A Higher-Order Strategy for Eliminating Common Subexpressions

R. Daniel Resler\textsuperscript{a,}\textsuperscript{*}, Victor Winter\textsuperscript{b}

\textsuperscript{a}Computer Science Dept., Virginia Commonwealth University, Richmond, Virginia 23284 USA

\textsuperscript{b}Computer Science Dept., University of Nebraska at Omaha, Omaha, Nebraska 68182 USA

Abstract

Optimizing compilers often perform an operation known as common subexpression elimination to improve code efficiency. Typically this is accomplished either by pruning a directed acyclic graph to replace eliminated subexpressions by memory fetches of stored values or by using partial redundancy elimination, a data-flow analysis method. In this paper a higher-order strategic method is presented that rewrites expression trees to eliminate common subexpressions using equivalences in the lambda calculus. This approach offers several advantages—it is intuitive, transformations can be defined and applied within a high-level rewrite system, and it uses transformations for which correctness preservation can be proven.

Key words: program transformation, higher-order strategies, strategic programming, rewriting, common subexpression elimination, distributed data problem

2000 MSC: 68N15, 68N19, 68N20, 68Q42

1 Introduction and Motivation

Optimizing compilers often perform an operation known as common subexpression elimination to improve code efficiency. The goal of common subexpression elimination is to store the result of the evaluation of a repeated common (semantically identical) subexpression and then replace all other occurrences

\textsuperscript{*} Corresponding author.

Email addresses: dresler@vcu.edu (R. Daniel Resler), vwinter@mail.unomaha.edu (Victor Winter).
of this subexpression with a memory or register fetch of the stored value. Such an optimization can reduce execution time by eliminating both redundant computations and unnecessary multiple memory fetches of subexpression operands.

Typically local block-level common subexpression elimination is performed either by pruning a directed acyclic graph (DAG) to replace eliminated subexpressions by memory fetches of stored values [1], or, more recently, as part of partial-redundancy elimination [2,3], a data-flow analysis method used to determine which expressions can be safely moved and/or removed. In this paper a strategic method [4] of rewriting expression trees will be considered \(^1\); such a method offers several advantages over the traditional approaches:

- operations can be formally expressed in familiar mathematical notation
- it offers a more abstract, simpler solution—transformations can be defined and automatically applied in the domain-independent language of a high-level strategic rewrite system; such a system allows for the distinct (i.e. clear), high-level, and abstract specification of rewrite rules apart from the strategies that direct how they will be applied [7]
- it can be shown that program transformations preserve the correctness of the program being transformed

In this proof-of-concept research, we are interested in exploring a new approach to a common compiler optimization. Using syntax-tree rewrite systems to perform traditional high-level source-to-target language compilation is not a new concept. Our approach, however, is novel in two ways. First, it uses systematic and automatically applied scope capturing transformations on \(\lambda\)-expressions to reveal and remove common subexpressions. Secondly, these transformations take advantage of an elegant solution (using transient combinators and higher-order strategies in TL [8]) to the distributed data problem [9], i.e. the problem of bringing together semantically related terms from syntactically unrelated portions of the parse tree.

Transformations using higher-order strategic application of simple rewrite rules are also conducive to formal automatable verification [10,11]. In this article we present formal proofs that have been constructed by hand. While machine generated proofs are considered to provide a greater degree of assurance (see Section 10), hand generated proofs have also been used in practice to provide the basis of an assurance argument [12–15].

The remainder of this paper is organized as follows. After we present related work, a formal description of the elimination process is offered (Section 3). Section 4 defines the intermediate forms used in our transformations. A high-

\(^1\) A description of previous work that contributed to this approach can be found at [5,6].
level overview of the transformation system is given in Section 5. Sections 6 and 7 present the HATS transformation system and the implementation language TL. The transformations themselves along with proofs of their correctness are given in Section 8. We conclude the paper with a brief example (Section 9), a discussion of future work, and a summary.

2 Related Work

Knoop, Rüthing and Steffen [16] proposed transformations on flowgraphs that eliminated global code redundancies while at the same time suppressing any unnecessary code motion. Their method, called lazy code motion, initially moves all expressions as far up in the flowgraph as possible, thus exposing redundancies, before then pushing them as far down in the flow graph as possible in order to minimize register lifetimes. Our approach for local redundancy elimination more closely resembles lazy code motion than it does the traditional DAG-pruning approach previously sited. Rather than starting with a flow graph we begin with a parse tree representing a “nest” of λ-expressions (where each expression can be considered a form of three-address code) (cf. Section 4.1), then redundancies are eliminated by applying scope capturing transformations in a bottom-up left-to-right manner (Section 8.2) until a fixed point is reached. Such transformations eliminate all redundancies while at the same time increasing scope only as much as needed (i.e. variables are bound to values as close to their first use as possible).

This code movement of λ-expressions is similar to Jones, Partain, and Santos’ “let-floating” optimizations in a Haskell compiler [17,18] and Appel’s “let-hoisting” in CPS [19, chap. 8]. Both methods propose either increasing or decreasing the scope of let bindings in order to expose possible code optimizations. Whereas Jones et al. does not explicitly mention using their method to perform the elimination of common subexpressions, Appel does [19, chap. 9]. Neither approach however uses an automatic rewrite system to perform the code movement (Visser et al. [7], however, do implement a form of let-hoisting using a strategic rewrite system). Appel and Jim [20] also present a method to “shrink” λ-expressions by performing dead-variable elimination, constant folding, and a restricted beta rule that inlines only functions that are called once.

Using a formal rules-based rewrite system (as opposed to an algorithmic approach) to perform transformations for code optimization in a compiler is a well-known and often used approach. Lacey and de Moor [21] (see also [14,15]) apply rewrite rules with temporal logic side conditions to expression trees and control flow graphs to perform constant-propagation, dead-code elimination, and strength reduction optimizations.
Visser et al. [7] define a strategic rewrite system using Stratego [22] that performs let-hoisting, dead-code elimination, and function inlining on an ML-like functional language. Stratego is also used to perform code optimizations (function inlining, constant folding, constant propagation, and dead-code elimination) in Visser’s Tiger-in-Stratego compiler, “an experiment in compilation by transformation” [23]. Olsmos and Visser [24] present, without proof of correctness, a strategic method of eliminating common subexpressions that has been applied to the special-purpose GNU Octave programming language.

3 A Formal Characterization of the Common Subexpression Elimination Problem

The general problem of minimal length code generation for register-based machines (even single register machines) has long been known to be NP-complete [25,26]. In this paper, we consider a variation of the classical code generation problem. We wish to produce code in which (1) syntactically equivalent subexpressions are evaluated only once, and (2) where the number of registers in use during evaluation is minimized.

Specifically, we consider a framework where arithmetic expressions are evaluated using a semantic model where all computational steps are defined in terms of special variables called virtual registers or registers for short. In this framework, a register-based computation\(^2\) is expressed as a sequence \(\langle \ldots \rangle\) whose elements are register bindings (mappings) of the form \(r \mapsto expr\) where \(r\) denotes a (virtual) register and \(expr\) is either (1) a numeric or symbolic constant (i.e., an identifier) or (2) a unary/binary operation on registers. The length of a sequence of register bindings is defined as the number of register bindings it contains. Our virtual registers distinguish themselves from machine registers in two important ways: (1) the set of virtual registers that we can draw upon is unbounded, (2) a virtual register may only be assigned to once. Figure 1 gives some examples of arithmetic expressions and corresponding register-based computations.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>(\langle r_1 \mapsto 5 \rangle)</td>
</tr>
<tr>
<td>5 + x</td>
<td>(\langle r_1 \mapsto 5, r_2 \mapsto x, r_3 \mapsto r_1 + r_2 \rangle)</td>
</tr>
<tr>
<td>y + 7*y</td>
<td>(\langle r_1 \mapsto y, r_2 \mapsto 7, r_3 \mapsto y, r_4 \mapsto r_2 * r_3, r_5 \mapsto r_1 + r_4 \rangle)</td>
</tr>
</tbody>
</table>

Fig. 1. Register-based evaluation of arithmetic expressions

---

\(^2\) Register-based computations are equivalent to lists of three-address code.
(Note that register bindings and their \(\lambda\)-expression representations (introduced in Section 4) are a variation of SSA form [27]. In order to simplify the use of def-use chains in dataflow analysis, SSA form requires that each variable in a program have only one definition [28,3,29].)

Due to the presence of identifiers (e.g., \(x\) and \(y\)) an arithmetic expression can be seen as a function whose value is dependent upon a given program state. Correspondingly, register-based computations are also functions whose bound values are dependent upon program state. Here we model a program state as a function from identifiers (e.g., \(x\) and \(y\)) to values (e.g., 3 and 4). For example, \(st = \{(x, 3), (y, 4)\}\) denotes a program state in which the identifier \(x\) is bound to the value 3 and the identifier \(y\) is bound to the value 4. The value of the arithmetic expression \(5 + x\) with respect to the program state \(st\) is 8. Similarly, in the context of \(\langle r_1 \mapsto 5, r_2 \mapsto x, r_3 \mapsto r_1 + r_2\rangle\) the value of \(r_3\) with respect to the program state \(\{(x, 4)\}\) is 9.

