a "superanalysis" approach to optimization enables such a proof. By analogy with linear algebra, we define the *nullspace* of an optimizer as those programs it reduces to the empty program. To span the nullspace, we define rewrite rules that de-optimize programs by introducing abstraction. For a model compiler we prove that any sequence of de-optimizing rewrite rule applications is undone by the optimizer. Thus, we are able to give programmers a clear mental model of what simplifications the compiler is guaranteed to perform, and make progress on the problem of "abstraction penalty" in imperative languages.

1 Introduction

In our experience developing high-performance numerical libraries for object-oriented languages [17, 14] we have found that industrial compilers fail to eliminate abstraction reliably. In fact, the best optimizing compilers, while achieving remarkably good performance, are capricious, achieving that performance only when some unpredictable phrasing of the program is presented. This paper reports on progress in devising a compiler structure that achieves guaranteed optimization for imperative languages.

Optimizing compilers map programs to programs. It is interesting to ask as one does in linear algebra: what is the nullspace (or kernel) of the mapping? Clearly optimizing compilers are not linear transforms, but the analogy is useful. The nullspace of a linear transform A is the set of vectors transformed to the zero vector: $null(A) = \{x \mid Ax = 0\}$. For compilers, a sensible choice of nullspace is the set of programs reduced by the optimizer to the empty program $\emptyset$. By $\emptyset$, we mean the syntactically minimal program generating the value 0; in C, it is the program `int main() { return 0; }`. We use the notation $\mathcal{O}[\![\cdot]\!] : \mathsf{Program} \rightarrow \mathsf{Program}$ for an optimizer, and define its nullspace to be $null(\mathcal{O}) = \{p \in \mathsf{Program} \mid \mathcal{O}[\![p]\!] = \emptyset\}$.

Program code may be loosely divided in two: "useful" code that implements desired semantics of the program, and "abstraction" code that does not change observable properties but is introduced to further software engineering goals such as encapsulation and modularity. For example, we might want our program to compute:

$$1 + 2$$

but for software engineering reasons we write:

$$\begin{array}{l}\mathsf{x} = \mathsf{new\ Integer}(1);\\ \mathsf{y} = \mathsf{new\ Integer}(2);\\ \mathsf{plus}(\mathsf{x}, \mathsf{y})\end{array}$$

The resulting performance hit is often called the *abstraction penalty* [15, 11, 10] and is a pressing concern for those attempting to meld good software engineering practice with high performance. Ideally, optimizing compilers would eliminate abstraction while preserving semantics. In other words, we would like the nullspace of compilers to encompass typical patterns of software engineering abstraction.

Carrying the analogy further, in linear algebra we have the property that if A is a linear transform, then $z \in \mathit{null}(A)$ implies $A(x + z) = Ax$; that is, adding something from the nullspace has no effect. How does one account for "adding" abstraction to a program? Here the analogy to linear algebra is weak, and we turn to rewrite systems.

Term rewriting systems have a long history of use in program optimization (e.g. [19, 4, 20]). We consider an application opposite to their usual use: as rules that de-optimize a program, introducing abstraction, rather than optimizing it. Here are two such rules, stated informally:

1. Replace any expression e with x, where x is a fresh variable, and insert $\mathsf{x} := e$ immediately before e.
2. Select a subset of local variables, allocate a record of sufficient size at an appropriate program point,[1] replace variable definitions with writes and variable uses with reads to appropriate slots of the record.

The above rules span rudimentary OOP-style encapsulation: applying them to "1" and "2", we can produce code resembling $\mathsf{x} = \mathsf{new\ Integer}(1)$; $\mathsf{y} = \mathsf{new\ Integer}(2)$.

We regard such de-optimizing rules as a basis for the desired nullspace; if p is a program, $\mathcal{O}[\![\cdot]\!]$ the optimizer, and $\{g^i\}$ are de-optimizing rules, we want $\mathcal{O}[\![g^k g^{k-1} \cdots g^1 p]\!] = \mathcal{O}[\![p]\!]$ for any sequence of rule applications $g^k g^{k-1} \cdots g^1$. This is our analogy to "additivity" in linear algebra.

To build an optimizer capable of undoing any application of such rules, it is necessary to avoid *phase ordering problems* – problems that arise when optimizers are constructed from discrete optimizing passes. We do this by adopting a

[1] To be exact: a program point that dominates all definitions and uses of the variables.

superanalysis approach in which all analyses are performed simultaneously, followed by a single transformation step. This approach is known to be more powerful than applying individual optimizations in sequence (even iteratively), since optimizations are synergistic ([22, 1]). It also enables a proof of the nullspace property: any sequence of rule applications is undone *by a single application of the optimizer*. The nullspace proof is an induction over rule applications: for each rule we consider the changes made to the analysis equations, show the resulting system of equations has a "consistent" solution, and that the transformation step erases the code added by the rule. We demonstrate the proof technique by proving the nullspace property for a small arithmetic language. In an accompanying technical report, we prove the nullspace property for a model optimizer of an imperative language; we summarize the results here.

This paper is not about a proof for proof's sake. Rather, the proof guides the design of the optimizer. To enable the nullspace proof one is obliged to make many changes to the structure of the optimizer; and once the proof is complete, one has an optimizer design with the desired property.

1.1 Related work and contributions

The notion of guaranteed optimization is well-known in the functional world. Compilers for functional languages often guarantee tail-call optimization; staged languages such as MetaML [16] guarantee that expressions annotated as static will be evaluated fully at compile time; deforestation [21] can reliably eliminate intermediate representations for certain expressions [6]. Sands [13] in his work on improvement theory shows that certain optimization algorithms offer strong guarantees. The use of rewrite rules for optimization can guarantee (in restricted cases) the existence of normal forms [5].

In the imperative world, the vagaries of effects, unruly control flow, and absence of the algebraic properties of pure functional languages make guaranteed optimization difficult. The contribution of our work is to propose the idea of a nullspace as a guiding principle for guaranteed optimization of imperative languages, and to introduce a proof technique for proving nullspace properties of compilers. We believe this to be a first step toward a practical theory of minimal normal forms for imperative programs.

2 Definitions

We adopt abstract interpretation-style lattices [2] in which $\perp$ is associated with "absent" and $\top$ with "conflicting information." This is opposite from the convention in data flow analysis. We use $\sqsubseteq$ for a partial order relation, $\sqcup$ for lattice join, and $\times$ for lattice direct product. A tuple of lattices $(D_1, \ldots, D_n)$ is understood to be the lattice $D_1 \times \ldots \times D_n$. We reserve $\wedge$ to mean logical conjunction.

Program analyses are often presented as special-purpose solvers that simultaneously build and solve lattice equations. To reason about analyses, we take the view that a program analysis builds a system of equations such as:

$$
\begin{aligned}
x_1 &= e_1(x_1, x_2, \ldots, x_n)\\
x_2 &= e_2(x_1, x_2, \ldots, x_n)\\
&\ \vdots \qquad\qquad \vdots\\
x_n &= e_n(x_1, x_2, \ldots, x_n)
\end{aligned}
$$

where the x_i are analysis variables and the e_i are monotone functions. We write $X = E(X)$ for the above system, where $X = (x_1, \ldots, x_n)$ and $E = (e_1, \ldots, e_n)$. We use the notation lfp E to mean the least fixpoint of the function E.

Definition 1. *A system of lattice equations S is a pair $S = (D, E)$ where $D = (D_1, \ldots, D_n)$ is a tuple of complete lattices and $E = (e_1, \ldots, e_n)$ is a tuple of monotone functions $e_i : D_1 \times \ldots \times D_n \to D_i$. For variables $X = (x_1, \ldots, x_n)$ with $x_i : D_i$, we define $X^\star(S) = (x_1^\star(S), \ldots, x_n^\star(S)) = \mathrm{lfp}\ E$ to be the solution (least fixpoint) of the system of equations $X = E(X)$.*

We will write $X^\star$ to mean $X^\star(S)$, and similarly for other parameters when they are apparent from context.

Definition 2. *An optimizer $\mathcal{O}$ is a pair $(\mathcal{A}, \mathcal{T})$ where:*

- *$\mathcal{A}$ is an analysis taking a program p and producing a system of lattice equations $S = (D, E)$ and a mapping M between program points and elements of E;*
- *$\mathcal{T}$ is a transformation taking a solution $X^\star(S)$, program p, mapping M, and producing a new program p'.*

and we write $\mathcal{O}[\![p]\!] = p'$.

We informally state the main theorem of this paper to establish our goal; the proper version is stated and proved in Section 6.2. We make forward reference to the de-optimizing rules, analysis and transform of our model compiler, defined in later sections.
Informal statement of Theorem 1. Let $G = \{g_1, \ldots, g_m\}$ be the set of rewrite rules of Section 4.2, $\mathcal{A}$ be the analysis of Section 4.3, $\mathcal{T}$ be the transformation of Section 5, and $\mathcal{O} = (\mathcal{A}, \mathcal{T})$. Let p be a program, and p' be a program derived from p by a finite number of applications of rules in G. Then $\mathcal{O}[\![p]\!] = \mathcal{O}[\![p']\!]$.

2.1 Structure of the paper

The remainder of this paper is devoted to proving Theorem 2 and is organized as follows. We give a simple example of optimizing arithmetic expressions (Section 3), which serves to introduce the proof technique. We then introduce an intermediate representation suitable for imperative languages (Section 4.1), and sketch a sample set of "de-optimizing" rules (Section 4.2). We present a model optimizer consisting of four simultaneous analyses: congruence, reachability, a backward escape analysis, and a transform analysis (Section 4.3). After the analysis equations are solved, a single transformation step produces the optimized program (Section 5). In Section 6 we introduce the proof technique and apply it to prove Theorem 2 for our model compiler and sample de-optimizing rules.

3 Example: optimization of arithmetic expressions

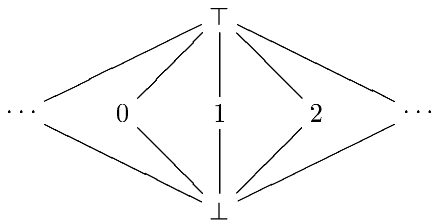
To introduce the proof technique we consider optimizing arithmetic expressions comprised of integers (n), variables (x) and addition:

$$e ::= n \mid \mathsf{x} \mid +(e, e)$$

We study two "de-optimizing" rewrite rules:

$$\begin{aligned} &g_1 : n \mapsto +(n_1, n_2) \quad \textit{where } n_1 + n_2 = n \\ &g_2 : e \mapsto +(e, 0) \end{aligned}$$

For example: $\mathsf{a} \overset{g_2}{\mapsto} +(\mathsf{a}, 0) \overset{g_1}{\mapsto} +(\mathsf{a}, +(4, -4)) \overset{g_1}{\mapsto} +(\mathsf{a}, +(+(1, 3), -4))$. We will define an optimizer, then prove that it undoes any sequence of applications of g_1 and g_2 in a single step. This is not an impressive result, but serves to introduce the proof technique. For this simple example, a single analysis suffices – constant propagation. We use the lattice:



The analysis $\mathcal{A}$ takes an expression and constructs (1) a system of analysis equations; and (2) a mapping between analysis variables and program points. We represent the mapping by annotating an expression with analysis variables, such as ${}^{c_0}+({}^{c_1}1, {}^{c_2}2)$. The analysis rules are:

Expression	**Equation**
${}^{c_0}n$	$c_0 = n$
${}^{c_0}\mathsf{x}$	$c_0 = \top$
${}^{c_0}+({}^{c_1}e_1, {}^{c_2}e_2)$	$c_0 = \hat{+}(c_1, c_2)$

where the abstract version of $\hat{+}$ is:

$$\hat{+}(x, y) \equiv \begin{cases} \perp & \textit{if } x = \perp \textit{ or } y = \perp; \textit{ else} \\ x + y & \textit{if } x \in \mathbb{Z} \textit{ and } y \in \mathbb{Z}; \textit{ else} \\ \top \end{cases} \tag{1}$$

A key requirement of our proof technique is that the analysis be *compositional*: the system of analysis equations for an expression is obtained by composing analyses of subexpressions. This implies that a rewrite on an expression induces a rewrite on the analysis equations in a straightforward way. Consider the single rewrite:

$$+(\mathsf{a}, 0) \overset{g_1}{\mapsto} +(\mathsf{a}, +(4, -4))$$

It is convenient to think of the rewrite in terms of its context and redex: in the above example we have the context $C = +(\mathsf{a}, [\,])$ and the redex 0, where $[\,]$ denotes a hole. The analysis of the two expressions is:

Mapping	${}^{c_0}+({}^{c_1}\mathsf{a}, {}^{c_2}0)$	${}^{c_0}+({}^{c_1}\mathsf{a}, {}^{c_2}+({}^{c_3}4, {}^{c_4}-4))$
Equations	$c_0 = \hat{+}(c_1, c_2)$ $c_1 = \top$ $c_2 = 0$	$c_0 = \hat{+}(c_1, c_2)$ $c_1 = \top$ $c_2 = \hat{+}(c_3, c_4)$ $c_3 = 4$ $c_4 = -4$

In the above table one can see how the term rewrite induces a rewrite on the analysis equations: the equations c_0, c_1 associated with the context $+(\mathsf{a}, [\,])$ are unchanged; the equation c_2 for the redex 0 has been altered, and new equations have been added for the subexpression $+(4, -4)$.

