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: $\mathit{null}(A) = \{x \mid Ax = 0\}$. 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 $\mathcal{O}[\![\cdot]\!] : \mathsf{Program} \to \mathsf{Program}$ for an optimizer, and define its nullspace to be $\mathit{null}(\mathcal{O}) = \{p \in \mathsf{Program} \mid \mathcal{O}[\![p]\!] = \emptyset\}$.

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 $\bot$ 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 $X^\star(S) = (x_1^\star(S), \ldots, x_n^\star(S)) =$ 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 $\mathcal{O}[\![p]\!] = p'$.

3 Example: optimization of arithmetic expressions

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

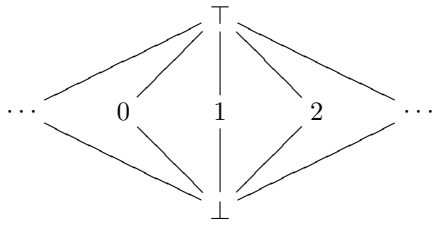
$$e ::= n \mid \mathsf{x} \mid +(e, e)$$

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:

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

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:

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

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

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

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 $\bot, E(\bot), E(E(\bot)), \ldots,$ 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 $X_0 = \bot$, $X_{i+1} = E(X_i)$. The solution*

$X^\star = \text{lfp}\ E$ *is the greatest element of the ascending solution chain. For a function* $f : D \to D'$ *where* D' *is a complete lattice, we define* $\text{asc}\ f = \{f(X_i) \mid i = 0, 1, \ldots\}$.

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*.*

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:

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

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

In the full version of this paper [18] we prove the nullspace property for a model optimizer of an imperative language. We summarize the language and the main results in this section.

4.1 An 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.

Top-level definitions	d ::=	`function` v_0`(`v_1`, ...,` 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, ..., t `)`	primitive
		| t `(` t, ..., 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.

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: $\boxed{\begin{array}{c} C[e] \mapsto \mathsf{x}\ := e; \\ C[\mathsf{x}] \end{array}}$, 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 $n \geq 0$

[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).

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]

– Rule g_4: $\boxed{e \mapsto \mathsf{try}\ e\ \mathsf{catch}(\mathsf{x})\ e'}$ 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]

```
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

[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 beyond 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] 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.

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 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 make some adjustments to enable the proof.

The proof follows the pattern developed in Section 3: a lemma for each rewrite rule, followed by a theorem proven inductively over the number of rewrites. The use of superanalysis – that is, simultaneous analyses – appears essential to proving consistency of systems of equations after a rewrite. We refer the reader to an expanded version of this paper for details [18].

5 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 4. *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. □

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 2. *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. □

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.

6 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.

6.1 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.