We introduce a function \(eval\) to make explicit the dependency on program state that is shared by arithmetic expressions, register-based computations, and \(\lambda\)-expressions (which we use in a transformational setting to model both arithmetic expressions as well as register-based computations). The function \(eval\) is overloaded and is defined as follows:

- \(eval(e_{math}, st) \triangleq v\). When \(e_{math}\) is an arithmetic expression, the value \(v\) corresponds to the value obtained from evaluating \(e_{math}\) with respect to the program state \(st\). For example, \(eval(5 + x), \{(x, 3)\} = 8\).
- \(eval(e_{register}, st) \triangleq v\). When \(e_{register}\) is a register-based computation, the value \(v\) corresponds to the value of the last register binding in \(e_{register}\). In determining the value \(v\) the state \(st\) is used to obtain the value of all non-register variables. For example, \(eval(\langle r_1 \mapsto 5, r_2 \mapsto x, r_3 \mapsto r_1 + r_2\rangle, \{(x, 3)\}) = 8\).
- \(eval(e_{\lambda}, st) \triangleq v\). When \(e_{\lambda}\) is a \(\lambda\)-expression\(^3\) (e.g., \((\lambda r. r)(x))\), the value \(v\) corresponds to the value obtained by evaluating \(e_{\lambda}\) with respect to the program state \(st\). Here, the purpose of \(st\) is to assign numeric values to (i.e. bind) the free variables in \(e_{\lambda}\). For example, \(eval((\lambda r_1. (\lambda r_2. (\lambda r_3. r_3)(r_1 + r_2))(x))(5), \{(x, 3)\}) = 8\).

In this paper, we will use the symbol \(\equiv\) to denote semantic equivalence, the symbol \(=\) to denote (uninterpreted) syntactic equality, and the symbol \(\hat{=}\) to denote definitional abstraction (e.g., instead of let \(x \defeq 5\) we write let \(x \hat{=} 5\)).

**Definition 3.1** Let \(s_1\) and \(s_2\) denote terms of type (1) arithmetic expression,

\(^3\) In order to distinguish ordinary arithmetic expressions from expressions belonging to our extended \(\lambda\)-calculus, we use the term “\(\lambda\)-expression” to denote any expression belonging to the \(\lambda\)-calculus. In contrast, and when the distinction is important, we use the term “\(\lambda\)-abstraction” to denote a \(\lambda\)-expression of the form \(\lambda x. body\).
(2) register-based computation, or (3) λ-expression. The semantic equivalence of $s_1$ and $s_2$ is defined as follows:

$$(s_1 \equiv s_2) \leftrightarrow \forall st: \text{eval}(s_1, st) \equiv \text{eval}(s_2, st)$$

Semantic equivalence provides the underpinnings for optimizing register-based computations.

**Definition 3.2** Let $s_1$ and $s_2$ denote two register-based computations.

$$\text{as\_good\_as}(s_1, s_2) \leftrightarrow (s_1 \equiv s_2 \land \text{length}(s_1) \leq \text{length}(s_2))$$

**Definition 3.3** The optimality of a register-based computation $s$ with respect to a given arithmetic expression $e$ can be defined as follows:

$$\text{optimal}(s, e) \hat{=} s \equiv e \land (\forall s_i: (s \equiv s_i) \rightarrow \text{as\_good\_as}(s, s_i))$$

The broad class of problems relating to optimizations over register-based computations is traditionally referred to as *common subexpression elimination*. Consider the following (register-based) computations:

$$s_1 \hat{=} \langle r_1 \mapsto a, r_2 \mapsto b, r_3 \mapsto r_1 + r_2, r_4 \mapsto a, r_5 \mapsto b, r_6 \mapsto r_4 + r_5, r_7 \mapsto r_3 - r_6 \rangle$$

$$s_2 \hat{=} \langle r_1 \mapsto a, r_2 \mapsto b, r_3 \mapsto r_1 + r_2, r_4 \mapsto r_3 - r_3 \rangle$$

The relations $s_1 \equiv (a + b) - (a + b)$ and $s_1 \equiv s_2$ both hold. However, $s_2$ is more optimal than $s_1$.

It should be noted that optimality is a semantic property. However, for the transformations discussed in this paper, it is more appropriate to discuss optimality in syntactic terms.

**Definition 3.4** Let $t$ denote a term that is either a register-based computation, or a λ-expression. We say that $t$ is pseudo-optimal if and only if all subexpressions in $t$ are unique. In a more formal setting, it will denote pseudo optimality by the predicate $\text{pseudo\_optimal}(t)$.

Figure 2 shows two semantically equivalent but not syntactically identical arithmetic expressions and corresponding pseudo-optimal register-based computations.

The examples in Figure 2 suggest that the parenthesized form (i.e., the structure) of an arithmetic expression can play a significant role in the optimality of the resulting register-based computation. These examples also show that pseudo-optimality is not equivalent to the semantic-based optimality given
in Definition 3.3.

A variety of optimizations can be employed to minimize the difference between pseudo-optimal register-based computations and optimal register-based computations. For example, one can normalize arithmetic expressions before performing the kind of common subexpression elimination described in this paper. While the discussion of such normalization lies beyond the scope of this paper, its goal is to rewrite an arithmetic expression in such a manner that its common subexpressions are maximally exposed (e.g., the expression \((a + b) \ast (b + a)\) would be rewritten \((a + b) \ast (a + b)\) to aid the discovery of syntactically common subexpressions).

Another kind of optimization can be achieved by normalizing the register-based computations produced by common subexpression elimination. The goal here is to minimize the number of registers that are “in use” at any given time. Such optimizations are important to efficiently map the register-based computations described here onto computational models having finite registers (e.g., physical machines).

Due to the complexity surrounding the construction of register-based computations that minimize the number of registers “in use”, the common subexpression algorithm presented employs a heuristic-based approach. This approach is based on the idea of minimizing the use-def distance for each register over all registers in a register-based computation. In this context, the use-def distance of a register is the number of bindings spanning the definition of a register and its rightmost use in a given register-based computation. For example, in Equation 1 the use-def distance for the register \(r_1\) is 3.

\[
\langle r_1 \leftrightarrow 3, r_2 \leftrightarrow r_1 + r_1, r_3 \leftrightarrow r_1 + r_2 \rangle
\]  

Reducing the use-def total for all registers in a register-based computation
can be accomplished through a reordering driven by judicious application of the following commutative rewrite rule:

\[
    r_i \mapsto e_i, r_k \mapsto e_k \Rightarrow r_k \mapsto e_k, r_i \mapsto e_i \quad \text{if } r_i \notin e_k
\]

(2)

The classification of the problem of “reordering to minimize the total use-def distance” as belonging either to P or NP remains open at this time. The concept of minimal ordering based on use-def is formalized below.

**Definition 3.5** For a given computational sequence \( s \) containing a register \( r \), we define \( \text{ud dist}(r, s) \) as the length from the single (i.e., unique) binding of \( r \) (i.e., the definition of \( r \)) up to and including the rightmost use of \( r \) in \( s \).

Due to its simplicity, we will not formally describe how to compute \( \text{ud dist} \). Instead, Figure 3 gives two concrete examples showing the computed value of \( \text{ud dist} \) for various registers within a register-based computation. An interesting thing to note about the examples given in Figure 3 is that \( s_1 \equiv s_2 \).

\[
    s_1 = \langle r_1 \mapsto a, r_2 \mapsto b, r_3 \mapsto r_1 + r_2, r_4 \mapsto c, r_5 \mapsto r_4 + r_1, r_6 \mapsto d, \\
    r_7 \mapsto r_6 + r_1, r_8 \mapsto r_5 + r_7, r_9 \mapsto r_3 + r_8 \rangle
\]

\[ \text{ud dist}(r_1, s_1) = 7, \quad \text{ud dist}(r_2, s_1) = 2, \quad \text{ud dist}(r_3, s_1) = 7, \]

\[ \text{ud dist}(r_4, s_1) = 2, \quad \text{ud dist}(r_5, s_1) = 4, \quad \text{ud dist}(r_6, s_1) = 2, \]

\[ \text{ud dist}(r_7, s_1) = 2, \quad \text{ud dist}(r_8, s_1) = 2, \quad \text{ud dist}(r_9, s_1) = 0 \]

\[
    s_2 = \langle r_1 \mapsto c, r_2 \mapsto a, r_3 \mapsto r_1 + r_2, r_4 \mapsto d, r_5 \mapsto r_4 + r_2, r_6 \mapsto r_3 + r_5, \\
    r_7 \mapsto b, r_8 \mapsto r_2 + r_7, r_9 \mapsto r_6 + r_8 \rangle
\]

\[ \text{ud dist}(r_1, s_2) = 3, \quad \text{ud dist}(r_2, s_2) = 7, \quad \text{ud dist}(r_3, s_2) = 4, \]

\[ \text{ud dist}(r_4, s_2) = 2, \quad \text{ud dist}(r_5, s_2) = 2, \quad \text{ud dist}(r_6, s_2) = 4, \]

\[ \text{ud dist}(r_7, s_2) = 2, \quad \text{ud dist}(r_8, s_2) = 2, \quad \text{ud dist}(r_9, s_2) = 0 \]

Fig. 3. Examples of \( \text{ud dist} \) calculations for sequences \( s_1 \) and \( s_2 \) where \( s_1 \equiv s_2 \)

**Definition 3.6** The total distance of a register-based computation \( s \) is the sum of the \( \text{ud dist} \) over all registers in \( s \). Let \( s_n \) denote a register-based computation binding \( n \) (unique) registers \( r_1, \ldots, r_n \). The total distance of \( s_n \) is then defined as follows:

\[
    \text{total dist}(s_n) \doteq \sum_{i=1}^{n} \text{ud dist}(r_i, s_n)
\]
**Definition 3.7** A register-based computation $s_1$ has a **minimal ordering** if and only if its total distance is less-than or equal-to the total distance of every other register-based computation to which it is semantically equivalent.

$$\text{minimal\_ordering}(s_1) \equiv \forall s_2 : s_1 \equiv s_2 \rightarrow \text{total\_dist}(s_1) \leq \text{total\_dist}(s_2)$$

The transformations presented in this paper are an approximation to this minimization problem. We refer to this approximation as a **pseudo-minimal ordering** and in a formal context we will write $\text{pseudo\_minimal\_ordering}(t)$ to denote that a term satisfies this loosely defined property. What exactly constitutes a pseudo-minimal ordering is discussed in more detail in Section 8.