For each rewrite rule, the proof technique has two parts: (1) showing that the rewrite induced on the analysis equation has no effect on the solution to analysis variables associated with the context; and (2) showing that the transformation step eliminates the code added by the term rewrite.

For the arithmetic example, the transformation step $\mathcal{T}[\![\cdot]\!]$ is given by these rules:

$$\mathcal{T}[\![{}^{c_0}n]\!] = n \tag{2}$$

$$\mathcal{T}[\![{}^{c_0}\mathsf{x}]\!] = \mathsf{x} \tag{3}$$

$$\mathcal{T}[\![{}^{c_0}+({}^{c_1}e_1, {}^{c_2}e_2)]\!] = \begin{cases} c_0 & \textit{if } c_0 \sqsubset \top;\ \textit{else} \\ \mathcal{T}[\![e_1]\!] & \textit{if } c_2 = 0;\ \textit{else} \\ \mathcal{T}[\![e_2]\!] & \textit{if } c_1 = 0;\ \textit{else} \\ +(\mathcal{T}[\![e_1]\!], \mathcal{T}[\![e_2]\!]) & \end{cases} \tag{4}$$

The rule Eqn. (4) says: if an addition expression always has a constant value, replace it by that constant; otherwise, if one of the arguments is always zero, replace the addition by the other argument.

To prove that the optimizer undoes any sequence of rewrites in a single step, we use induction over the number of rewrites. Before we begin the proof, we define some terms.

3.1 Consistency of equations

We define a notion of *consistency* between systems of analysis equations. Intuitively, a system S_a is consistent with S_b if the system $X_b = E_b(X_b)$ has some extra variables compared to $X_a = E_a(X_a)$, but the variables they have in common map to the same program points and have the same lfp solution.

Definition 3. *Let $S_a = (D_a, E_a)$ and $S_b = (D_b, E_b)$ be two systems and $D_a = (D_1, \ldots, D_n)$. We say S_a and S_b are consistent and write $S_a \preceq S_b$ if for some permutation of applied to the elements of both D_b and E_b, $D_b = (D_1, \ldots, D_n, D_{n+1}, \ldots, D_m)$, $m \geq n$ and $x_i^\star(S_a) = x_i^\star(S_b)$ for $1 \leq i \leq n$ and the first n components of E_a and E_b map to the same program points.*

The consistency relation $\preceq$ is transitive and reflexive.

To guarantee the lfp is the same for common variables, we often need to reason about the fixpoint construction itself, i.e. the Kleene sequence $\perp, E(\perp)$, $E(E(\perp)), \ldots,$ lfp E. We define *ascending solution chains*, which generalize the Kleene sequence to arbitrary subterms of a system $X = E(X)$.

Definition 4. *Let $S = (D, E)$ be a system. The ascending solution chain of E is the set $\{X_i \in D \mid i = 0, 1, \ldots\}$ with $X_0 = \perp$, $X_{i+1} = E(X_i)$. The solution $X^\star = \mathrm{lfp}\ E$ is the greatest element of the ascending solution chain. For a function $f : D \to D'$ where D' is a complete lattice, we define $\mathrm{asc}\ f = \{f(X_i) \mid i = 0, 1, \ldots\}$.*

We have found two techniques useful for proving consistency. The first is to make use of results that show certain transformations on systems of equations preserve the fixpoint; such transformations have been studied systematically in [23]. We state three useful results here:

Lemma 1. *Let $S = (D, E)$ be a system of equations. The following rewrites on $X = E(X)$ result in a system $S' = (D', E')$ satisfying $S \preceq S'$:*

1. *Adding an equation $y = e_y(X)$, where y is a new variable and e_y is any monotone function;*
2. *For an equation $x_i = e_i(x_1, \ldots, x_n)$, choosing some subterm e' of e_i, adding an equation $y = e'$ and replacing the occurrence of e' in e_i with y, where y is a new variable;*
3. *For an equation $x_i = e_i(x_1, \ldots, x_n)$, replacing a subterm e' of e_i with $h(e')$, where h is an identity over the lattice of e' for the ascending solution chain of E.*

For brevity, when we say "h is an identity", we mean in the sense of the conditions of Case 3 of Lemma 1. The second technique for proving consistency is reason directly about the fixpoint of S'. In doing so, chaotic iteration [3, 7] – the principle that fixpoints can be constructed by updating variables in any order until convergence – is useful.

3.2 Proof for the arithmetic example

The nullspace proof is structured as follows: (1) a lemma for each rewrite rule; (2) a theorem that any number of rewrites is undone in a single step by the optimizer. The proof structure is highly modular: one can add rewrite rules by adding a new lemma, and the rest of the proof is unaffected.

For the arithmetic example, we consider rule g_1 first. From now on, when we write $\mathcal{T}[\![\cdot]\!]$, we imply that $\mathcal{T}$ is using the lfp solution to the analysis equations.

Lemma 2. *Let e be an expression, $(S, M) = \mathcal{A}[\![e]\!]$ and $(S', M') = \mathcal{A}[\![g_1 e]\!]$. Then $S \preceq S'$ and $\mathcal{T}[\![e]\!] = \mathcal{T}[\![g_1 e]\!]$.*

Proof. We consider the rewrites induced on the analysis equations by application of g_1:

$$^{c_0}n \mapsto {}^{c_0} + ({}^{c_1}n_1, {}^{c_2}n_2) \qquad \textit{where } n_1 + n_2 = n$$

This rewrite is applied to a subexpression inside some larger context; we write $X = E(X)$ for the unknown analysis equations associated with the rewrite context. The systems S and S' are:

S	S'
$X = E(X)$	$X = E(X)$
$c_0 = n$	$c_0 = \hat{+}(c_1, c_2)$
	$c_1 = n_1$
	$c_2 = n_2$

Consistency. In the system S, we have the equation $c_0 = n$, so clearly $c_0^\star = n$. In the system S', we have $c_1^\star = n_1$ and $c_2^\star = n_2$ and hence $c_0^\star = \hat{+}(n_1, n_2) = n_1 + n_2 = n$. Therefore $S \preceq S'$.

Transformation. To show $\mathcal{T}[\![e]\!] = \mathcal{T}[\![g_1 e]\!]$, we consider the context and redex of the rewrite separately. Since $S \preceq S'$, the analysis variables guiding transformation of the rewrite context will have the same solution in S and S'; therefore $\mathcal{T}[\![e]\!] = \mathcal{T}[\![g_1 e]\!]$ for the rewrite context. Next we consider the redex. In the original expression e we have $\mathcal{T}[\![^{c_0^\star}n]\!] = n$ by Eqn. (2). In the rewritten expression $g_1 e$, we have $\mathcal{T}[\![^{c_0^\star} + (n_1, n_2)]\!] = c_0^\star = n$ by Eqn. (4). Therefore $\mathcal{T}[\![e]\!] = \mathcal{T}[\![g_1 e]\!]$. □

Lemma 3. *Let e be an expression, $(S, M) = \mathcal{A}[\![e]\!]$ and $(S', M') = \mathcal{A}[\![g_2 e]\!]$. Then $S \preceq S'$ and $\mathcal{T}[\![e]\!] = \mathcal{T}[\![g_2 e]\!]$.*

Proof. We consider the rewrites induced on the analysis equations by application of g_2:

$$^{c_a}e \mapsto {}^{c_0} + ({}^{c_a}e, {}^{c_1}0)$$

Equations in the original system S which refer to c_a will refer to c_0 after rewriting. We use the notation $[c_0/c_a]$ to indicate this substitution:

S	S'
$X = E(X)$	$X = E(X)[c_0/c_a]$ $c_0 = \hat{+}(c_a, c_1)$ $c_1 = 0$

Consistency. In S' we have the equation $c_1 = 0$. Therefore $c_1^\star = 0$. From the definition of $\hat{+}$ (Eqn. (1)), we have $\hat{+}(c_a, 0) = c_a$. Using chaotic iteration, the fixpoint could be constructed by evaluating c_1 first; in subsequent iterations we could always follow an evaluation of c_a by an evaluation of c_0 to maintain the invariant $c_0 = c_a$. Therefore $c_0^\star = c_a^\star$, and $S \preceq S'$.

Transformation. Since $S \preceq S'$, $\mathcal{T}[\![e]\!] = \mathcal{T}[\![g_2 e]\!]$ for the rewrite context. For the redex, we consider cases over $c_a^\star$:

- Case $c_a^\star = \top$: then $c_0^\star = \top$ by Eqn. (1), and $\mathcal{T}[\![^{c_0^\star} + (^{c_a^\star}e, {}^{c_1^\star}0)]\!] = \mathcal{T}[\![^{c_a^\star}e]\!]$ by Eqn. (4).
- Case $c_a^\star \sqsubset \top$: then $\mathcal{T}[\![^{c_0^\star} + (^{c_a^\star}e, {}^{c_1^\star}0)]\!] = c_a^\star$ by Eqn. (4). Next we show $\mathcal{T}[\![^{c_a^\star}e]\!] = c_a^\star$ by cases over e:
  - Case $e = {}^{c_a}n$: then $c_a^\star = n$ and $\mathcal{T}[\![^{c_a^\star}n]\!] = n = c_a^\star$ by Eqn. (2).
  - Case $e = {}^{c_a}x$: then $c_a^\star = \top$ which contradicts the assumption $c_a^\star \sqsubset \top$, so this case does not apply;
  - Case $e = {}^{c_a} + (e_1, e_2)$: then $\mathcal{T}[\![^{c_a^\star} + (e_1, e_2)]\!] = c_a^\star$ by Eqn. (4).

Since $S \preceq S'$, the rewrite context will be transformed identically in both e and $g_2 e$; therefore $\mathcal{T}[\![e]\!] = \mathcal{T}[\![g_2 e]\!]$. □

We now prove that any sequence of rewrites is undone in a single step by the optimizer.

Theorem 1. *Let e be an expression, $G = \{g_1, g_2\}$ be the rewrite rules, $\mathcal{A}$ be the analysis, and $\mathcal{T}$ be the transformation. Let S be the analysis system for e. Let $g^n \cdots g^2 g^1$ be a sequence of rewrites, $g^i \in G$. Let S_n be the analysis system produced by $\mathcal{A}[\![g^n \cdots g^1 e]\!]$. Then $S \preceq S_n$ and $\mathcal{T}[\![g^n \cdots g^1 e]\!] = \mathcal{T}[\![e]\!]$.*

Proof. By induction over the number of rule applications. The base case of zero rule applications is trivial since $\preceq$ is reflexive and $\mathcal{T}[\![e]\!] = \mathcal{T}[\![e]\!]$. For the induction step, let e' be an expression and let S' be the analysis system for e'. Assume $S \preceq S'$ and $\mathcal{T}[\![e']\!] = \mathcal{T}[\![e]\!]$. We consider applying a single rewrite $g \in G$ to e'. Let S'' be the analysis system for ge'. By Lemma 2 or 3, $S' \preceq S''$ and $\mathcal{T}[\![ge']\!] = \mathcal{T}[\![e']\!]$. □

Corollary 1. *For any sequence of rewrites $g^n \cdots g^2 g^1$, $g^i \in G$ and any program p, $\mathcal{O}[\![g^n \cdots g^2 g^1 p]\!] = \mathcal{O}[\![p]\!]$.*

4 Nullspace proofs for imperative languages

4.1 An intermediate language

We now turn to a more substantial example: proving nullspace properties for an imperative language with exceptions and loops.

Top-level definitions	d ::=	`function` $v_0(v_1, \ldots, v_n)$ b	function definition
Block	b ::=	s ; $\cdots$; s; f	
Statement	s ::=	e	
		| $v := e$	variable definition
Final statement	f ::=	e	
		| `return` t	function return
		| `break`	break from loop
		| `throw` t	raise an exception
Expression	e ::=	t	trivial
		| p `(` $t, \ldots, t$ `)`	primitive
		| t `(` $t, \ldots, t$ `)`	function call
		| `if` t `then` b `else` b	if expression
		| `try` b `catch(`v`)` b	exception handling
		| `loop` b	loops
Trivial expression	t ::=	n	integer literal
		| v	variable use

Fig. 1: Grammar for the intermediate language.

Optimizing compilers typically lower the source form to a highly constrained intermediate language (IL) over which optimizations are performed. Figure 1 shows the intermediate language we consider. This IL retains high-level control structures to simplify the analyses and proof. Expressions are syntactically constrained to be in quadruple (e.g. [9]) or A-normal [12] form; this disallows nested expressions such as $\mathsf{f}(3+4)$ in favour of $\mathsf{a} := 3+4; \mathsf{f}(\mathsf{a})$. Control structures are those typical of structured imperative languages: loops, exceptions, and if expressions.

We distinguish two notions of dominance. By *static dominance*, we mean the dominance relation apparent in a pre-analysis control flow graph. By *analysis dominance*, we mean an "online" dominance relation determined by reachability analysis in which dead edges are considered removed from a control flow graph.

Some notes about the IL:

– Variables may only be defined once (statically), and the scope of a variable extends throughout the analysis dominance region of its definition. We assume that variables have been renamed by a global α-conversion pass to ensure uniqueness. To avoid dealing with assignments – a necessity for a real optimizer but beyond the scope of this paper – we assume they are handled by boxing.
– Primitive operations do not generate exceptions.
– if expressions use branch-if-zero tests.
– blocks, if, and try/catch may all generate values. This block produces the value 5:

```
a := if 0 then
        3
     else
        4
+(a,1)
```

4.2 De-optimizing rules

We consider the following set of simple de-optimizing rules. These rules are somewhat weak, deliberately so to make the proof of reasonable size. Each rule applies only if the resulting program would pass syntactic and semantic checks; when we say "b is any block", this is shorthand for "b is any syntactically valid block containing no variables unbound in the context in which it is to be placed." We define a *subblock* of a block $e_1; e_2; \ldots; e_n$ to be a non-empty sequence $e_i; \ldots; e_j$ with $1 \leq i \leq j \leq n$. We write $p \mapsto p'$ to indicate a single rewrite, and $p \overset{*}{\mapsto} p''$ to indicate a sequence of rewrites.

– Rule g_1: $\boxed{e \mapsto \mathsf{if\ 0\ then\ } e' \mathsf{\ else\ } e}$ where e is any subblock, and e' is a new block free of abruptions.[2]
– Rule g_2: $\boxed{\begin{array}{l} C[e] \mapsto \mathsf{x} := e; \\ \quad C[\mathsf{x}] \end{array}}$, where $C[e]$ is a statement decomposed into a context C and a subexpression e, the subexpression e is a trivial expression,[3] and x is a new globally unique identifier.
– Rule g_3: Insert at any statement position: $\boxed{\mathsf{x} := \mathsf{ref}(e)}$ where x is a new globally unique identifier and e is a trivial expression; and insert at $n \geq 0$ statement positions in the static dominance region of the definition of x: $\boxed{\mathsf{setref}(\mathsf{x}, e_i)}$ where the e_i are trivial expressions, $i = 1 \ldots n$. The primitive $\mathsf{ref}(e)$ allocates a box on the heap and stores in it the value of expression e; $\mathsf{deref}(e)$ dereferences a box, and $\mathsf{setref}(e_1, e_2)$ overwrites the value in a box.[4]

[2] This assumption is not necessary, but makes the proof more manageable.
[3] A trivial expression is either a variable use or a literal (Figure 1).
[4] This is a weak form of a "box any variable" rule, weakened to avoid dealing with propagation through boxes. Propagation through boxes is a complex analysis be-

– Rule g_4: $\boxed{e \mapsto \textsf{try}\ e\ \textsf{catch}(\textsf{x})\ e'}$ where e is any subblock free of function calls and throw statements, x is a new globally unique identifier, and e' is a new block free of abruptions.

By applying these rules to a program that computes $+(1, 2)$ we can obtain an elaborate version that has a debugging mode, boxes the integers, and handles exceptions:[5][6]

```
function main()        ↦*   function main()
  r := +(1, 2)                debugMode := 0
  return r                    a := 1
                              b := 2
                              x := ref(a)
                              y := ref(b)
                              if debugMode then
                                print(a)
                                print(b)
                              else
                                try
                                  r := +(a, b)
                                  return r
                                catch(err)
                                  g := deref(x)
                                  h := deref(y)
                                  print(g)
                                  print(h)
                                  exit(1)
```

Our optimizer is guaranteed to undo all the rule applications, effectively reversing the direction of the arrow $\overset{*}{\mapsto}$.

An obvious question is: why pose the rewrites as de-optimizing rules? Why not reverse the direction of the rewrites and apply the inverse rules directly to programs? The problem is this: deciding whether some of the inverse rules apply requires a supporting analysis. For example, deciding whether Rule g_3^{-1} applies to a given box requires knowing that any reads from the box are unreachable and that the box does not escape. Doing such analyses for every rewriting step would

yond the scope of this paper. Instead we allow any value to escape into a box, and guarantee that if the box is never read, then the boxing will be undone in the transformation step, and other optimizations will be performed as if the value did not escape.

[5] Throughout this paper we omit the statement separator ";" when it is apparent from indentation.

[6] To avoid propagation through boxes, we have manually applied this optimization, replacing uses of c and d with a and b, respectively. Also, in this example we assume print and exit are primitives.

be inefficient; and devising a rewriting strategy that would allow a convergence proof for a general compiler – with interprocedural analysis and heap analysis – appears difficult. Instead, we prove that our optimizer effectively subsumes a rewrite system implementing the inverse rules. If the the optimizer runs in $O(|p|\log|p|)$ time, then the rewrites are undone in $O(|p|\log|p|)$ time.

4.3 Analysis equations

We limit our attention to "simplifying" optimizations that reduce or eliminate code. Optimizations that reorder code – for example, loop nest optimizations – are interesting but we regard them as an orthogonal issue.

We formulate the analyses as an optimistic superanalysis [22, 1]. An optimistic analysis is one that starts with the optimistic assumption of $\perp$ and iterates to a least fixpoint. In an optimistic analysis, intermediate results during a fixpoint iteration are not conservative and may not be used for transformation. We adopt the terminology of [8] in calling simultaneous, interacting analyses a "superanalysis." We consider four simultaneous analyses:

- *Reachability* decides whether expressions may be reached during execution. Unreachable expressions are dead code and may be eliminated.
- *Congruence* partitions program points into equivalence classes based on whether the values reaching them are always the same.
- *Escape*, a backward analysis that determines how a value produced at a program point may be used.
- *Transform*, a simple analysis deciding whether subexpressions are to be processed for value or effect in the transformation step.

Our analyses are for the most part conventional and well-understood analyses drawn from the literature; we have made some adjustments to enable the proof. For the purposes of this paper, we regard soundness of the stated analyses as an important but tangential issue.

By analogy with logical implication, we adopt the notation $\Rightarrow$ in the usual way:

$$p(x) \Rightarrow y \quad \equiv \quad \begin{cases} \perp & \textit{if}\ p(x) = \mathsf{false} \\ y & \textit{if}\ p(x) = \mathsf{true} \end{cases} \tag{5}$$

where $p(x)$ is a predicate on an analysis variable x, and p is monotone on the lattice $\begin{array}{c}\mathsf{true}\\ | \\ \mathsf{false}\end{array}$. We use $\Rightarrow$ to encode interactions between analyses.

Throughout this paper we identify analysis variables corresponding to program points using superscripts: the notation ${}^{c_0}\mathsf{if}\ {}^{c_1}e_1\ \mathsf{then}\ {}^{c_2}e_2\ \mathsf{else}\ {}^{c_3}e_3$ asserts that c_1 is the congruence analysis variable corresponding to subexpression e_1, and similarly for c_2, c_3 and c_0.

4.4 Reachability analysis

We associate with every program point a reachability (flow) analysis variable f whose value is an element of the lattice:

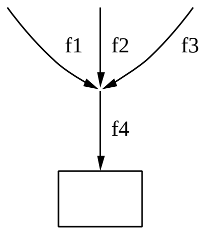
$$D_{\mathrm{f}} = \begin{array}{c} \mathsf{live} \\ | \\ \mathsf{dead} \end{array}$$

We define the predicate

$$\mathit{live}(f) \equiv (f \sqsupseteq_{\mathrm{f}} \mathsf{live}) \tag{6}$$

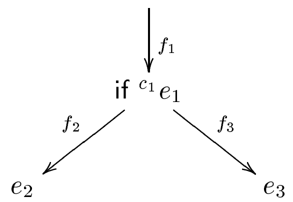
Reachability equations are built using control flow graphs, of which one is constructed for each function in a program. Each vertex in a flow graph represents a basic block – a sequence of statements in which control is guaranteed to flow sequentially from top to bottom. Edges represent control flow: if branches, return statements, exception throws, loops, and try/catch. For each function we have pads that are vertices containing no code but serve to collect return and throw edges.

A block is reachable if any of its incoming edges are live:



$$f_4 = f_1 \sqcup f_2 \sqcup f_3$$

To resolve control flow of if expressions, we use the results of congruence analysis (defined later), in the manner of conditional propagation [22]. Consider the expression $\mathsf{if}\ {}^{c_1}e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$ where c_1 is the congruence analysis variable for the test. The control flow subgraph is:



We resolve control flow by writing the equations:

$$f_2 = nonzero(c_1) \Rightarrow f_1$$
$$f_3 = zero(c_1) \Rightarrow f_1$$

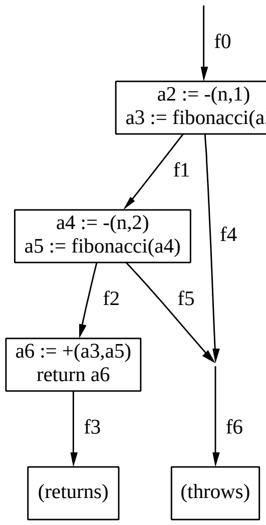
where:

$$zero(x) \equiv x \sqsupseteq_c 0 \tag{7}$$
$$nonzero(x) \equiv \begin{cases} \mathsf{true} \ \ \textit{if } x \sqsupseteq_c z \textit{ for some } z \in (\mathbb{Z} - \{0\}) \\ \mathsf{false} \ \textit{otherwise} \end{cases} \tag{8}$$

The predicates *zero* and *nonzero* determine whether their arguments *could* be zero or nonzero, respectively; if the branch cannot be determined either way then both predicates are true.
Straight-line code is split into multiple blocks at function calls. Consider this code:

$$\begin{aligned} \mathsf{a2} \ &:= -(\mathsf{n}, 1) \\ \mathsf{a3} \ &:= \mathsf{fibonacci}(a2) \\ \mathsf{a4} \ &:= -(\mathsf{n}, 2) \\ \mathsf{a5} \ &:= \mathsf{fibonacci}(a4) \\ \mathsf{a6} \ &:= +(a3, a5) \\ &\mathsf{return\ a6} \end{aligned}$$

Prior to analysis we do not know whether fibonacci() might throw an exception, so we split the sequence into three blocks and add edges to account for possible exception throws:



For this paper, we limit ourselves to intra-procedural analysis (rather than inter-procedural analysis) and assume that any reachable function call may complete normally or throw an exception. For the above code we generate the equations $f_1 = f_0$, $f_4 = f_0$, $f_2 = f_1$, $f_5 = f_1$, $f_3 = f_2$, $f_6 = f_4 \sqcup f_5$. We assume any function may be called, and hence for a function body with initial edge f_0 we add the equation $f_0 = \mathsf{live}$.