### 3.1 A Transformational Perspective

An arithmetic expression $e$ can be transformed into a corresponding (unoptimized) register-based computation using an induction on the structure of expressions:

1. If $e = c$ where $c$ is a constant, generate $\langle r \mapsto c \rangle$ where $r$ is a unique register. We say that $r$ is bound to the value of the expression $c$.
2. If $e = x$ where $x$ is a variable, generate $\langle r \mapsto x \rangle$ where $r$ is a unique register. We say that $r$ is bound to the value of the expression $x$.
3. Let $e_{\text{register}_1}$ and $e_{\text{register}_2}$ denote register-based computations that respectively correspond to arithmetic expressions $e_1$ and $e_2$. Let $r_1$ and $r_2$ respectively denote the names of the last registers bound in $e_{\text{register}_1}$ and $e_{\text{register}_2}$ (i.e. $r_1$ and $r_2$ are the registers bound to the values resulting from evaluating $e_1$ and $e_2$), and let $r$ denote a fresh (unique) register. Also, let $\circ$ denote a list-like concatenation operation on register-based computations. Under these assumptions, the arithmetic expression $e$ can be transformed into a register-based computation as follows:
   - (a) If $e = e_1 + e_2$ generate $e_{\text{register}_1} \circ e_{\text{register}_2} \circ \langle r \mapsto r_1 + r_2 \rangle$
   - (b) If $e = e_1 - e_2$ generate $e_{\text{register}_1} \circ e_{\text{register}_2} \circ \langle r \mapsto r_1 - r_2 \rangle$
   - (c) If $e = e_1 \times e_2$ generate $e_{\text{register}_1} \circ e_{\text{register}_2} \circ \langle r \mapsto r_1 \times r_2 \rangle$
   - (d) If $e = e_1/e_2$ generate $e_{\text{register}_1} \circ e_{\text{register}_2} \circ \langle r \mapsto r_1/r_2 \rangle$
   - (e) If $e = -e_1$ generate $e_{\text{register}_1} \circ \langle r \mapsto -r_1 \rangle$

The primary question in which we are interested is the following: “Given an arithmetic expression $e$, how can a pseudo-optimal computation $s$ having a pseudo-minimal ordering be obtained in a reliable and systematic fashion?” This paper describes a solution to this problem in which transformation is used to pass arithmetic expressions through a series of **canonical forms** which are modelled using familiar $\lambda$-expressions.
4 Modelling

Because they are well-suited for use in mechanized proofs and they can model both arithmetic expressions as well as register-based computations, we have chosen to represent arithmetic expressions as $\lambda$-expressions. This syntactic framework allows us to describe the common subexpression elimination problem in terms of transformations based on well-known definitions in the $\lambda$-calculus [30].

In our modelling framework, arithmetic expressions are modelled as $\lambda$-expressions that make explicit each computational step performed during the evaluation of the expression. Essentially, the $\lambda$-representation of an arithmetic expression corresponds to a fully parenthesized version of the expression where each parenthesized expression $e$ is associated with a corresponding identity $\lambda$-expression of the form $(\lambda i.i)e$. Figure 4 shows how the arithmetic expression $a + b \times c$ is modelled.

\[
\begin{align*}
  a + b \times c \\
  \Rightarrow \\
  (\lambda i_5.i_5)((\lambda i_4.i_4)a \\
  + \\
  (\lambda i_3.i_3)((\lambda i_2.i_2)b \\
  * \\
  (\lambda i_1.i_1)c))
\end{align*}
\]

Fig. 4. Modelling arithmetic expressions in the $\lambda$-calculus

Register-based computations are modelled as nested $\lambda$-expressions where $\lambda$-variables model registers, and the values bound to registers are modelled as arguments to $\lambda$-expressions. Let $s = \langle r_1 \mapsto e_1, ..., r_n \mapsto e_n \rangle$ denote an arbitrary register-based computation. The computation $s$ can be modelled as a $\lambda$-expression as shown in Figure 5.

4.1 Canonical Forms

Our approach to transforming arithmetic expressions is based on three canonical forms: $cf_0$, $cf_1$ and $cf_2$. The canonical form $cf_0$ represents the input domain to our transformation and its elements are terms denoting arithmetic expressions. The forms $cf_1$ and $cf_2$ each correspond to a class of $\lambda$-expressions having distinct structural properties. As an example, Figure 7 shows the canonical
\[ \langle r_1 \mapsto e_1, r_2 \mapsto e_2, \ldots, r_n \mapsto e_n \rangle \]
\[ \Rightarrow (\lambda r_1. (\lambda r_2. \ldots (\lambda r_n. r_n) e_n \ldots) e_2) e_1 \]

Fig. 5. Modelling register-based computations in the \( \lambda \)-calculus

forms \( cf_1 \) and \( cf_2 \) for the expression \( a + (-b \ast c - a) \) which is shown as an abstract parse tree in Figure 6.

Fig. 6. Abstract parse tree for expression \( a + (-b \ast c - a) \)

Informally, our first transformational step is to rewrite arithmetic expressions into corresponding \( \lambda \)-expressions (form \( cf_1 \)). Consider the abstract expression tree representation for our example in Figure 6. An identity \( \lambda \)-expression is generated for each sub-expression as each node is visited using a bottom-up, left-to-right traversal of the tree. More complex sub-expressions are composed as the traversal progresses finally resulting in form \( cf_1 \). The process used to arrive at form \( cf_2 \) is discussed in Section 5.1.

In the definitions that follow, predicates are formally defined that can be used to decide whether an arbitrary term is in canonical form \( cf_0 \), \( cf_1 \) or \( cf_2 \). The definitions of these predicates require the ability to distinguish between registers, arithmetic constants and variables. These distinctions are articulated using two predicates \( \text{atom} \) and \( \text{register} \) whose definitions are assumed but not formally defined. In particular, \( \text{atom}(e) \) is true if and only if \( e \) is either an arithmetic constant (e.g., an integer) or a variable (e.g., \( x \)). Similarly, \( \text{register}(r) \)
is true if and only if \( r \) is a register (i.e., in our model, a \( \lambda \)-bound variable).

In the definitions that follow, we assume that \( \text{bop} \in \{+, -, *, /\} \) denotes a binary arithmetic operation.

**Definition 4.1** We introduce a predicate \( \text{CF}_0 \) to assert that a term is in canonical form \( \text{cf}_0 \) (i.e., \( t \) is an arithmetic expression). For a given term \( t \), \( \text{CF}_0(t) \) can be deduced using the following rules:

<table>
<thead>
<tr>
<th>( \text{atom}(t) )</th>
<th>( \text{CF}_0(t_1) )</th>
<th>( \text{CF}_0(t_2) )</th>
<th>( \text{CF}_0(t) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{CF}_0(t) )</td>
<td>( \text{CF}_0(t_1 \text{ bop } t_2) )</td>
<td>( \text{CF}_0(-t) )</td>
<td></td>
</tr>
</tbody>
</table>

**Definition 4.2** We introduce a predicate \( \text{WFX} \) to assert that an argument to a \( \lambda \)-expression is well-formed. For a given argument \( e \), \( \text{WFX}(e) \) can be deduced using the following rules:

<table>
<thead>
<tr>
<th>( \text{atom}(a) )</th>
<th>( \text{register}(r_1) )</th>
<th>( \text{register}(r_2) )</th>
<th>( \text{register}(r) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{WFX}(a) )</td>
<td>( \text{WFX}(r_1 \text{ bop } r_2) )</td>
<td>( \text{WFX}(-r) )</td>
<td></td>
</tr>
</tbody>
</table>

**Definition 4.3** We introduce a predicate \( \text{CF}_1 \) to assert that a \( \lambda \)-expression,
containing no free variables, is in canonical form \( cf_1 \). For a given \( \lambda \)-expression \( t \), \( CF_1(t) \) can be deduced using the following rules:

\[
\begin{array}{c}
\text{atom}(a) \\
\overline{\text{CF}_1\left( (\lambda r.r)(a) \right)} \\
\text{CF}_1\left( (\lambda r. t)(e) \right) \\
\overline{\text{CF}_1\left( (\lambda r. (\lambda r. t)(-r_2))(e) \right)}
\end{array}
\]

\[
\begin{array}{c}
\text{CF}_1(t_1) \\
\overline{\text{CF}_1(t_2)}
\end{array}
\]

\[
\text{CF}_1\left( (\lambda r. (t_1 \text{ bop } t_2) \right)
\]

\textbf{Definition 4.4} We introduce a predicate \( CF_2 \) to assert that a \( \lambda \)-expression is in canonical form \( cf_2 \). For a given \( \lambda \)-expression \( t \), \( CF_2(t) \) can be deduced using the following rules:

\[
\begin{array}{c}
\text{atom}(t) \\
\overline{\text{CF}_2\left( t \right)} \\
\text{register}(t) \\
\overline{\text{CF}_2\left( t \right)} \\
\text{CF}_2\left( (\lambda r. t)(e) \right) \land \text{WFX}(e)
\end{array}
\]

\[
\begin{array}{c}
\text{CF}_2\left( t \right) \\
\overline{\text{CF}_2\left( (\lambda r. t)(e) \right)}
\end{array}
\]

The domains \( cf_0, cf_1 \) and \( cf_2 \) can now be formally defined in terms of set comprehensions based on \( CF_0, CF_1 \) and \( CF_2 \) (e.g., \( cf_1 = \{ t \mid CF_1(t) \} \)).