4.5 Congruence analysis

Congruence analysis partitions program points into equivalence classes based on whether the values reaching them are always the same. For simplicity, we present only the constant and copy propagation component of congruence analysis; we omit value-numbering and other techniques for finding congruences. The efficient way to do this analysis is via partitioning [?,?]; to facilitate the proof we instead propagate equivalence class representatives. Our congruence lattice has as its domain $D_c = \{\top, \bot\} \cup \mathbb{Z} \cup \mathsf{Vars}$ where $\mathbb{Z}$ is the set of integers, and Vars the set of all program variables. We associate with every value-producing program point i an analysis variable $c_i \in D_c$. An inference $c_i = j$ where $j \in \mathbb{Z}$ asserts that program point i always has the integer value j. An inference $c_i = \mathsf{x}$ where $\mathsf{x} \in \mathsf{Vars}$ asserts that program point i always has a value equal to the value of variable x; such an inference is only valid in the analysis dominance region of the definition of x. Our analysis ensures that such inferences do not escape their respective dominance regions. The partial order $\sqsubseteq_c$ is the least reflexive, transitive closure of:

$\bot \sqsubseteq_c y$	for all $y \in D_c$
$y \sqsubseteq_c \top$	for all $y \in D_c$
$i \sqsubseteq_c \mathsf{x}$	for all $\mathsf{x} \in \mathsf{Vars}$ and $i \in \mathbb{Z}$
$\mathsf{x}_1 \sqsubseteq_c \mathsf{x}_2$	if $\mathsf{x}_1, \mathsf{x}_2 \in \mathsf{Vars}$ and the definition of x_1 statically dominates the definition of x_2

If two program points i and j satisfy $c_i = c_j$ and $c_i \neq \top$, then i and j are in the same congruence equivalence class. Initially, all variables are assumed to be $\bot$, and hence all program points are in the same equivalence class; as analysis progresses, equivalence classes are effectively split until a fixpoint is reached.

This table summarizes analysis equations for straight-line code. We write f_a for the reachability analysis variable of the enclosing basic block:

Expression	Analysis variables	Equation
Function defn	function ${}^{c_0}\mathsf{f}({}^{c_1}\mathsf{x}_1, \ldots, {}^{c_k}\mathsf{x}_k)\ e$	$c_0 = \top, c_1 = \top, \ldots, c_k = \top$
Integer literal	${}^{f_a,c_0}n$	$c_0 = \mathit{live}(f_a) \Rightarrow n$
Variable		
- Definition	${}^{f_a,c_0}\mathsf{x} := {}^{c_x}e$	$c_0 = \mathit{live}(f_a) \Rightarrow \top$
- Use	${}^{f_a,c_1}\mathsf{x}$	$c_1 = \mathit{live}(f_a) \Rightarrow \mathit{putname}(\mathsf{x}, c_x)$
Ref	${}^{f_a,u_0,c_0}\mathsf{ref}({}^{c_1}e_1)$	$c_0 = \mathit{live}(f_a) \Rightarrow (\mathit{read}(u_0) \Rightarrow \top)$
Primitive	${}^{f_a,c_0}\mathsf{p}({}^{c_1}e_1, \ldots, {}^{c_k}e_k)$	$c_0 = \mathit{live}(f_a) \Rightarrow \hat{p}(c_1, \ldots, c_k)$
Function call	${}^{f_a,c_0}\ {}^{c_1}e_1({}^{c_2}e_2, \ldots, {}^{c_k}e_k)$	$c_0 = \mathit{live}(f_a) \Rightarrow \top$
Return	${}^{f_a,c_0}\mathsf{return}\ e$	$c_0 = \mathit{live}(f_a) \Rightarrow \top$
Throw	${}^{f_a,c_0}\mathsf{throw}\ e$	$c_0 = \mathit{live}(f_a) \Rightarrow \top$
Break	${}^{f_a,c_0}\mathsf{break}$	$c_0 = \mathit{live}(f_a) \Rightarrow \top$
Loop	${}^{f_a,c_0}\mathsf{loop}\ e$	$c_0 = \mathit{live}(f_a) \Rightarrow \top$

A function $\hat{p}$ is an abstract lattice approximation of a primitive operation p. The function $\mathit{putname}(\mathsf{x}, y)$ introduces a name x as an equivalence class representative if $y = \top$:

$$\mathit{putname}(\mathsf{x}, y) = \begin{cases} y & \mathit{if}\ y \neq \top \\ \mathsf{x} & \mathit{if}\ y = \top \end{cases} \tag{9}$$

The predicate $\mathit{read}(\cdot)$ tests whether a value might be a box that is read from at some future program point, and is described in Section 4.6. We use it here to support compile-time garbage collection: until the analysis discovers that a box created by ref could be read from, the ref propagates $\top$.

At control-flow joins due to if, return, throw, or try/catch incoming values are combined using a *merge* function (distinguished from the lattice operator $\sqcup$):

$$\mathit{merge}(a, b) = \begin{cases} a & \mathit{if}\ b = \top \\ b & \mathit{if}\ a = \top \\ a & \mathit{if}\ a = b \\ \top & \mathit{otherwise} \end{cases} \tag{10}$$

The expression ${}^{c_0}\mathsf{if}\ {}^{c_1}e_1\ \mathsf{then}\ {}^{c_2}e_2\ \mathsf{else}\ {}^{c_3}e_3$ would have the equation $c_0 = \mathit{merge}(c_2, c_3)$, where f_2 and f_3 are the reachability analysis variables of the control flow edges from the branches of the if. Similarly for an expression ${}^{c_0}\mathsf{try}\ {}^{c_1}e_1\ \mathsf{catch}({}^{c_x}x)\ {}^{c_2}e_2$: the congruence equations would be $c_0 = \mathit{merge}(c_1, c_2)$ and the equation for c_x would combine the congruence variables from all throw points of e_1 using the *merge* function.

4.6 Escape analysis

Escape analysis decides how a value may be used at future program points. In this paper we use it to support two simple transformations: eliminating dead variables and rudimentary compile-time garbage collection [?]. To support these transforms we need to know for every value (1) whether it is used; (2) whether it is read from as a box; (3) whether it is written to as a box. We associate with every value-producing program point an analysis variable u, and use as a lattice:

$$D_{\mathrm{u}} = \begin{array}{c}\mathsf{used}\\ |\\ \bot\end{array} \times \begin{array}{c}\mathsf{read}\\ |\\ \bot\end{array} \times \begin{array}{c}\mathsf{written}\\ |\\ \bot\end{array}$$

where $\times$ denotes lattice direct product. As shorthand we will write used to mean $(\mathsf{used}, \bot, \bot)$ and similarly for read and $\mathsf{written}$. We will write $\top$ for $(\mathsf{used}, \mathsf{read}, \mathsf{written})$ and $\bot$ for $(\bot, \bot, \bot)$. We define predicates:

$$used(u) \equiv u \sqsupseteq_{\mathrm{u}} \mathsf{used} \quad (11)$$
$$read(u) \equiv u \sqsupseteq_{\mathrm{u}} \mathsf{read} \quad (12)$$
$$written(u) \equiv u \sqsupseteq_{\mathrm{u}} \mathsf{written} \quad (13)$$

This table summarizes analysis equations for straight-line code:

Expression	Analysis variables	Equation
Function	$\mathsf{function}\ \mathsf{x}_0(\mathsf{x}_1, \ldots, \mathsf{x}_k)\ {}^{u_0}e_0$	$u_0 = \bot$
Block	${}^{u_a}\begin{bmatrix} {}^{u_1}e_1 \\ \vdots \\ {}^{u_{k-1}}e_{k-1} \\ {}^{u_k}e_k \end{bmatrix}$	$\begin{aligned} u_1 &= \bot \\ &\vdots \\ u_{k-1} &= \bot \\ u_k &= u_a \end{aligned}$
Variable		
- Definition	${}^{u_a}\mathsf{x} := {}^{u_0}e$	$u_0 = u_1 \sqcup \ldots \sqcup u_k$
- Uses	${}^{u_1}\mathsf{x}$ $\vdots$ ${}^{u_k}\mathsf{x}$	
Ref	${}^{f_a,u_a}\mathsf{ref}({}^{u_1}e_1)$	$u_1 = \mathit{live}(f_a) \Rightarrow (\mathit{read}(u_a) \Rightarrow \top)$
Deref	${}^{f_a,u_a}\mathsf{deref}({}^{u_1}e_1)$	$u_1 = \mathit{live}(f_a) \Rightarrow \mathsf{read}$
Setref	${}^{f_a,u_a}\mathsf{setref}({}^{u_1,c_1}e_1, {}^{u_2}e_2)$	$u_1 = \mathit{live}(f_a) \Rightarrow \mathsf{written}$ $u_2 = \mathit{live}(f_a) \Rightarrow ((c_1 \sqsupset_c \top) \Rightarrow \top)$
Other primitives	${}^{f_a,u_a}\mathsf{p}({}^{u_1}e_1, \ldots, {}^{u_k}e_k)$	$u_1 = u_a$ $\vdots$ $u_k = u_a$
Function call	${}^{f_a,u_a}\ {}^{u_1}e_1({}^{u_2}e_2, \ldots, {}^{u_k}e_k)$	$u_1 = \mathit{live}(f_a) \Rightarrow \mathsf{used}$ $u_2 = \mathit{live}(f_a) \Rightarrow \top$ $\vdots$ $u_k = \mathit{live}(f_a) \Rightarrow \top$
If	${}^{f_a,u_a}\mathsf{if}\ {}^{u_1}e_1\ \mathsf{then}\ {}^{u_2}e_2\ \mathsf{else}\ {}^{u_3}e_3$	$u_1 = \mathit{live}(f_a) \Rightarrow \top$
Loop	${}^{f_a,u_a}\mathsf{loop}\ {}^{u_1}e_1$	$u_1 = \top$

At a control-flow join, we add equations:

$$\begin{tikzcd} u_1 \arrow[dr, "f_1"'] & u_2 \arrow[d, "f_2"'] & \cdots & u_n \arrow[dll, "f_n"] \\ & u_0 & & \end{tikzcd}$$

$$\begin{aligned} u_1 &= \mathit{live}(f_1) \Rightarrow u_0 \\ u_2 &= \mathit{live}(f_2) \Rightarrow u_0 \\ &\vdots \\ u_n &= \mathit{live}(f_n) \Rightarrow u_0 \end{aligned}$$

At the return and throw pad for a function, we add equations:

$$u_0 = \mathit{live}(f_a) \Rightarrow \top$$

where f_a is the reachability of the return or escape pad.

4.7 Transform analysis

This is a helper analysis to make the transformation step simpler; it decides for every expression whether to process for value or effect. It associates with most program points an analysis variable t, and uses the simple lattice:

$$D_{\mathrm{t}} = \begin{array}{c}\textsf{value}\\ | \\ \textsf{effect}\end{array}$$

4.8 Transform analysis (details)

This table summarizes the analysis equations:

Expression	**Analysis variables**	**Equation**
Function	$\textsf{function } \textsf{x}_0(\textsf{x}_1,\ldots,\textsf{x}_k)\; {}^{t_0}e_0$	$t_0 = \textsf{effect}$
Block	${}_{t_a}\begin{bmatrix} {}^{t_1}e_1 \\ \vdots \\ {}^{t_{k-1}}e_{k-1} \\ {}^{t_k}e_k \end{bmatrix}$	$\begin{array}{ll} t_1 & = \textsf{effect} \\ \vdots & \\ t_{k-1} & = \textsf{effect} \\ t_k & = t_a \end{array}$
Var Def	$\textsf{x} := {}^{c_0,u_0,t_0}e$	$t_0 = (c_0 \sqsupseteq_{\mathrm{c}} \top) \wedge \mathit{used}(u_0) \Rightarrow \textsf{value}$
Return	$\textsf{return } {}^{t_0}e_0$	$t_0 = \textsf{value}$
Throw	$\textsf{throw } {}^{t_0}e_0$	$t_0 = \textsf{value}$
If	${}^{t_a}\textsf{if } e_1 \textsf{ then } {}^{t_2}e_2 \textsf{ else } {}^{t_3}e_3$	$t_2 = t_a, t_3 = t_a$
Try/catch	${}^{t_a}\textsf{try } {}^{t_1}e_1 \textsf{ catch(x) } {}^{c_2}e_2$	$t_1 = t_a, t_2 = t_a$
Loop	${}^{t_a}\textsf{loop } {}^{t_1}e_1$	$t_1 = \textsf{effect}$

4.9 Solution step

After the analysis equations are built, they are solved by constructing the least fixpoint solution. Our implementation does this in the usual way by constructing a dependence graph in which each analysis variable is a vertex and there is an edge (x_i, x_j) if x_j appears in the equation for x_i, building the acyclic condensation graph, and doing fixpoint iteration over sets of mutually dependent analysis variables using the worklist algorithm. This yields the solution $X^\star$ of the system $X = E(X)$.