\section{5 Transformation Design}

Our transformational approach to common subexpression elimination has the signature:

\[
\begin{align*}
\text{cf}_0 & \xrightarrow{\text{transform}_1} \text{cf}_1 \\
\text{cf}_1 & \xrightarrow{\text{transform}_2} \text{cf}_2
\end{align*}
\]

This signature states the following:

\[
\forall t_0 : t_0 \in \text{cf}_0 \rightarrow \text{transform}_1(t_0) \in \text{cf}_1 \tag{3}
\]

\[
\forall t_1 : t_1 \in \text{cf}_1 \rightarrow \text{transform}_2(t_1) \in \text{cf}_2 \tag{4}
\]

Our overall objective is to transform an input term \( t_0 \in \text{cf}_0 \) into an output term \( t_2 \) such that:

\[
(t_2 \in \text{cf}_2) \land (t_0 \equiv t_2) \land \text{pseudo\_optimal}(t_2) \land \text{pseudo\_minimal\_ordering}(t_2)
\]

As mentioned earlier, these transformations do not eliminate common subexpressions across statement or block boundaries. Research is ongoing (see Section 10) to apply these methods to ever increasing scopes in order to eventually perform these optimizations program wide.
5.1 $\lambda$-based Manipulation

As mentioned in Section 4, in our transformational framework $\lambda$-expressions are used to model arithmetic expressions. Transformation is predominantly based on the following four well-known lambda equivalences [30]. In the definitions that follow, $bop \in \{+,-,\ast,/\}$ denotes a binary arithmetic operation, $uop \in \{-\}$ denotes a unary arithmetic operation, and $FV(e)$ denotes the set of free-variables in the $\lambda$-expression $e$.

**Equivalence 5.1** Abstraction: $e \equiv (\lambda x.x)e$

**Equivalence 5.2** Scope Capture Left:

$$(\lambda x_1.x_1)((\lambda x_2.x_2)e_1 \ bop \ (\lambda x_3.x_3)e_2) \equiv \lambda x_2.((\lambda x_1.x_1)(x_2 \ bop \ (\lambda x_3.x_3)e_2))e_1$$

*if* $x_2 \notin FV(e_2)$

**Equivalence 5.3** Scope Capture Right:

$$(\lambda x_1.x_1)((\lambda x_2.x_2)e_1 \ bop \ (\lambda x_3.x_3)e_2) \equiv \lambda x_3.((\lambda x_1.x_1)((\lambda x_2.x_2)e_1 \ bop \ x_3))e_2$$

*if* $x_3 \notin FV(e_1)$

**Equivalence 5.4** Scope Capture Unary:

$$(\lambda x_1.x_1)(uop \ (\lambda x_2.x_2)e_2) \equiv \lambda x_2.((\lambda x_1.x_1)(uop \ x_2))e_2$$

Informally $\text{transform}_1$ will rewrite all arithmetic subexpressions in form $cf_0$ as identity $\lambda$-expressions as presented in Section 4. Common subexpressions are eliminated in $\text{transform}_2$ by increasing the scopes of all $\lambda$-expressions in $cf_1$ while at the same time eliminating all equivalent $\lambda$-expressions that are ‘captured’ by the expanding scope. An example of applying $\text{transform}_2$ to the $cf_1$ form of expression $a + a$ is given in Figure 8.

In Section 8, a strategy based on Equivalence 5.1 is used to transform an arithmetic expression $e_0$ into a corresponding $\lambda$-expression $e_1 \in cf_1$. A strategy based on a refinement of the Equivalences 5.2, 5.3, and 5.4 is used to transform a $\lambda$-expression $e_1 \in cf_1$ into an equivalent $\lambda$-expression such that $e_2 \in cf_2 \land \text{pseudo\_optimal}(e_2)$. In addition, the application of the Equivalences 5.2 and 5.3 is controlled in such a manner that $\text{pseudo\_minimal\_ordering}(e_2)$ is achieved.
6 Program Transformation Using HATS

HATS [31] is an IDE that provides a variety of capabilities germane to transformation-based software development. These capabilities include: (1) an engine where transformation can be performed through the execution of programs written in a special purpose language called TL, (2) a parser generator having GLR-like [32] capabilities accepting as input extended BNF grammars together with precedence and associativity rules, (3) an abstract pretty printer, (4) graphical display facilities for viewing the structure of parse trees, (5) text editors, (6) a display showing various metrics associated with TL program execution (e.g., number of rewrites applied, etc.) and (7) some rudimentary tracing capabilities to assist in debugging transformations [33].

7 TL

TL is a language that has been developed exclusively for describing transformation-based computation [9,8]. The principle artifacts manipulated during the execution of a TL program are parse trees, which we will also refer to as terms. TL provides a notation for describing parse tree structures relative to a given (assumed) grammar \( G \). Trees expressed using this notation are referred to as patterns.

For example, suppose we are given a grammar where the derivation \( stmt \Rightarrow id = 5 \) is possible. The pattern \( stmt[ id_1 = 5 ] \) describes a tree corresponding to this derivation. In this context, the subscripted variable \( id_1 \) denotes a typed variable quantified over the syntactic category of all trees having the nonterminal \( id \) as their root node.

A pattern is either a subscripted nonterminal (e.g., \( B_1 \)) or an expression of the
form $B[\alpha']$. An expression $B[\alpha']$ is well-formed if and only if the derivation $B \Rightarrow \alpha$ is possible according to the underlying grammar and $\alpha'$ is obtained from $\alpha$ by subscripting all nonterminals occurring in $\alpha$. Note that $\alpha' = \alpha$ when $\alpha$ consists entirely of terminal symbols.

In TL, transformation is accomplished by the application of rewrite rules to terms (i.e., parse trees). In TL, a rewrite rule has the following syntactic structure:

$$lhs \rightarrow rhs \ [ \text{if condition} \ ]$$

where [ and ] are syntactic meta symbols indicating that the enclosed section (i.e., the conditional portion) of a rule is optional. In order for a rule to be well-formed it is necessary that $lhs$ be a pattern, that $rhs$ be a strategic expression, and that condition be a boolean composition of strategic expressions and match expressions.

A strategic expression is an expression whose evaluation yields a strategy. In order for this definition to be suitable in a first-order setting, we define a term to be a strategy of order 0. In the context of a condition, a strategic expression has an abstract interpretation in which the boolean value $true$ indicates that the evaluation of the strategic expression resulted in the successful application of a strategy to a pattern. The value $false$ indicates unsuccessful application.

A match expression is a first-order match between two patterns. In TL, the binary symbol $\ll$ is used to explicitly denote first-order matching between a non-ground and ground term $\text{[34]}. Let t_1 denote a pattern, possibly non-ground, and let t_2 denote a ground pattern. The expression $t_1 \ll t_2$ denotes a match expression and evaluates to $true$ if and only if a substitution $\sigma$ can be constructed so that $\sigma(t_1) = t_2$. For example, if $a$ and $b$ are legal identifiers in a language, the match expression $\text{expr} J id_1 + id_2 K \ll \text{expr} J a + b K$ would evaluate to true resulting in the binding of $a$ to $id_1$ and $b$ to $id_2$.

In TL, labelled rules can be written using the traditional rewrite notation, in curried functional form, or a mixture of both. The following shows three different syntactic styles defining the label $r$. All definitions shown are semantically equivalent.

$$r : \text{lhs}_2 \rightarrow \text{lhs}_1 \rightarrow \text{rhs}_1$$

$$r \text{lhs}_2 : \text{lhs}_1 \rightarrow \text{rhs}_1$$

$$r \text{lhs}_2 \text{lhs}_1 : \text{rhs}_1$$

Which syntactic form to use is largely a matter of preference.

\text{[34] The choice of this symbol was inspired its use in the $\rho$-calculus [34].}
7.1 Standard Control Mechanisms

It is widely recognized that generalized explicit control is the primary distinction between (pure) rewriting and rewrite-based transformation (e.g., strategic programming). In contrast to a pure rewriting system, TL requires that the application of rewrite rules to terms (i.e., the transformation process) be explicitly controlled. A specification of such control is called a strategy. The presence of strategies classifies TL as a strategic programming language [4]. Other examples of strategic programming systems include Stratego [7] and Strafunski [35].

In TL, a strategy is an expression composed of various elements including rewrite rules, rewrite rule abstractions (i.e., rule labels), combinators, and traversals. The application of a strategy $s$ to a tree $t$ is expressed using the traditional function application syntax: $s(t)$. Within a strategy, control is expressed using the following standard control mechanisms:

(1) The application of rules to a single term is controlled as follows:
   (a) At the rule level control is exercised through first-order matching and optional rule conditions (i.e., conditional rewrite rules).
   (b) At the strategy level control is exercised through combinators. The set of standard binary strategic combinators includes the sequential composition ($<_$) and left-biased choice ($<_+$) combinators. Let $s_1$ and $s_2$ denote two strategies. The composition $s_1 <; s_2$ is a strategy that when applied to a term $t$ will first apply $s_1$ and then apply $s_2$. In contrast, the composition $s_1 <;+ s_2$ is a strategy that when applied to a term $t$ will first apply $s_1$ to $t$ and only apply $s_2$ if the application of $s_1$ to $t$ is unsuccessful.

(2) The application of a strategy to a term structure is controlled by traversals. TL provides a rich framework for defining traversals as well as a library of standard generic first-order traversals. Traversals specify the order in which the sub-terms of a given term are visited. Standard first-order traversal include top-down left-to-right ($TDL$) and bottom-up left-to-right ($BUL$). For example, let $s$ denote a strategy. In TL, the expression $TDL\{s\}$ denotes a strategy that will traverse the term to which it is applied in a top-down left-to-right fashion and apply the strategy $s$ to every (sub)term visited.

   From an operational perspective, traversals in TL are defined using recursive equations that involve strategic combinators and one-layer traversals. Thus, the TL library of traversals is simply a predefined set of such definitions. Users can also extend this library by defining their own custom traversals using these mechanisms.
7.2 Special Control Mechanisms

In addition to the standard strategic mechanisms described, TL also provides several mechanisms whose origins are unique to TL. These mechanisms include a number of unary combinators and constructs that enable generic traversals to be generalized to higher-order strategies. In [36], a rigorous treatment of the complete set of special combinators can be found. In this section, we describe the semantics of the special combinators \textit{transient}, \textit{hide}, and \textit{lift} in a more informal manner.

7.2.1 The \textit{transient} Combinator

An important unary combinator in TL is the \textit{transient} combinator. This combinator restricts a strategy so that it may be applied \textit{at most once}. The “at most once” property is the hallmark of the \textit{transient} combinator. Such a strategic behavior has been recognized as being useful in practice. For example, Stratego also provides a transient-like ability supporting “the application of dynamic rules only once” [37].

Theoretically, a \textit{transient} can be understood as follows. Let $s$ denote a transient-free strategy enclosed in the context \textit{transient}(...$s$...), where the ellipsis denote a transient-free strategic context. Furthermore, suppose that $s$ has just been successfully applied to a term. Under these conditions, the following strategy reduction occurs:

$$\text{transient(...}s\text{...)} \rightarrow \text{SKIP}$$

(5)

where \textit{SKIP} is a strategic constant whose application to a term is never successful. Intuitively, \textit{SKIP} behaves as if it is not present. That is, a strategy that is reduced to \textit{SKIP} is simply removed from the transformational process. Formally, the behavior of \textit{SKIP} is defined as follows:

$$\text{SKIP}(t) \equiv t$$

$$\text{(SKIP }<\!\!\!\rightarrow s)(t) \equiv s(t)$$

(6)

Transients open the door to \textit{self-modifying} strategies. When using a traversal to apply a self-modifying strategy to a term, it becomes possible to apply a different strategy to every term encountered during a traversal. It should be noted that the \textit{transient} combinator becomes particularly useful in a framework (like TL) where strategies can be created dynamically. Especially, when higher-order traversals are used to create strategies [9].
7.2.1.1 Example. Conceptually speaking, transients are useful for establishing correspondences between terms. These correspondences can be 1-1, 1-to-many, many-to-1 or many-to-many. In the degenerate case, a transient can even be used to simulate a counter as is done in the strategy countLambda shown in Figure 12.

A textbook example of a more interesting correspondence is the 1-1 correspondence that exists between the formal parameters of a function and the actual parameters of a call to that function. This has application in a well-known compiler optimization called in-lining in which a function call is replaced by an instance of the body of the function. The instance of the function body is created by substituting all occurrences of formal parameters in the body with the corresponding actual parameters of the call. Within the framework of TL, such a correspondence can be established through the use of transients. Here we present a simplified version of the heart of the correspondence problem. We will revisit this problem in a more realistic setting in a later example involving higher-order transformation.

Figure 9 defines a small grammar in which it is possible to create lists whose elements are either: (1) identifiers – which we will use to represent formal parameters, (2) integers – which we will use to represent actual parameters, and (3) tuples – which we will use to model the correspondence between an identifier and a value.

<table>
<thead>
<tr>
<th>list</th>
<th>::=</th>
<th>{ item_list }</th>
</tr>
</thead>
<tbody>
<tr>
<td>item_list</td>
<td>::=</td>
<td>item , item_list</td>
</tr>
<tr>
<td>item</td>
<td>::=</td>
<td>id</td>
</tr>
<tr>
<td>id</td>
<td>::=</td>
<td>ident</td>
</tr>
<tr>
<td>number</td>
<td>::=</td>
<td>integer</td>
</tr>
</tbody>
</table>

Fig. 9. A small grammar for highlighting in-lining

Here, our goal is to construct a strategy that will combine the list $\text{list}\text{actuals} = \{10, 20, 30\}$ and $\text{list}\text{formals} = \{x, y, z\}$ to produce the list $\{(x, 10), (y, 20), (z, 30)\}$. More specifically, we would like to achieve the following transformation:

\[
\{x, y, z\} \xrightarrow{\text{transform}} \{(x, 10), (y, 20), (z, 30)\}
\]

This transformational idea can be implemented in TL using transients as shown below:
When applied to the term \textit{list formals}, the strategy \texttt{establish\_correspondence} will visit the items of the list in a left-to-right fashion. It is only to these item sub-terms that the strategy \texttt{hardcoded\_zip} can be successfully applied. Since \texttt{hardcoded\_zip} composed from smaller strategies using the \texttt{<} operator, only one sub-strategy will be applied to any given item. Furthermore, after a transient sub-strategy is successfully applied, the sub-strategy is reduced to \texttt{SKIP} and effectively “goes away”. Thus, the first term of the form \textit{item\[id \_1\]} encountered will be transformed to \textit{item\[(id \_1,10)\]}, while the second such term will be transformed to \textit{item\[(id \_1,20)\]}, and so on.

7.2.2 Identity-based versus Failure-based Semantics

In a strategic framework, standard combinators such as left-biased choice (\texttt{<}) exercise control over rewriting based on an abstract interpretation of the application operation. In particular, the application of a strategy to a term is interpreted as being either \textit{successful} or \textit{unsuccessful}. This approach assumes the ability to \textit{observe} the successful/unsuccessful nature of strategy application. A fundamental question concerns itself with how this observation is made. That is, what mechanism is used to perform such observations?

At the present time, there are two primary schools of thought on the semantics of observation. One school of thought has its origins in logic programming and treats the unsuccessful application of a strategy to a term as \textit{failure}. We refer to strategic programming systems that treat strategy application in this fashion as \textit{failure-based systems}. A second school of thought has its origins in rewriting and treats the unsuccessful application of a strategy to a term as an \textit{identity} on terms. We refer to strategic programming systems that treat strategy application in this fashion as \textit{identity-based systems}.

The essential difference between identity/failure-based systems is highlighted by the following example. Let \texttt{s} denote a strategy whose application to the term \texttt{t} is unsuccessful. The evaluation of the expression \texttt{s(t)} for both types of systems is shown below.
Failure-based: \[ s(t) \xrightarrow{\text{transform}} \text{fail} \]

Identity-based: \[ s(t) \xrightarrow{\text{transform}} t \]

The semantics of TL is *identity-based*. In contrast, at the present time, the majority of strategic programming systems have a *failure-based* semantics [7,22,35]. However, the importance of identity-based strategies is beginning to emerge. For example, TOM [38,39] is a failure-based system in which all strategies are seen as an extension of either the Identity strategy or the Fail strategy. In [40], *conditional transformations* are defined within a logic-based framework. In this logic-based framework, transformations can be composed into OR-sequences as well as AND-sequences. The behavior of OR-sequences has similarities with the identity-based semantics of TL.

A detailed discussion of TL’s identity-based semantics can be found in [36]. For the purpose of this discussion, it is sufficient to think of identity-based strategic control as being realized by an observer function \( f_{observe} \), whose internals are unimportant, that implicitly resides within the implementation of TL. Let \( s_1 \) and \( s_2 \) denote two strategies and \( t \) denote a term. Furthermore, assume that the application of \( s_1 \) to \( t \) is successful and yields \( t' \) and that the application of \( s_2 \) to \( t \) is unsuccessful. In TL, this produces the following behavior.

<table>
<thead>
<tr>
<th>Application</th>
<th>Concrete Result</th>
<th>Abstract Observation by ( f_{observe} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1(t) )</td>
<td>( t' )</td>
<td>( \text{true} )</td>
</tr>
<tr>
<td>( s_2(t) )</td>
<td>( t )</td>
<td>( \text{false} )</td>
</tr>
</tbody>
</table>

### 7.2.3 Observing the Application of Traversals

In a strategic programming idiom, a typical first-order strategy is one that traverses a term in a bottom-up left-to-right fashion and applies some strategy \( s \) to every term encountered. In TL, such a strategy would be expressed \( BUL\{s\} \). Due to the heterogeneous nature of term structures, the strategy \( s \) can typically only apply to a few select sub-terms during the course of a traversal. This is the power of the generic traversal.

There is a tension between the generality of generic traversal and the unforgiving nature of failure-based semantics. Specifically, in order for a generic traversal like \( BUL\{s\} \) to succeed in a failure-based system, the strategy \( s \) will need to successfully apply to every sub-term visited during the traversal. In a failure-based system, this tension is typically resolved by conditionally extending \( s \) with the strategic constant \( ID \) (i.e., \( BUL\{s \Leftarrow ID\} \)), where the constant \( ID \) denotes a strategy that can be successfully applied to any term and whose application to a term leaves the term unchanged. Note that the
strategy $s \leftarrow ID$ will always successfully apply to any term.

In TL, there is no tension between generic traversal and its identity-based semantics. In particular, it is unnecessary to extend strategies with $ID$ in order to achieve a reasonable traversal semantic. As a consequence, observing whether the application of $BUL\{s\}$ to a term $t$ succeeds is meaningful. In particular, the application $BUL\{s\}(t)$ succeeds if and only if there exists one or more sub-terms $t_1, \ldots, t_n$ in $t$ to which the application $s(t_i)$ can be observed by $f_{observe}$. It is important to note that in order for the application $BUL\{s\}(t)$ to succeed, $s$ need not apply to every term encountered during the $BUL$ traversal of $t$. It is precisely this behavior that an identity-based semantics permits, and which enables the strategic control offered by combinators to be seamlessly integrated with the mechanism of generic traversal. As a result, choice combinators such as $\leftarrow +$ can be used to compose a traversal-based strategy such as $BUL\{s_1\}$ with an arbitrary strategy $s_2$ in a meaningful fashion (e.g., $BUL\{s_1\} \leftarrow + s_2$). Similarly, strategies (e.g., traversals and such) may form the conditional portion of a rule. In TL, for example, one can write:

$$lhs \rightarrow rhs \text{ if } BUL\{s\}(lhs)$$  

(7)

Such conditions are useful for checking term structures for a variety of syntactic properties (see [36] for examples of this).