5 Transformation step

These are the minimal transformations that allow the proof; a reasonable optimizer would do much more. We use $\Box$ to represent an empty expression. The transformation step implicitly takes as a parameter the solution $X^\star$ to the analysis equations; we omit it to reduce syntactic noise.

$$\mathcal{T}[\![{}^{c_0,t_0}n]\!] = \begin{cases} c_0 \textit{ if } t_0 \sqsupseteq_{\mathrm{t}} \mathsf{value} \\ \square \textit{ otherwise} \end{cases} \tag{14}$$

$$\mathcal{T}[\![{}^{c_0,t_0}\mathsf{x}]\!] = \begin{cases} c_0 \textit{ if } t_0 \sqsupseteq_{\mathrm{t}} \mathsf{value} \\ \square \textit{ otherwise} \end{cases} \tag{15}$$

$$\mathcal{T}[\![\mathsf{x} := {}^{t_0}e]\!] = \begin{cases} \mathsf{x} := \mathcal{T}[\![e]\!] \textit{ if } t_0 \sqsupseteq_{\mathrm{t}} \mathsf{value} \\ \mathcal{T}[\![e]\!] \qquad \textit{otherwise} \end{cases} \tag{16}$$

$$\mathcal{T}[\![\mathsf{if}\ e_1\ \mathsf{then}\ {}^{f_2}e_2\ \mathsf{else}\ {}^{f_3}e_3]\!] = \begin{cases} \mathcal{T}[\![e_3]\!] \qquad \textit{if live}(f_3) \wedge \neg \textit{live}(f_2) \\ \mathsf{if}\ \mathcal{T}[\![e_1]\!]\ \mathsf{then}\ \mathcal{T}[\![e_2]\!]\ \mathsf{else}\ \mathcal{T}[\![e_3]\!] \quad \textit{otherwise} \end{cases} \tag{17}$$

$$\mathcal{T}[\![\mathsf{try}\ e_1\ {}^{f_2}\mathsf{catch(x)}\ e_2]\!] = \begin{cases} \mathcal{T}[\![e_1]\!] \qquad \textit{if } \neg \textit{live}(f_2) \\ \mathsf{try}\ \mathcal{T}[\![e_1]\!]\ \mathsf{catch(x)}\ \mathcal{T}[\![e_2]\!]\ \textit{otherwise} \end{cases} \tag{18}$$

$$\mathcal{T}[\![{}^{c_0}\mathsf{ref}(e)]\!] = \begin{cases} \square \qquad \textit{if } c_0 = \bot \\ \mathsf{ref}(\mathcal{T}[\![e]\!])\ \textit{otherwise} \end{cases} \tag{19}$$

$$\mathcal{T}[\![\mathsf{setref}({}^{c_1}e_1, e_2)]\!] = \begin{cases} \square \qquad \textit{if } c_1 = \bot \\ \mathsf{setref}(\mathcal{T}[\![e_1]\!], \mathcal{T}[\![e_2]\!])\ \textit{otherwise} \end{cases} \tag{20}$$

For other expressions e not covered by the above cases, $\mathcal{T}[\![e]\!]$ is constructed by applying $\mathcal{T}[\![\cdot]\!]$ over subexpressions and reconstructing e in the natural way; for example, $\mathcal{T}[\![\mathsf{loop}\ e]\!] = \mathsf{loop}\ \mathcal{T}[\![e]\!]$. Two of the transforms require explanation:

- If the value of a trivial expression is needed, we do $\mathcal{T}[\![{}^{c_0}e]\!] = c_0$ (Eqns. (14,15)). This is possible since the congruence analysis variables at reachable trivial expressions are always either an integer value or a variable name (Lemma 5).
- For compile-time garbage collection, at a box creation point ${}^{c_0}\mathsf{ref}(e)$ we propagate $c_0 = \bot$ until escape analysis determines that the box may be read from. If the box is never read, then $c_0^\star = \bot$ and the $\mathsf{ref}(e)$ expression and all $\mathsf{setref}(\ldots)$ to the box are dead code, and are eliminated by Eqn. (19) and Eqn. (20).

6 Proof of the main theorem

Proofs of lemmas not given in the main text are found in Appendix A.2.

6.1 Structure of the proof

To prove Theorem 2, we consider each rule $g_i \in G$ in turn. Let $(S, M) = \mathcal{A}[\![p]\!]$ be the analysis equations for p, and $(S', M') = \mathcal{A}[\![g_i p]\!]$. We study the rewrites induced on S by g_i, and show that $S \preceq S'$. Therefore the gfp solution of S and S' are equal for their shared variables. We then consider the transformation $\mathcal{T}$ and prove sufficient bounds on analysis variables to show that $\mathcal{T}[\![p]\!] = \mathcal{T}[\![g_i p]\!]$.

We prove the lemma for Rule g_2 in this section; proofs for the remaining rules are found in Appendix A.2.

Lemma 4. (**Rule $\mathbf{g_1}$**) *Let p be a program, $(S, M) = \mathcal{A}[\![p]\!]$ and $(S', M') = \mathcal{A}[\![g_1 p]\!]$. Then $S \preceq S'$ and $\mathcal{T}[\![p]\!] = \mathcal{T}[\![g_1 p]\!]$.*

The following lemma is required in the proof for Rule g_2.

Lemma 5. *For a trivial subexpression ${}^{c_0}e$, $\top \notin \text{asc } c_0$.*

Lemma 6. (**Rule $\mathbf{g_2}$**) *Let p be a program, $(S, M) = \mathcal{A}[\![p]\!]$ and $(S', M') = \mathcal{A}[\![g_2 p]\!]$. Then $S \preceq S'$ and $\mathcal{T}[\![p]\!] = \mathcal{T}[\![g_2 p]\!]$.*

Proof. We consider the changes induced by g_2 on the analysis equations:

p	$g_2 p$
$C[{}^{f_a,c_a,u_a,t_a}e]$	${}^{f_a,c_2,u_2,t_2}\mathsf{x} := {}^{c_a,u_1,t_1}e$ $C[{}^{f_a,c_1,u_a,t_a}\mathsf{x}]$

The system S' is:

$$
\begin{aligned}
X &= E(X)[c_1/c_a, u_1/u_a] \\
c_1 &= \mathit{putname}(\mathsf{x}, c_a) \\
c_2 &= \mathit{live}(f_a) \Rightarrow \top \\
u_1 &= u_a \\
u_2 &= \bot \\
t_1 &= (c_a \sqsupseteq_{\mathrm{c}} \top) \wedge \mathit{used}(u_1) \Rightarrow \mathsf{value} \\
t_2 &= \mathsf{effect}
\end{aligned}
$$

We now show that each of the substitutions $[c_1/c_a, u_1/u_a]$ and added equations are consistent with S.

- For $[c_1/c_a]$: By application of Lemma 5, $\top \notin \text{asc } c_a$. We have the equation $c_1 = \mathit{putname}(\mathsf{x}, c_a)$; from the definition of *putname* (Eqn. (9), Appendix 4.5) and $\top \notin \text{asc } c_a$, $\mathit{putname}(\mathsf{x}, c_a)$ is an identity on c_a and Lemma 1(2,3) applies.
- For $[u_1/u_a]$: We have the equation $u_1 = u_a$. Lemma 1(2) applies.
- For the remaining equations, Lemma 1(1) applies.

Therefore $S \preceq S'$. Next we consider the transformations $\mathcal{T}[\![p]\!]$ and $\mathcal{T}[\![g_2 p]\!]$:

- For $\mathcal{T}[\![p]\!]$: we have $\mathcal{T}[\![{}^{c_a^\star,t_a^\star}e]\!] = c_a^\star$ if $t_a^\star \sqsupseteq_{\mathrm{t}} \mathsf{value}$ and $\Box$ otherwise (by either Eqn. (14) or Eqn. (15)).
- For $\mathcal{T}[\![g_2 p]\!]$: we have from previous arguments that $c_1^\star = c_a^\star$, since $\mathit{putname}(\mathsf{x}, c_a)$ is an identity. We have $t_1 = (c_a \sqsupseteq_{\mathrm{c}} \top) \wedge \mathit{used}(u_1) \Rightarrow \mathsf{value}$; since $c_a^\star \neq \top$ by Lemma 5, $t_1^\star \sqsubset_{\mathrm{t}} \mathsf{value}$. Therefore $\mathcal{T}[\![\mathsf{x} := {}^{t_1^\star}e]\!] = \mathcal{T}[\![{}^{t_1^\star}e]\!]$ by Eqn. (16), and $\mathcal{T}[\![{}^{t_1^\star}e]\!] = \Box$ by either Eqn. (14) or Eqn. (15). For the use of x, we have $\mathcal{T}[\![{}^{c_1^\star,t_a^\star}\mathsf{x}]\!] = c_1^\star = c_a^\star$ if $t_a^\star \sqsupseteq_{\mathrm{t}} \mathsf{value}$ and $\Box$ otherwise, by Eqn. (15).

Therefore $\mathcal{T}[\![p]\!] = \mathcal{T}[\![g_2 p]\!]$. $\square$

Lemma 7. *(**Rule $\mathbf{g_3}$**) Let p be a program, $(S, M) = \mathcal{A}[\![p]\!]$ and $(S', M') = \mathcal{A}[\![g_3 p]\!]$. Then $S \preceq S'$ and $\mathcal{T}[\![p]\!] = \mathcal{T}[\![g_3 p]\!]$.*

Lemma 8. *(**Rule $\mathbf{g_4}$**) Let p be a program, $(S, M) = \mathcal{A}[\![p]\!]$ and $(S', M') = \mathcal{A}[\![g_4 p]\!]$. Then $S \preceq S'$ and $\mathcal{T}[\![p]\!] = \mathcal{T}[\![g_4 p]\!]$.*

6.2 Proof of the main theorem

Theorem 2. *Let p be a program, $G = \{g_1, g_2, g_3, g_4\}$ be the set of rewrite rules of Section 4.2, $\mathcal{A}$ be the analysis of Section 4.3, $\mathcal{T}$ be the transformation of Section 5, and $\mathcal{O} = (\mathcal{A}, \mathcal{T})$. Let S be the analysis system for p. Let $g^n \cdots g^2 g^1$ be a sequence of rewrites, $g^i \in G$. Let S_n be the analysis system produced by $\mathcal{A}(g^n \cdots g^1 p)$. Then $S \preceq S_n$ and $\mathcal{T}[\![g^n \cdots g^1 p]\!] = \mathcal{T}[\![p]\!]$.*

Proof. By induction over the number of rule applications. The base case of zero rule applications is trivial since $\preceq$ is reflexive and $\mathcal{T}[\![p]\!] = \mathcal{T}[\![p]\!]$. For the induction step, let p' be a program and let S' be the analysis system for p'. Assume $S \preceq S'$ and $\mathcal{T}[\![p']\!] = \mathcal{T}[\![p]\!]$. We consider applying a single rewrite $g \in G$ to p'. Let S'' be the analysis system for gp'. By one of Lemmas 4, 6, 7 or 8, $S' \preceq S''$ and $\mathcal{T}[\![gp']\!] = \mathcal{T}[\![p']\!]$. $\square$

7 Nullspace compilers are true optimizers

In this section, we show a consequence of a nullspace proof: that under some weak assumptions, optimized programs are "minimal" in some metric.

We review some necessary rewrite terminology [4, 5]. We write $x \mapsto y$ to mean that x rewrites to y in a single step, and $\overset{*}{\mapsto}$ ("derives") for the reflexive, transitive closure of $\mapsto$. We write $x \leftrightarrow y$ ("convertible") to mean $x \mapsto y$ or $y \mapsto x$, and $\overset{*}{\leftrightarrow}$ for the reflexive, transitive closure of $\leftrightarrow$. Then $\overset{*}{\leftrightarrow}$ is an equivalence relation and partitions the set of programs into equivalence classes. We use the notation $[p]$ for the equivalence class of a program p under $\overset{*}{\leftrightarrow}$.

Remark. The nullspace theorem guarantees $\mathcal{O}[\![p]\!] = \mathcal{O}[\![p']\!]$ for all $p' \in [p]$.

We write $|p|$ for the textual size of a program. We make three assumptions:

1. Rewrites strictly increase size: $|p| < |gp|$ for a rewrite g and program p.
2. $\mathcal{O}$ is nonincreasing in program size: $|\mathcal{O}[\![p]\!]| \leq |p|$.
3. For any p, $\mathcal{O}[\![p]\!]$ is a fixpoint: $\mathcal{O}[\![\mathcal{O}[\![p]\!]]\!] = \mathcal{O}[\![p]\!]$.