7.2.4 The Combinators hide and lift

In this section, we describe two non-standard unary combinators hide and lift. In a standard strategic framework, conditional composition, of the kind realized by $\leftarrow +$, is a primary mechanism for controlling rule and strategy application. The hide combinator is a unary combinator that when applied to a strategy $s$, prevents the function $f_{observe}$ from observing whether the application of $s$ to a given term is successful. The semantics of hide is formally described as follows:

$$\text{hide}(s_1) \leftarrow + s_2 \equiv s_1 <; s_2$$  

(8)

Given the equivalence above, a natural reaction at this point is to question whether hide combinator does anything new. A full discussion of this issue lies beyond the scope of this paper (see instead [36] for a complete discussion). However, we would like to point out three characteristics of TL that are germane to such a discussion.

**First**, in TL, the application of strategies to terms is identity-based [9,8,36].

**Second**, TL generalizes generic traversal to higher-order strategies. It is in this...
setting that the hide (and lift) combinators become particularly useful. The intuition behind this is that higher-order strategies produce strategy sequences (composed via binary combinators) using traversals that visit terms in some particular order. The hide (and lift) combinators enable the specification of strategic control spanning such sequences.

**Third,** TL supports a variety of non-standard combinators such as transient. The interplay between these non-standard combinators and dynamic strategy creation provides a rich environment for expressing control.

An example of a non-trivial strategy using the hide combinator can be found in [31]. In this article, we present an example of its use below and in Section 8.2.

Within the context of a hide, the unary combinator lift exposes the outcome of strategy application to the observer function \( f_{\text{observe}} \). Let \( s_1 \) denote a hide-free strategy (i.e., a strategy containing no occurrences of the hide combinator) whose application to a given term \( t \) is successful. Given these assumptions, the semantics of lift can be formally described as follows with respect to its application to the term \( t \).

\[
(hide(...lift(s_1)....) <\leftrightarrow s_2)(t) \equiv hide(...lift(s_1)....)(t) \tag{9}
\]

where the ellipsis here denote hide-free strategic contexts.

### 7.2.4.1 Example.

Conceptually speaking, the hide combinator is useful primarily for re-factoring a strategy. The purpose of this is to change the control flow that would otherwise exist within the strategy and is typically only of practical value in a setting where strategies are dynamically created.

However, an interesting idiom arises when the hide and lift combinators are used together. Under these conditions, it becomes possible to design strategies that can be used to check some property of a term without actually rewriting the term. Specifically, it is the observed successful/unsuccesful application of the strategy that implies the presence/absence of the given property. In this setting, lift behaves as an exception-like mechanism for communicating to the enclosing context that the given property has been discovered.

A textbook example of a property-checking strategy is one that checks to see if an expression contains any free-variables. The example given here assumes expressions have been defined by a grammar similar to the one shown in Figure 10. We pick up an instance of the problem of checking for free variables at the point where the set of bound variables for a given context has been determined to be \( \{x, y, z\} \). For simplicity, we also assume that the set of bound variables is fixed. In particular, the expressions to which we will apply our strategy may not contain \( \lambda \)-abstractions introducing additional bound variables. Under
these assumptions, the strategy `contains_free` (shown below) can be used to check an expression for the presence of free variables.

\[
\begin{align*}
\text{hardcoded_bound:} & \quad id[x] &\rightarrow id[x] &\leftarrow id[y] &\rightarrow id[y] &\leftarrow id[z] &\rightarrow id[z] \\
\text{free_var:} & \quad id_1 &\rightarrow id_1 \\
\text{contains_free:} & \quad BUL\{\text{hide(hardcoded_bound} &\leftarrow \text{lift(free_var)})\}
\end{align*}
\]

Specifically, if it can be observed that the strategy `contains_free` successfully applies to an expression, then that expression contains one or more free variables. The reason for this is as follows: When applied to an expression, the strategy `contains_free` attempts to apply `hide(hardcoded_bound & lift(free_var))` to every sub-term encountered. In particular, within the context of the `hide` combinator, the sub-strategy `hardcoded_bound` will successfully apply to any bound variable encountered, thereby preventing the application of the strategy `lift(free_var)` with which it is conditionally composed. It is only in the case where a free variable is encountered, that control will pass to the strategy `lift(free_var)`, which is an identity strategy on identifiers. Because of the `lift` combinator, the successful application of `free_var` will be observable in the environment surrounding (i.e., external to) `hide(hardcoded_bound & lift(free_var))`. This idea can be extended to enable the application of the scope expanding $\lambda$ equivalences, like Equivalence 5.2 and Equivalence 5.3 that were mentioned in Section 5, to be controlled.

### 7.2.5 Higher-Order Generic Traversal

In addition to unary combinators, TL also lifts the notion of generic traversal to higher-order strategies. In TL, a higher-order strategy is a strategy that when applied to a term returns a strategy as a result instead of returning a term. An abstract example of a rule having order $k + 1$ is the following:

\[
\text{pattern}_{k+1} \rightarrow \text{pattern}_k \rightarrow \cdots \rightarrow \text{pattern}_0
\]  

(10)

The successful application of this strategy to a term $t$ yields the result:

\[
\text{pattern'}_k \rightarrow \cdots \rightarrow \text{pattern'}_0
\]

(11)

where \(\text{pattern'}_i\) is an instance of \(\text{pattern}_i\) that has been instantiated with the bindings obtained from \(\text{pattern}_{k+1} \ll t\). Thus, higher-order rules are simply rules whose parameters are curried. However, their interplay with generic traversal is particularly interesting.

Let \(s^{n+1}\) denote a strategy of order $n + 1$. If \(s^{n+1}\) is applied to a term using a traversal, the list of strategies \(<s^n_1, \ldots, s^n_m>\) will be produced (assuming that \(s^{n+1}\) applies successfully $m$ times during this traversal). This list of strate-
gies can be turned into a single strategy by composing the $s^n_i$ using a binary combinator such as $\Leftarrow$ or $\Leftarrow\Leftarrow$, e.g., $s^n_1 \Leftarrow s^n_2 \Leftarrow \ldots \Leftarrow s^n_m$. To generate these kinds of strategies, TL provides a library of higher-order traversals that includes lcond_tdl and lseq_tdl. The expression $\text{lcond} \{ s^{n+1} \} (t)$ will traverse the term $t$ in a top-down left-to-right (TDL) fashion and compose the resulting strategies using the combinator $\Leftarrow$. The expression $\text{lseq} \{ s^{n+1} \} (t)$ is similar and composes the results using the combinator $\Leftarrow\Leftarrow$.

Higher-order strategies provide a mechanism in which data (e.g., sub-terms), distributed over a term structure, can be collected. In spirit, higher-order strategies can be viewed as the strategic version of the “container-like” constructs that can be found in more traditional programming languages (e.g., Java’s container classes). For more discussion on this see [9].

7.2.5.1 Example. The previous two examples demonstrated how first-order strategies could be used to achieve particular transformational objectives. In these examples, specific constants were statically hardcoded into strategies (i.e., hardcoding took place at the source code level). In the context of automatic transformation such hardcoding is generally not practical or even feasible. In this example, we show how higher-order strategies can be used to dynamically create instances of first-order strategies. The resulting first-order strategy instances still contain specific hardcoded constants, it is just that higher-order mechanisms are used to dynamically create these instances.

Here we consider a variation of the in-lining example discussed in Section 7.2.1. As done previously, we assume the actual parameters of the given function call are $\{10, 20, 30\}$ and this information is still statically hardcoded. However, this time, we want to use this information to dynamically create a strategy in which all occurrences of the formal parameters in the function will be rewritten to their corresponding values. Thus, the transformational objective has shifted from simply binding formals with actuals to propagating this information into the function body. For example, occurrences of $x$ should be rewritten to 10, and so forth.

The strategy $\text{subst}$ (shown below) achieves the transformation objective stated in the previous paragraph.

\[
\text{zip2: } \quad \text{transient}(\text{item}[id_1] \rightarrow \text{item}[id_1] \rightarrow \text{item}[10]) \\
\Leftarrow \quad \text{transient}(\text{item}[id_1] \rightarrow \text{item}[id_1] \rightarrow \text{item}[20]) \\
\Leftarrow \quad \text{transient}(\text{item}[id_1] \rightarrow \text{item}[id_1] \rightarrow \text{item}[30]) \\
\text{subst: } \quad \text{list}_\text{formals} \rightarrow BUL\{\text{lcond} \{ \text{zip2} \} \{ \text{list}_\text{formals} \}\}
\]

Note that $\text{zip2}$ is a second-order strategy. When applied to a list of formal pa-
rameters \( \text{list}_{\text{formals}} = \{x, y, z\} \) the strategic expression \( \text{lcond}_\text{tl} \{ \text{zip2} \}[\text{list}_{\text{formals}}] \) will dynamically create the following first-order strategy:

\[
\begin{align*}
\text{item}[x] & \rightarrow \text{item}[10] \\
\leftrightarrow \quad \text{item}[y] & \rightarrow \text{item}[20] \\
\leftrightarrow \quad \text{item}[z] & \rightarrow \text{item}[30]
\end{align*}
\]

This dynamically created strategy can now be applied to the function body. In a similar fashion, the statically hardcoded dependency upon the values of the actual parameters can also be factored out of the strategy, yielding the version of \( \text{zip} \) shown below.

\[ \text{zip3:} \quad \text{item}_{\text{actual}} \rightarrow \text{transient}(\text{item}[\text{id}_1] \rightarrow \text{item}[\text{id}_1] \rightarrow \text{item}_{\text{actual}}) \]

Note that in this case, a (second) higher-order traversal of the actual parameters is required.

8 Implementation

This section describes how the transformational design presented in Section 5 can be implemented in TL. The BNF grammar defining the structures that we are interested in transforming (e.g., arithmetic and \( \lambda \) expressions) is shown in Figure 10. There are two important aspects of the grammar shown. First, the syntactic structure of arithmetic expressions is defined using associativity and precedence rules. Second, expression structures can consist of pure arithmetic expressions, pure \( \lambda \)-expressions, as well as a mixture of arithmetic and \( \lambda \)-expressions.