The third assumption is reasonable for a transformation $\mathcal{T}[\![\cdot]\!]$ that makes the minimal changes to support the nullspace proof. However, we allow the possibility that more improvements are performed by $\mathcal{T}[\![\cdot]\!]$ than required by the nullspace proof.

Lemma 9. *Let p be a program. There is a nonempty set of "least programs" $P_0 \subseteq [p]$ such that (1) for every least program $p_0 \in P_0$ there is no $z \in [p]$ such that $z \mapsto p_0$; (2) p is derivable from some $p_0 \in P_0$.*

Proof. Since $|p| < |gp|$ for any rewrite g and program p, the relation $\overset{*}{\mapsto}$ is antisymmetric (i.e. there are no cycles $p_1 \mapsto p_2 \mapsto \ldots \mapsto p_n \mapsto p_1$). Furthermore, $\overset{*}{\mapsto}$ is reflexive and transitive by definition. Therefore $([p], \overset{*}{\mapsto})$ is a poset, and the least programs $P_0 \subseteq [p]$ are simply the least elements of the poset. $\square$

Definition 5. *The* abstraction level *of a program p, written $AL(p)$, is the minimum over all least programs $p_0 \in [p]$ of the number of rewrite steps in the shortest derivation $p_0 \mapsto \ldots \mapsto p$.*

Theorem 3. *For all programs p, $AL(\mathcal{O}[\![p]\!]) = 0$.*

Proof. (By contradiction). Choose p, and assume $AL(\mathcal{O}[\![p]\!]) > 0$. Then there is a program p' and rewrite g such that $gp' = \mathcal{O}[\![p]\!]$. Apply the optimizer again: $\mathcal{O}[\![gp']\!] = \mathcal{O}[\![p']\!] = \mathcal{O}[\![\mathcal{O}[\![p]\!]]\!] = \mathcal{O}[\![p]\!]$ by the nullspace theorem and the fixpoint assumption. But then $|\mathcal{O}[\![p]\!]| = |\mathcal{O}[\![p']\!]| \leq |p'| < |gp'| = |\mathcal{O}[\![p]\!]|$, which is a contradiction. $\square$

Thus, an optimizing compiler with the nullspace property and satisfying the above assumptions is a *true optimizer*: it produces a program which is minimal in the metric AL.

8 Conclusion

We have demonstrated a technique for proving guaranteed optimization for imperative compilers. The "de-optimizing" rules used by our proof technique give programmers an intuitive feel for what abstractions they may introduce without performance loss. The proof shows that any sequence of de-optimizing rule applications is undone by a single application of the optimizer; hence, the optimizer effectively subsumes a rewrite system implementing the inverse (optimizing) rules. The proof relies on adopting a superanalysis approach to optimization – that is, simultaneous analyses followed by a single transformation step.

A key unanswered question is: will the approach extend to more complex analyses and transformations? We believe so, within limits. We identify four requirements for adding to the framework: (1) the analysis must be posable as equations in a lattice framework; (2) the analysis equations must be composable – that is, if one assembles an expression from subexpressions, there is a corresponding way to assemble the analyses of subexpressions; (3) one must be able to describe suitable de-optimizing rewrite rules; and (4) the rewrite rules must admit a proof along the lines of Lemmas (2,3). We have explored – although not yet in a rigorous way – adding function pointers, interprocedural analysis and propagation of values through nonmutable heap objects, and these appear to meet the requirements. In future work we plan to incorporate them formally into our proof framework.

8.1 Acknowledgments

We thank Kenneth Chiuk, Kent Dybvig, Ron Garcia, Jaakko Järvi, Amr Sabry, Jeremy Siek and Jeremiah Willcock for their comments on drafts of this paper, Arch Robison for discussions, and the reviewers for helpful comments.

References

1. Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.
2. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Principles of Programming Languages*, pages 238–252, January 1977.
3. Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 1–12, 1977.
4. Nachum Dershowitz. A taste of rewrite systems. In *Proc. Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 199–228, 1993.
5. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. 1990.
6. A. J. Gill and S. L. Peyton Jones. Cheap deforestation in practice: An optimiser for Haskell. In Bjørn Pehrson and Imre Simon, editors, *Proceedings of the IFIP 13th World Computer Congress. Volume 1 : Technology and Foundations*, pages 581–586, Amsterdam, The Netherlands, August 28–September 1 1994. Elsevier Science Publishers.
7. Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206. ACM SIGACT and SIGPLAN, ACM Press, 1973.
8. Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. *ACM SIGPLAN Notices*, 31(1):270–282, January 2002.
9. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 2000.
10. Matthias Müller. Abstraction benchmarks and performance of C++ applications. In *Proceedings of the Fourth International Conference on Supercomputing in Nuclear Applications*, 2000.
11. Arch D. Robison. The abstraction penalty for small objects in C++. In *POOMA'96: The Parallel Object-Oriented Methods and Applications Conference*, Feb. 28 - Mar. 1 1996. Santa Fe, New Mexico.
12. Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3–4):289–360, 1993.
13. David Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.
14. Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A unifying framework for numerical linear algebra. In *Parallel Object Oriented Scientific Computing*. ECOOP, 1998.

15. Alex Stepanov. Abstraction penalty benchmark, 1994.
16. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. *ACM SIGPLAN Notices*, 32(12):203–217, 1997.
17. Todd L. Veldhuizen. Arrays in Blitz++. In *Computing in Object-Oriented Parallel Environments: Second International Symposium (ISCOPE 98)*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230. Springer-Verlag, 1998.
18. Todd L. Veldhuizen and Andrew Lumsdaine. Guaranteed optimization: Proving nullspace properties of compilers. Technical Report TR564, Indiana University Computer Science, 2002.
19. Eelco Visser. A survey of rewriting strategies in program transformation systems. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.
20. Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).
21. P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.
22. Mark N. Wegman and Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.
23. Jiawang Wei. Correctness of fixpoint transformations. *Theoretical Computer Science*, 129(1):123–142, June 1994.

A Lemmas

A.1 Statement of additional lemmas

Lemma 10. *Let D_1 and D_2 be complete lattices, $f : D_1 \times D_2 \to D_1$ and $g : D_1 \times D_2 \to D_2$ be monotone functions, $J(x,y) = (f(x,y), g(x,y))$, and $(x^\star, y^\star) = \text{lfp } J$. Let $K(x,y,z) = (f(x,y), g(z,y), x)$. Then $\text{lfp } K = (x^\star, y^\star, x^\star)$.*

Lemma 11. *Let $X = E(X)$ be a system S and ${}^{u_0}e$ be an expression whose normal completion edge has reachability f_1. Then the system S' given by*

$$\begin{aligned} X &= E(X)[u_2/u_0] \\ u_2 &= \mathit{live}(f_1) \Rightarrow u_0 \end{aligned}$$

satisfies $S \preceq S'$.

Lemma 12. *Let e be an expression whose reachability f_e satisfies $\text{asc } f_e = \{\mathsf{dead}\}$. If e' is an expression in the static dominance region of e with reachability, congruence and escape variables $f_{e'}$, $c_{e'}$ and $u_{e'}$, respectively, then $\text{asc } f_{e'} = \{\mathsf{dead}\}$, $\text{asc } c_{e'} = \{\bot\}$ and $\text{asc } u_{e'} = \{\bot\}$.*

Lemma 13. *If a system $X = E(X)$ contains a subterm $x \sqcup y$, then $\text{asc } (x \sqcup y) \subseteq \{\alpha \sqcup \beta \mid \alpha \in \text{asc } x \wedge \beta \in \text{asc } y\}$.*

Lemma 14. *If a system* $X = E(X)$ *contains an equation* $y = f(x)$ *then* $\text{asc } y = (\{\bot\} \cup f(\text{asc } x))$.

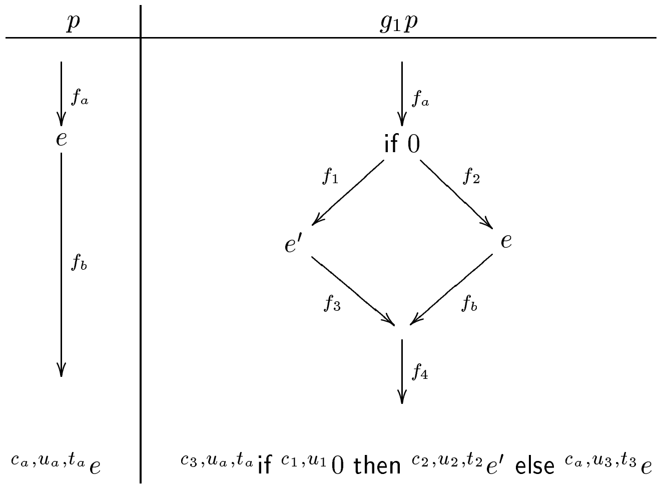
Lemma 15. *If a system* $X = E(X)$ *contains an equation* $y = f(x)$*, then* $\text{asc } y \subseteq (\{\bot\} \cup \text{range}(f))$.

Lemma 16. *If a system* $X = E(X)$ *contains a subterm* $f(x, y)$*, then* $\text{asc } f(x, y) \subseteq \{f(\alpha, \beta) \mid \alpha \in \text{asc } x, \beta \in \text{asc } y\}$.

A.2 Proofs of Lemmas

Proof. (**Lemma 1**) (1) The added variable y does not occur in E', so the least fixpointis unchanged for variables in X. (2) By application of Lemma 10. (3) If $h(\alpha) = \alpha$ for $\alpha \in \text{asc } e'$ then $E'(X) = E(X)$ during least fixpointconstruction, and the least fixpointis unchanged. □

Proof. (**Lemma 4**) We consider the changes induced by g_1 on the analysis equations $S = (X, E)$. For easier reading, we give newly introduced analysis variables numeric subscripts, and pre-existing analysis variables alphabetic subscripts. The table below shows the flow graph and analysis variables for the subexpression e before and after application of g_1.



In the block e' we may have uses of variables not local to e'. This causes additional terms to be inserted in the escape equations. For each variable used in e' but not defined in e', we have additions of the form:

Definition of x prior to e'	$\mathsf{x} := {}^{u_x}e_x$	$u_x = U_x(X) \sqcup u_i$
Use of x in e'	${}^{u_i}\mathsf{x}$	$u_i = U_i(X')$

where $U_x(X)$ is the previous escape function for x, and $U_i(X')$ is some unknown function that analyzes the use of x in e'. Let C, F, U, T be the congruence, flow, escape and transform functions for the analysis of p. Uses of variables in e' cause U to be rewritten to U' as described above.

Let X' be a set of analysis variables for $g_1 p$. The system S' is:

$$
\begin{aligned}
X &= (C(X), F(X), U'(X'), T(X))[c_3/c_a, f_2/f_a, f_4/f_b, u_3/u_a, t_3/t_a] \\
c_1 &= \mathit{live}(f_a) \Rightarrow 0 \\
c_2 &= C_2(X') \\
c_3 &= \mathit{merge}(c_2, c_a) \\
f_1 &= \mathit{nonzero}(c_1) \Rightarrow f_a \\
f_2 &= \mathit{zero}(c_1) \Rightarrow f_a \\
f_3 &= F_3(X') \\
f_4 &= f_3 \sqcup f_b \\
u_1 &= \mathit{live}(f_a) \Rightarrow \top \\
u_2 &= \mathit{live}(f_3) \Rightarrow u_a \\
u_3 &= \mathit{live}(f_b) \Rightarrow u_a \\
t_3 &= t_a \\
t_2 &= t_a
\end{aligned}
$$

where C_2 and F_3 are some congruence and flow functions.

We consider the C,F,U and T equations in turn. We show for the revised equations U' and for each of the substitutions $[c_3/c_a, f_2/f_a, f_4/f_b, u_3/u_a, t_3/t_a]$ that the resulting system is consistent, generally by showing that one of the cases of Lemma 2 applies.