The remainder of the paper will use the following notation conventions. As used in prior sections, canonical forms will be denoted using lower-case italics (e.g. \( \text{cf}_1 \)) while predicates will use upper-case italics (e.g. \( \text{CF}_1 \)). A fixed-width font (e.g. \( \text{cf1} \)) will be used for implementation strategies.

8.1 Phase I: \( \text{cf}_0 \xrightarrow{\text{transform}_{\text{cf1}}} \text{cf}_1 \)

A strategy called \( \text{transformToCF1} \), shown in Figure 11, is responsible for transforming an arithmetic expression \( \text{e}_0 \) into a \( \lambda \)-expression \( \text{e}_1 \in \text{cf}_1 \). In this phase, transformation is based entirely on Equivalences 5.1 and 5.4 mentioned in Section 5.1. In Figure 11, the rules \( \text{model}_{\text{sym}} \), \( \text{model}_{\text{lit}} \), \( \text{model}_{\text{unaryOp}} \), \( \text{model}_{\text{addOp}} \), and \( \text{model}_{\text{multOp}} \) each represent an instance of the abstract
%LEFT_ASSOC  + − L1
%LEFT_ASSOC  * / L2

\[\begin{align*}
\text{expr} & ::= \text{expr addOp expr} & \%\text{PREC L1} \\
\text{expr} & ::= \text{expr multOp expr} & \%\text{PREC L2} \\
\text{expr} & ::= \text{primary} \\
\text{primary} & ::= \text{entity} | \text{unaryOp primary} \\
\text{entity} & ::= \text{literal} | \text{name} | \text{functionCall} | ( \text{expr} ) \\
\text{functionCall} & ::= \text{functionExpr pendingArgs} \\
\text{functionExpr} & ::= \text{name} | \text{lambdaAbstraction} \\
\text{lambdaAbstraction} & ::= \lambda \text{name} . \text{expr} \\
\text{pendingArgs} & ::= ( \text{argumentList} ) \\
\text{argumentList} & ::= \text{expr} | \text{argumentList} , \text{expr} \\
\text{addOp} & ::= + | − \\
\text{multOp} & ::= * | / \\
\text{unaryOp} & ::= − \\
\text{literal} & ::= \text{number} \\
\text{name} & ::= \text{id} \\
\text{id} & ::= \text{identifier}
\end{align*}\]

Fig. 10. A grammar fragment describing arithmetic expressions and \(\lambda\)-expressions rewrite rule \(e \to (\lambda x.x)e\) accounting for a distinct syntactic structure as defined by the grammar in Figure 10. For example, the rule \text{model addOp} corresponds to the structure defined in line 4 of the grammar, the rule \text{model lit} corresponds to line 11, and so on.

Associated with each of these rules is a condition of the form: \(\text{if } id_r \ll \text{new var}(id[ r ])\). The purpose of this condition is to generate a unique register name\(^5\). For example, the expression \text{new var}(id[ r ]) generates an identifier name beginning with the prefix \(r\). In the rule \text{model sym}, identifiers are generated beginning with the prefix \(r\_sym\). In \text{model lit} the identifiers generated begin with the prefix \(r\_lit\), and so on. From a given level of abstraction, this naming convention can be seen as yielding identifier names containing type information. For example, the identifier \(r\_lit\) is a \text{register} that is bound to a \text{literal} value.

\(^5\) The ability to generate unique identifiers is a built-in operation of TL.
transformToCF1: BUL{cf1}
cf1:
  model_sym <+ model_lit <+ remove_paren <+
  model_unaryOp <+ model_addOp <+ model_multOp
model_sym:
  entity[ id_sym ] → entity[ \lambda id_r_sym. id_r_sym (id_sym) ]
  if id_r_sym \ll new_var(id[r_sym])
model_lit:
  entity[ literal ] → entity[ \lambda id_lit. id_lit (literal) ]
  if id_lit \ll new_var(id[r_lit])
remove_paren:
  primary[ (primary_1) ] → primary_1
model_unaryOp:
  primary[ unaryOp_1 \lambda id_1. expr_body (expr_1) ]
  → primary[ \lambda id_2. \lambda id_1. expr_body (unaryOp_1 id_2) (expr_1) ]
  if id_2 \ll new_var(id[r_expr])
model_addOp:
  expr[ expr_1 addOp_1 expr_2 ]
  → expr[ \lambda id_1. id_1 (expr_1 addOp_1 expr_2) ]
  if id_1 \ll new_var(id[r_expr])
model_multOp:
  expr[ expr_1 multOp_1 expr_2 ]
  → expr[ \lambda id_1. id_1 (expr_1 multOp_1 expr_2) ]
  if id_1 \ll new_var(id[r_expr])

Fig. 11. A strategy realizing $c_0 \xrightarrow{\text{transform}} c_1$

In order to simplify implementation, rule model_unaryOp produces a form that
differs syntactically from the example given in Figure 7. However, the forms
are semantically equivalent, i.e., given the initial expression $-a$,

$$(\lambda r_1.r_1)(-(\lambda a_1.a_1)a) \equiv (\lambda r_1.((\lambda a_1.a_1) - r_1))a$$  (12)
Theorem 8.1 Assuming that \( expr \rightarrow t \) implies \( t \in cf_0 \).

\[
\forall t \in cf_0 : CF_1(t') \quad \text{where} \quad t' = \text{transformToCF1}(t) \tag{13}
\]

Proof (by structural induction): Let \( expr_{in} \) denote the subject term to which \( \text{transformToCF1} \) is applied. The strategy \( \text{transformToCF1} \) will traverse \( expr_{in} \) in a bottom-up left-to-right fashion as dictated by the traversal \( BUL \).

To each term that is encountered during this traversal, the strategy \( cf_1 \) will be applied.

A bottom-up traversal assures that the sub-terms of \( expr_{in} \) will be visited according to a well-founded ordering in which terms corresponding to \( \text{atoms} \), as defined in Section 4.1, in \( expr_{in} \) will be encountered before the more complex expressions to which they belong. Terms that are atoms will be rewritten using the rules \( \text{model_sym} \) and \( \text{model_lit} \) producing resulting terms satisfying \( CF_1 \).

The inductive hypothesis implies that by the time \( cf_1 \) is applied to a non-atomic expression structure such as \( expr_1 \ texts { \\ bop } expr_2 \) or \( - \ expr_3 \), the predicates \( CF_1(expr_1) \land CF_1(expr_2) \) and \( CF_1(expr_3) \) respectively hold. Non-atomic term structures of the form \( expr_1 \ bop \ expr_2 \) or \( - \ expr_3 \) will be rewritten either by \( \text{model_addOp} \), \( \text{model_multOp} \), or \( \text{model_unaryOp} \). In each case, the resulting term will satisfy \( CF_1 \).

Within the strategy \( cf_1 \), the rule \( \text{remove_paren} \) is responsible for the removal of parenthesis that are either superfluous in \( expr_{in} \) or that become superfluous as a result of the application of the rules \( \text{model_sym} \), \( \text{model_lit} \), \( \text{model_unaryOp} \), \( \text{model_addOp} \), and \( \text{model_multOp} \). The rule \( \text{remove_paren} \) has the form:

\[
\text{primary}[ ( \text{primary}_1 ) ] \rightarrow \text{primary}_1
\]

The soundness argument for \( \text{remove_paren} \) is based on precedence and associativity laws of arithmetic operators and an understanding that derivations of the form \( \text{primary} \rightarrow \ expr \ \text{addOp} \ expr \) and \( \text{primary} \rightarrow \ expr \ \text{multOp} \ expr \) are not possible in the given grammar.

The completeness argument for \( \text{remove_paren} \) follows from the following observations:

- The generic traversal \( BUL \) will visit every term in \( expr_{in} \).
- The parenthesis introduced by the successful application of the rules in \( cf_1 \) form the arguments to lambda abstractions and can all be derived from the grammar production \( \text{pendingArgs} ::= ( \text{argumentList} ) \). In particular, the application of \( cf_1 \) does not create any new instances of terms whose structure matches \( \text{primary}[ ( expr_1 ) ] \).
- Let \( expr_1 \ll expr[ \alpha' ] \) where \( \alpha' \) consists entirely of terminal symbols (i.e.,
\[ \alpha' = \alpha \). If \( \text{CF}_1(\text{expr}_1) \), then the derivation of \( \text{expr}_1 \) must have the form \( \text{expr} \Rightarrow \text{primary} \Rightarrow \alpha \). Combining this with our inductive assumption implies that if a term can be matched with \( \text{primary}[(\text{expr}_1)] \) then it can also be matched with \( \text{primary}[(\text{primary}_1)] \). Thus, all instances of \( (\text{expr}_1) \) are accounted for. ■

**Theorem 8.2** The results of the first transformation phase are correct.

\[ \forall t \in \text{cf}_0 : t \equiv \text{transformToCF1}(t) \] (14)

**Proof:** The correctness argument for the rule remove\_paren has already been made. All other rules are based on a directed application of the \( \lambda \)-equivalence \( e \equiv (\lambda x.x)e \). ■

### 8.2 Phase II: \( \text{cf}_1 \xrightarrow{\text{transform}_2} \text{cf}_2 \)

A strategy called transformToCF2, shown in Figure 12, is responsible for transforming \( \lambda \)-expressions \( \text{expr}_1 \in \text{cf}_1 \) to corresponding \( \lambda \)-expressions \( \text{expr}_2 \in \text{cf}_2 \). The strategy transformToCF2 will traverse the term it is applied to using the traversal BUL. During this traversal, the strategy \( \text{cf}_2 \) is applied to each term encountered.

Within \( \text{cf}_2 \), \( \lambda \)-reordering is based on the lambda Equivalences 5.2, 5.3, and 5.4 discussed in Section 5.1. These equivalences are directly realized by the four rewrite rules: expandLeftAddOp, expandLeftMultOp, expandRightAddOp, and expandRightMultOp. It is worth mentioning that the \( \lambda \)-expressions being transformed have been created in such a way (in Phase I) that all \( \lambda \)-variables are unique. Thus, all free variable checks associated with the scope capture identities (e.g., \( x_2 \notin \text{FV}(e_2) \)) are vacuously true and may be removed from the implementation.