- For the additions in U': We have the equation $c_1 = \mathit{live}(f_a) \Rightarrow 0$; by Lemma 15, asc $c_1 \subseteq \{\bot, 0\}$. We have the equation $f_1 = \mathit{nonzero}(c_1) \Rightarrow f_a$; by the definition of *nonzero* (Eqn. (8)), asc $f_1 = \{\mathsf{dead}\}$. By application of Lemma 12, each of the added equations $u_i = U_i(X')$ satisfy asc $u_i = \{\bot\}$; therefore the escape terms of the form $\cdot \sqcup u_i$ added to U' are identities, and Lemma 1(3) applies.
- For $[u_3/u_a]$: we have $u_3 = \mathit{live}(f_b) \Rightarrow u_a$. This is consistent by Lemma 11.
- For $[f_4/f_b]$: From previous arguments asc $f_1 = \{\mathsf{dead}\}$; by Lemma 12, asc $f_3 = \{\mathsf{dead}\}$. We have $f_4 = f_3 \sqcup f_b$; since asc $f_3 = \{\mathsf{dead}\}$, $f_3 \sqcup \cdot$ is an identity. Lemma 1(2,3) applies.
- For $[f_2/f_a]$: Since we have assumed that e completes normally, the equation for f_b is an identity on f_2. We have $f_2 = \mathit{zero}(c_1) \Rightarrow f_a$ and $c_1 = \mathit{live}(f_a) \Rightarrow 0$; by Lemma 10, a system containing the equation $f_2 = \mathit{zero}(c_1) \Rightarrow f_a$ has the same solution when the equation for f_2 is replaced with $f_2 = \mathit{zero}(\mathit{live}(f_a) \Rightarrow 0) \Rightarrow f_a$, which is an identity on f_a (Eqns. (5,6,7)). Lemma 1(2,3) applies.
- For $[c_3/c_a]$: We have $c_3 = \mathit{merge}(c_2, c_a)$. From previous arguments asc $f_1 = \{\mathsf{dead}\}$ and by application of Lemma 12, asc $c_2 = \{\bot\}$; therefore $\mathit{merge}(c_2, \cdot)$ is an identity (Eqn. (10)). Lemma 1(2,3) applies.

– For $[t_3/t_a]$: We have $t_3 = t_a$. Lemma 1(2) applies.

– For the remaining equations, Lemma 1(1) applies.

Therefore $S \preceq S'$.

We now consider the transformation step. From previous arguments asc $f_1 = \{\mathsf{dead}\}$; therefore $f_1^\star = \mathsf{dead}$, and $\mathcal{T}[\![\mathsf{if}\ 0\ \mathsf{then}\ {}^{f_1}e'\ \mathsf{else}\ {}^{f_2}e]\!] = \mathcal{T}[\![e]\!]$ by Eqn. (17).

Therefore $\mathcal{T}[\![p]\!] = \mathcal{T}[\![g_1 p]\!]$. □

Proof. (**Lemma 5**) Trivial expressions are constrained to be either integers or variable uses (Figure 1). Case e is an integer literal n: Then the analysis equation is $c_0 = n$, so asc $c_0 = \{\bot, n\}$. Case e is a variable use x: Then the analysis equation is $c_0 = \mathit{putname}(\mathsf{x}, c_i)$ where c_i is some analysis variable. By Lemma 15, asc $c_0 \subseteq \{\bot\} \cup \mathrm{range}(\mathit{putname})$. Since $\bot \notin \mathrm{range}(\mathit{putname})$ (Eqn. (9)), $\bot \notin$ asc c_0. □

Lemma 6. See Section 6.1.

Proof. (**Lemma 7**) We consider the changes induced by g_3 on the analysis equations $S = (X, E)$.

We label analysis variables for the i^{th} $\mathsf{setref}(\ldots)$ statement with a superscript i.

$$
\begin{array}{l}
{}^{f_a,c_3,u_3,t_3}\mathsf{x} := {}^{u_0,c_0,f_a,t_0}\mathsf{ref}({}^{c_1,u_1}e) \\
\vdots \\
{}^{f^1t^1}\mathsf{setref}({}^{c_1^1u_1^1}\mathsf{x}, {}^{c_2^1u_2^1}e_1) \\
\vdots \\
{}^{f^nt^n}\mathsf{setref}({}^{c_1^nu_1^n}\mathsf{x}, {}^{c_2^nu_2^n}e_n)
\end{array}
$$

For each expression e_i a term of the form $\cdot \sqcup u_2^i$ is added to the escape equation corresponding to e_i. For the expression e a term $\cdot \sqcup u_1$ is added to its escape equation. Let U' be the revised escape equations.

The system S' is:

$$
\begin{aligned}
X &= (C(X), F(X), U'(X'), T(X)) \\
c_0 &= \mathit{live}(f_a) \Rightarrow (\mathit{read}(u_0) \Rightarrow \top) \\
c_1^1 &= \mathit{live}(f_a) \Rightarrow \mathit{putname}(\mathsf{x}, c_0) \\
&\vdots \\
c_1^n &= \mathit{live}(f_a) \Rightarrow \mathit{putname}(\mathsf{x}, c_0) \\
c_3 &= \mathit{live}(f_a) \Rightarrow \top \\
u_0 &= u_1^1 \sqcup \ldots \sqcup u_1^n \\
u_1 &= \mathit{live}(f_a) \Rightarrow (\mathit{read}(u_0) \Rightarrow \top) \\
u_1^1 &= \mathit{live}(f^1) \Rightarrow \mathsf{written} \\
&\vdots \\
u_1^n &= \mathit{live}(f^n) \Rightarrow \mathsf{written} \\
u_2^1 &= \mathit{live}(f^1) \Rightarrow ((c_1^1 \sqsupset_{\mathrm{c}} \bot) \Rightarrow \top) \\
&\vdots \\
u_2^n &= \mathit{live}(f^n) \Rightarrow ((c_1^n \sqsupset_{\mathrm{c}} \bot) \Rightarrow \top) \\
u_3 &= \bot \\
t_0 &= (c_0 \sqsupseteq_{\mathrm{c}} \top) \wedge \mathit{used}(u_0) \Rightarrow \mathsf{value} \\
t^1 &= T_1(X') \\
&\vdots \\
t^k &= T_k(X') \\
t_3 &= \mathsf{effect} \\
f^1 &= F_1(X') \\
&\vdots \\
f^k &= F_k(X')
\end{aligned}
$$

where the $T_i(X')$ and $F_i(X')$ are some functions.

We have $u_1^i = \mathit{live}(f^i) \Rightarrow \mathsf{written}$, so asc $u_1^i \subseteq \{\bot, \mathsf{written}\}$ by Lemma 15. We have $u_0 = u_1^1 \sqcup \ldots \sqcup u_1^n$, so asc $u_0 \subseteq \{\bot, \mathsf{written}\}$ by repeated application of Lemma 13. We have $c_0 = \mathit{live}(f_a) \Rightarrow (\mathit{read}(u_0) \Rightarrow \top)$; by Lemma 14, asc $(\mathit{read}(u_0) \Rightarrow \top) = \{\bot\}$. We know $f_a \in \{\mathsf{dead}, \mathsf{live}\}$:

- Case $f_a = \mathsf{live}$: then asc $c_0 \subseteq \{\bot\}$ by Lemma 14.
- Case $f_a = \mathsf{dead}$: then asc $c_0 \subseteq \{\bot\}$ by Lemma 15.

Therefore asc $c_0 \subseteq \{\bot\}$.

We have $c_1^i = \mathit{live}(f_a) \Rightarrow \mathit{putname}(\mathsf{x}, c_0)$. By cases on f_a we have asc $c_1^i \subseteq \{\bot\}$. We have $u_2^i = \mathit{live}(f^i) \Rightarrow ((c_1^i \sqsupset_{\mathrm{c}} \bot) \Rightarrow \top)$; by cases on f^i we have asc $u_2^i \subseteq \{\bot\}$. Therefore the terms $\cdot \sqcup u_2^i$ are identities, and Lemma 1(3) applies.

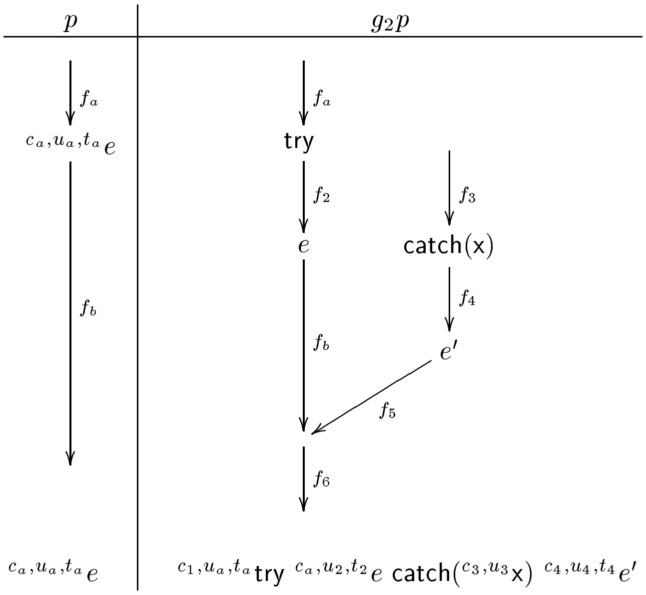
We have $u_1 = \mathit{live}(f_a) \Rightarrow \mathit{read}(u_0) \Rightarrow \top$; by cases on f_a, asc $u_1 \subseteq \{\bot\}$. Therefore the term $\cdot \sqcup u_1$ is an identity, and Lemma 1(3) applies.

Therefore $S \preceq S'$.

Next we consider the transformation step. We have from previous arguments that asc $c_0 = \{\bot\}$; therefore $c_0^\star = \bot$ and $t_0^\star = \mathsf{effect}$. Then $\mathcal{T}[\![\mathsf{x} := {}^{c_0^\star t_0^\star}\mathsf{ref}(e)]\!] = \mathcal{T}[\![{}^{c_0^\star t_0^\star}\mathsf{ref}(e)]\!] = \square$ by Eqn. (16) and Eqn. (19). For each $\mathsf{setref}({}^{c_1^1}\mathsf{x}, e_1)$ statement we have asc $c_1^i = \{\bot\}$ from previous arguments, and therefore ${c_1^i}^\star = \bot$. Therefore $\mathcal{T}[\![\mathsf{setref}({}^{{c_1^1}^\star}\mathsf{x}, e_1)]\!] = \square$ by Eqn. (20).

Therefore $\mathcal{T}[\![p]\!] = \mathcal{T}[\![g_3 p]\!]$. $\square$

Proof. (**Lemma 8**) We consider the changes induced by g_4 on the analysis equations $S = (X, E)$.



In the block e' we may have uses of variables not local to e'. This causes additional terms to be inserted in the escape equations. For each variable used in e' but not defined in e', we have additions of the form:

Definition of x prior to e'	$\mathsf{x} := {}^{u_x} e_x$	$u_x = U_x(X) \sqcup u_i$
Use of x in e'	${}^{u_i}\mathsf{x}$	$u_i = U_i(X')$

where $U_x(X)$ is the previous escape function for x, and $U_i(X')$ is some unknown function that analyzes the use of x in e'. Let C, F, U, T be the congruence, flow, escape and transform functions for the analysis of p. Uses of variables in e' cause U to be rewritten to U' as described above.

Since e was assumed to be free of function calls and throw statements, there are no throw points in e. The system S' is:

$$
\begin{aligned}
X &= (C(X), F(X), U'(X'), T(X))[c_1/c_a, f_2/f_a, f_6/f_b, u_2/u_a, t_2/t_a] \\
c_1 &= c_a \sqcup c_4 \\
c_3 &= \bot \\
c_4 &= C_4(X') \\
f_2 &= f_a \\
f_3 &= \mathsf{dead} \\
f_4 &= f_3 \\
f_5 &= F_5(X') \\
f_6 &= f_b \sqcup f_5 \\
u_2 &= \mathit{live}(f_b) \Rightarrow u_a \\
u_3 &= U_3(X') \\
u_4 &= U_4(X') \\
t_2 &= t_a \\
t_4 &= t_a
\end{aligned}
$$

where C_4, F_5, U_3 and U_4 are some functions. We show for the revised equations U' and for each of the substitutions $[c_1/c_a, f_2/f_a, f_6/f_b, u_2/u_a, t_2/t_a]$ that the result is consistent, generally by showing that one of the cases of Lemma 1 applies.

- For $[f_6/f_b]$: We have $f_3 = \mathsf{dead}$; therefore asc $f_3 = \{\mathsf{dead}\}$ and asc $f_4 = \{\mathsf{dead}\}$. By Lemma 12, asc $f_5 = \{\mathsf{dead}\}$. Therefore in the equation $f_6 = f_b \sqcup f_5$, $\cdot \sqcup f_5$ is an identity. Lemma 1(2,3) applies.
- For $[f_2/f_a]$: we have $f_2 = f_a$. Lemma 1(2) applies.
- For $[c_1/c_a]$: From previous arguments asc $f_3 = \{\mathsf{dead}\}$ and by Lemma 12, asc $c_4 = \{\bot\}$. Therefore in the equation $c_1 = c_a \sqcup c_4$, $\cdot \sqcup c_4$ is an identity and Lemma 1(2,3) applies.
- For $[u_2/u_a]$: We have the equation $u_2 = \mathit{live}(f_b) \Rightarrow u_a$. This is consistent by Lemma 11.
- For the additions in U': from previous arguments we have asc $f_4 = \{\mathsf{dead}\}$. By Lemma 12, each of the variables u_i corresponding to uses of variables in e' satisfy asc $u_i = \{\bot\}$. Therefore each of the added terms $\cdot \sqcup u_i$ are identities, and Lemma 1(3) applies.
- For $[t_2/t_a]$: We have the equation $t_2 = t_a$. Lemma 1(2) applies.
- For the other equations, Lemma 1(1) applies.

Therefore $S \preceq S'$.

We now consider the transformation step. From previous arguments asc $f_3 = \{\mathsf{dead}\}$; therefore $f_3^\star = \mathsf{dead}$, and $\mathcal{T}[\![\mathsf{try}\ e \overset{f_3^\star}{} \mathsf{catch(x)}\ e']\!] = \mathcal{T}[\![e]\!]$ by Eqn. (18).

Therefore $\mathcal{T}[\![p]\!] = \mathcal{T}[\![g_3 p]\!]$. □

Proof. (**Lemma 10**) By construction of the least fixpoint: let $(x_0, y_0) = (\bot, \bot)$, $(x_{i+1}, y_{i+1}) = J(x_i, y_i)$, $(x'_0, y'_0, z'_0) = (\bot, \bot, \bot)$ and $(x'_{i+1}, y'_{i+1}, z'_{i+1}) = K(x'_i, y'_i, z'_i)$. Then:

$$\begin{array}{rcccl} x_i & \sqsubseteq & z'_{i+1} & \sqsubseteq & x_{i+1} \\ x_i & \sqsubseteq & x'_i & \sqsubseteq & x_{i+1} \\ y_i & \sqsubseteq & y'_i & \sqsubseteq & y_{i+1} \end{array}$$

by induction over i and monotonicity of f, g, and therefore $\lim_{i\to\infty}(x'_i, y'_i, z'_i) = (x^\star, y^\star, x^\star)$. □

Proof. (**Lemma 14**) By construction of the least fixpoint. Consider the iteration $X_0 = \bot$, $X_{i+1} = E(X_i)$ and let (x_i, y_i) be the value of (x, y) in X_i. Then $y_0 = \bot$ and $y_{i+1} = f(x_i)$; therefore $\alpha \in \text{asc}\ x \Leftrightarrow f(\alpha) \in \text{asc}\ y$, and $\text{asc}\ y = (\{\bot\} \cup f(\text{asc}\ x))$. □

Proof. (**Lemma 15**) By Lemma 14, since $f(\text{asc}\ x) \subseteq \mathit{range}(f)$. □

Proof. (**Lemma 11**) We invoke soundness of the analysis. The value generated by e may be used if and only if the normal completion edge of e is live; therefore $\text{asc}\ (\mathit{live}(f_1) \Rightarrow u_0) = \text{asc}\ u_0$. □

Proof. (**Lemma 12**) Since e' is in the dominance region of e, we invoke soundness of the analysis to assert that $\text{asc}\ f_{e'} \subseteq \text{asc}\ f_e$.

For the congruence analysis variables, the proof is by bottom-up induction over expression trees. The base case is given by leaf expressions (literals, variables uses, and break). For an expression e_0 in the dominance region:

- Case $e_0 = {}^{c_0}n$. Then we have $c_0 = \mathit{live}(f_0) \Rightarrow n$. Since $\text{asc}\ f_0 \subseteq \{\mathsf{dead}\}$, $\text{asc}\ c_0 = \{\bot\}$ by Lemma 14.
- Case $e_0 = {}^{f_0,u_0,c_0}\mathsf{x}$. Then we have $c_0 = \mathit{live}(f_0) \Rightarrow \mathit{putname}(\mathsf{x}, c_1)$. Since $\text{asc}\ f_0 \subseteq \{\mathsf{dead}\}$, $\text{asc}\ c_0 = \{\bot\}$ by Lemma 14.
- Case $e_0 = {}^{f_0,c_0}\mathsf{x} := e_0$. Then we have $c_0 = \mathit{live}(f_0) \Rightarrow \top$. Since $\text{asc}\ f_0 \subseteq \{\mathsf{dead}\}$, $\text{asc}\ c_0 = \{\bot\}$ by Lemma 14.
- Case $e_0 = {}^{c_0}\mathsf{return}\ e_1$. Then we have $c_0 = \mathit{live}(f_0) \Rightarrow \top$. Since $\text{asc}\ f_0 \subseteq \{\mathsf{dead}\}$, $\text{asc}\ c_0 = \{\bot\}$ by Lemma 14.
- Case $e_0 = {}^{c_0}\mathsf{break}$. Then we have $c_0 = \mathit{live}(f_0) \Rightarrow \top$. Since $\text{asc}\ f_0 \subseteq \{\mathsf{dead}\}$, $\text{asc}\ c_0 = \{\bot\}$ by Lemma 14.
- Case $e_0 = {}^{c_0}\mathsf{throw}\ e_1$. Then we have $c_0 = \mathit{live}(f_0) \Rightarrow \top$. Since $\text{asc}\ f_0 \subseteq \{\mathsf{dead}\}$, $\text{asc}\ c_0 = \{\bot\}$ by Lemma 14.
- Case $e_0 = {}^{c_0}p({}^{c_1}e_1, \ldots, {}^{c_k}e_k)$. Then we have $c_0 = \mathit{live}(f_0) \Rightarrow \hat{p}(c_1, \ldots, c_k)$. Since $\text{asc}\ f_0 \subseteq \{\mathsf{dead}\}$, $\text{asc}\ c_0 = \{\bot\}$ by Lemma 14.
- Case $e_0 = {}^{c_0}\ {}^{c_1}e_1({}^{c_2}e_2, \ldots, {}^{c_k}e_k)$. Then we have $c_0 = \mathit{live}(f_0) \Rightarrow \top$. Since $\text{asc}\ f_0 \subseteq \{\mathsf{dead}\}$, $\text{asc}\ c_0 = \{\bot\}$ by Lemma 14.
- Case $e_0 = {}^{c_0}\mathsf{if}\ e_1\ \mathsf{then}\ {}^{c_2}e_2\ \mathsf{else}\ {}^{c_3}e_3$. Then we have $c_0 = \mathit{merge}(c_2, c_3)$. Since e_2 and e_3 are dominated by e_0, the induction hypothesis applies, so $\text{asc}\ c_2 = \{\bot\}$ and $\text{asc}\ c_3 = \{\bot\}$. By Lemma 14, $\text{asc}\ c_0 = \{\bot\}$.

- Case $e_0 = {}^{c_0}\mathsf{try}\ {}^{c_1}e_1\ \mathsf{catch(x)}\ {}^{c_2}e_2$. Then we have $c_0 = \mathit{merge}(c_1, c_2)$. Since e_1 and e_2 are dominated by e_0, the induction hypothesis applies and asc $c_1 = \{\bot\}$ and asc $c_2 = \{\bot\}$. By Lemma 14, asc $c_0 = \{\bot\}$.
- Case $e_0 = {}^{c_0}\mathsf{loop}\ e_1$. Then we have $c_0 = \mathit{live}(f_0) \Rightarrow \top$. Since asc $f_0 \subseteq \{\mathsf{dead}\}$, asc $c_0 = \{\bot\}$ by Lemma 14.

Therefore asc $c_{e'} = \{\mathsf{dead}\}$.

For the escape analysis variables, the proof is by top-down induction over expression trees. For a block ${}^{u_1}e_1; \ldots; {}^{u_{k-1}}e_{k-1}; {}^{u_k}e_k$ we have the equations:

$$\begin{aligned} u_1 &= \bot \\ &\vdots \\ u_{k-1} &= \bot \\ u_k &= u_a \end{aligned}$$

where u_a is the escape variable from the expression receiving the value of e_k. Clearly asc $u_i = \{\bot\}$ for $i = 1 \ldots k-1$, and $u_a = \bot$ by the induction hypothesis.

For an expression e_0 in the dominance region:

- Case $e_0 = \mathsf{x} := {}^{u_0}e_0$. We have $u_0 = u_1 \sqcup \ldots \sqcup u_k$ where the variables $u_1 \ldots u_k$ are associated with the uses of x.
- Case $e_0 = {}^{f_0}\mathsf{return}\ {}^{u_1}e_1$. The value of e_1 will go to a control-flow join at the return pad, and so we have the equation $u_1 = \mathit{live}(f_0) \Rightarrow u_a$ where u_a is the use at the return pad. Since asc $f_0 \subseteq \{\mathsf{dead}\}$, asc $u_1 = \{\bot\}$ by Lemma 14.
- Case $e_0 = {}^{f_0}\mathsf{throw}\ {}^{u_1}e_1$. The value of e_1 will go to a control-flow join at either the throw pad or a catch block, and so we have the equation $u_1 = \mathit{live}(f_0) \Rightarrow u_a$ where u_a is the use at the control-flow join. Since asc $f_0 \subseteq \{\mathsf{dead}\}$, asc $u_1 = \{\bot\}$ by Lemma 14.
- Case $e_0 = {}^{f_0,u_0}p({}^{u_1}e_1, \ldots, {}^{u_k}e_k)$. For each u_i, $1 \leq i \leq k$, we have an equation $u_i = u_0$. Since asc $u_0 = \{\bot\}$ by the induction hypothesis, asc $u_i = \{\bot\}$ for $1 \leq i \leq k$.
- Case $e_0 = {}^{f_0,u_0}\ {}^{u_1}e_1({}^{u_2}e_2, \ldots, {}^{u_k}e_k)$. We have $u_1 = \mathit{live}(f_a) \Rightarrow \mathsf{used}$. Since asc $f_0 \subseteq \{\mathsf{dead}\}$, asc $u_1 = \{\bot\}$ by Lemma 14. For the u_i, $2 \leq i \leq k$ we have equations $u_i = \mathit{live}(f_a) \Rightarrow \top$. Since asc $f_0 \subseteq \{\mathsf{dead}\}$, asc $u_i = \{\bot\}$ by Lemma 14.
- Case $e_0 = {}^{f_0}\mathsf{if}\ {}^{u_1}e_1\ \mathsf{then}\ {}^{u_2}e_2\ \mathsf{else}\ {}^{u_3}e_3$. We have $u_1 = \mathit{live}(f_0) \Rightarrow \top$. Since asc $f_0 \subseteq \{\mathsf{dead}\}$, asc $u_1 = \{\bot\}$ by Lemma 14. Assuming (conservatively) that e_2 and e_3 may complete normally, let f_2 and f_3 be the liveness of their normal completion edges. we will have an equation $u_i = \mathit{live}(f_i) \Rightarrow u_1$ for $i = 2, 3$. Since e_2 and e_3 are clearly dominated by e_0, asc $f_2 \subseteq$ asc f_0 and asc $f_3 \subseteq$ asc f_0. Therefore asc $u_2 = \{\bot\}$ and asc $u_3 = \{\bot\}$ by Lemma 14.
- Case $e_0 = {}^{f_0,u_0}\mathsf{try}\ {}^{u_1}e_1\ \mathsf{catch(x)}\ {}^{u_2}e_2$. Assuming (conservatively) that e_1 and e_2 may complete normally, let f_1 and f_2 be the liveness of their normal completion edges. We will have an equation $u_i = \mathit{live}(f_i) \Rightarrow u_0$ for $i = 1, 2$. Since e_1 and e_2 are clearly dominated by e_0, asc $f_1 \subseteq$ asc f_0 and asc $f_2 \subseteq$ asc f_0. Therefore asc $u_1 = \{\bot\}$ and asc $u_2 = \{\bot\}$ by Lemma 14.

– Case $e_0 = {}^{f_0,u_0}\mathsf{loop}\ {}^{u_1}e_1$. Then we will have an equation $u_1 = \bot$, so asc $u_1 = \{\bot\}$.

Therefore asc $u_{e'} = \{\mathsf{dead}\}$.

$\square$

Proof. (**Lemma 13**) By construction of the least fixpoint; let $X_0 = \bot$, $X_{i+1} = E(X_i)$, and let (x_i, y_i) be the value of (x, y) in X_i. Then $\{x_i \sqcup y_i \mid i = 0, 1, \ldots\} \subseteq \{x_i \sqcup y_j \mid i, j \in \{0, 1, \ldots\}\}$. $\square$