The rules expandLeftAddOp, expandLeftMultOp, expandRightAddOp, and expandRightMultOp differ from the lambda equivalences on which they are based in one important respect: the rules are conditionally refined. Specifically, the conditions associated with these rules perform an additional transformation that assures that resulting terms are pseudo-optimal as defined in Section 3.

The pseudo-minimal ordering property (also discussed in Section 3) is established by controlling the order in which the rules expandLeftAddOp, expandLeftMultOp, expandRightAddOp, and expandRightMultOp are applied. This control is exercised by the condition

\[ \text{if rightThenLeft}(\text{primary}_1) \] (15)
transformToCF2: BUL\{cf2\}

\( cf2: \) \[ primary_1 \rightarrow TDL\{right<+ left\}(primary_1) \]
\[ \text{if } \text{rightThenLeft}(primary_1) \]
\[ \leftarrow \]
\[ primary_1 \rightarrow TDL\{left<+ right\}(primary_1) \]

left: expandLeftAddOp \leftarrow expandLeftMultOp
	right: expandRightAddOp \leftarrow expandRightMultOp

\( \text{trip}: \) \( \text{lambdaAbstraction}_{any} \rightarrow \text{lambdaAbstraction}_{any} \)

\( \text{countLambda}: \) \( \text{lambdaAbstraction}_1 \rightarrow \text{transient}(\text{trip}) \)

\( \text{measureAdd}: \) (* other measure rules differ only in operator *)
\[ primary\] \( \text{lambdaAbstraction}_{top} \) \( (\text{primary}_{left} \text{addOp}_1 \text{primary}_{right}) \)
\[ \rightarrow \]
\[ primary\] \( \text{lambdaAbstraction}_{top} \) \( (\text{primary}_{left} \text{addOp}_1 \text{primary}_{right}) \)
\[ \text{if } TDL\{\text{hide(lcond_tdl}{\text{countLambda}}[\text{primary}_{left}] \]
\[ \leftarrow \text{lift}(\text{trip}))\}(primary_{right}) \]

\( \text{rightThenLeft}: \) measureAdd \leftarrow measureMult

\( \text{expandLeftAddOp}: \) (* other expand rules differ only in the operator *)
\[ primary\] \( \lambda \text{id}_{outer} . \text{entity}_{outerBody} \)
\[ (\lambda \text{id}_{inner}. \text{entity}_{innerBody} (expr_{innerArg} \text{addOp}_1 \text{primary}_{right})) \]
\[ \rightarrow \]
\[ primary\] \( \lambda \text{id}_{inner} . \)
\[ \lambda \text{id}_{outer} . \text{entity}_{outerBody}(\text{entity}_{innerBody} \text{addOp}_1 \text{primary}_{pseudo\_optimal}) \]
\[ (expr_{innerArg}) \]
\[ \text{if } \text{primary}_{pseudo\_optimal} \leftarrow \]
\[ \text{BUL}\{\text{pseudoOptimize}[id_{inner}][expr_{innerArg}]\}(primary_{right}) \]

\( \text{pseudoOptimize id}_1 \text{expr}_{common}: \)
\[ \text{entity}\] \( \lambda \text{id}_2 . \text{entity}_{body\_to\_rename} (expr_{common}) \]
\[ \rightarrow \]
\[ \text{BUL}\{id_2 \rightarrow id_1\}(\text{entity}_{body\_to\_rename}) \]

Fig. 12. A strategy realizing \( cf_1 \xrightarrow{\text{transform}_2} cf_2 \)

which is associated with the strategy \( cf_2 \). In particular, for each term \( primary_1 \) encountered by \( cf_2 \) during its BUL traversal, the above mentioned
condition determines whether \( primary_1 \) should be transformed using the strategy \( TDL\{\text{right} \leftrightarrow \text{left}\} \) or the strategy \( TDL\{\text{left} \leftrightarrow \text{right}\} \). A more detailed discussion of the strategy \( \text{rightThenLeft} \) follows later in the paper.

Theorem 8.3

\[ \forall t \in cf_1 : \text{CF}_2(t') \quad \text{where} \quad t' \doteq \text{transformToCF}_2(t) \quad (16) \]

Proof (by structural induction): Let \( expr_in \) denote the subject term to which \( \text{transformToCF}_2 \) is applied. The bottom-up traversal used by \( \text{transformToCF}_2 \) assures that the sub-terms of \( expr_in \) will be visited by \( cf_2 \) according to a well-founded ordering in which \( \lambda \)-term structures are transformed in an inside-out fashion. In particular, we know from the assumption \( CF_1(t) \) that the first \( \lambda \)-terms encountered will have one of the following two shapes:

\[ (\lambda r. r)(a) \quad \text{where} \quad \text{atom}(a) \]  
\[ (\lambda r_1. r_1)(-r_2) \quad \text{where} \quad \text{register}(r_2) \]

Given the accompanying “where” assertions, \( CF_2((\lambda r. r)(a)) \) and \( CF_2((\lambda r_1. r_1)(-r_2)) \) is immediate. Next, there are two general cases that must be considered. In case 1, the \( \lambda \)-term \( primary_1 \) encountered has the shape \((\lambda r. t_0)(t_1 \text{ bop } t_2)\) where \( CF_2(t_0) \wedge CF_2(t_1) \wedge CF_2(t_2) \) holds by inductive hypothesis. To the term \( primary_1 \), either the strategy \( TDL\{\text{right} \leftrightarrow \text{left}\} \) or \( TDL\{\text{left} \leftrightarrow \text{right}\} \) will be applied depending upon the outcome of the evaluation of the condition \( if \text{rightThenLeft}(primary_1) \). To whichever strategy is used, a second inductive argument, based on the number of applications of the strategy (e.g., \( \text{right} \leftrightarrow \text{left} \) or \( \text{left} \leftrightarrow \text{right} \)), can be applied to show that the result of a top-down application of this strategy yields a term satisfying \( CF_2 \).

In case 2, the \( \lambda \)-term encountered has the shape \((\lambda r_1. t)(-r_2)\). By inductive hypothesis we know \( CF_2(t) \). By inspection, we know WFX\((-r_2)\). Thus \( CF_2((\lambda r_1. t)(-r_2)) \).

\[ \blacksquare \]

Theorem 8.4 The results of the second transformation phase are pseudo-optimal.

\[ \forall t \in cf_1 : \text{pseudo-optimal}(t') \quad \text{where} \quad t' \doteq \text{transformToCF}_2(t) \quad (19) \]

Proof (by structural induction): This proof essentially has the same structure as the previous one (and so we omit many of the details). The case of interest involves the \( \lambda \)-term \( primary_1 \) having the shape
(λr. entity_body)(primary_left bop primary_right). Here,

\[
\begin{align*}
\text{pseudo_optimal(entity_body)} & \land \\
\text{pseudo_optimal(primary_left)} & \land \text{pseudo_optimal(primary_right)}
\end{align*}
\]

holds by inductive hypothesis. We also know, from the previous theorem, that

\[
\text{CF}_2(\text{primary_left}) \land \text{CF}_2(\text{primary_right})
\]

holds. If \text{primary}_1 is in normal form with respect to the strategy \text{cf}_2, it implies that \text{register(\text{primary}_1)} \land \text{register(\text{primary}_2)} and thus \text{primary}_1 is pseudo-optimal. If \neg(\text{register(\text{primary}_1)} \land \text{register(\text{primary}_2)}), then the next rewrite step will be carried out by one of the rules expandLeftAddOp, expandLeftMultOp, expandRightAddOp, or expandRightMultOp. Without loss of generality, let us assume that the rule expandLeftAddOp is applied. In the conditional portion of this rule, the sub-term \text{primary}_2 is traversed using \text{BUL}. The strategy applied during this traversal is:

\[
\text{pseudoOptimize[id\_inner][expr\_inner\_Arg]}
\]

This strategy will apply to λ-applications, encountered in \text{primary}_right, whose arguments match \text{expr\_inner\_Arg} (i.e., a common subexpression). When a λ-application is matched, a β-like reduction is performed. However, instead of using a substitution like \[id_2 \mapsto expr\_inner\_Arg\], as is traditionally the case in β-reduction, the substitution \[id_2 \mapsto id\_inner\] is used. Note that such a substitution preserves the structural integrity, with respect to \text{CF}_2, of any term to which it is applied. The substitution is performed using a \text{BUL} traversal. The name capture problem is sidestepped because of the invariant that all lambda variables are unique.

At a more abstract/intuitive level, one can understand the behavior of the strategy \text{pseudoOptimize[id\_inner][expr\_inner\_Arg]} as producing a term of the form \[x_2 \mapsto x_1 body_2\] for the following contexts:

\[
(\lambda x_1. body_1)(expr) \\
\text{bop} \\
(\lambda x_2. body_2)(expr) \text{ where } x_1 \not\in body_2
\]

Here the notation \([x_2 \mapsto x_1 body_2]\) denotes the traditional application of the substitution \([x_2 \mapsto x_1]\) to the term \text{body}_2 subject to the constraints set forth by the λ-calculus. Note that both λ-abstractions have the same argument \text{expr}. This enables the λ-abstractions to share their formal parameter – in effect
“commoning” them. In particular, both can share $x_1$, provided $x_1$ does not occur in $body_2$. This provides the basis for the following transformation:

$$
(\lambda x_1.body_1)(expr) \\
bop \\
(\lambda x_2.body_2)(expr) \text{ where } x_1 \not\in body_2 \\
\Rightarrow \\
(\lambda x_1.body_1 \ bop [x_2 \mapsto x_1]body_2)
$$

Claim 1  The results of the second transformation phase have a pseudo-minimal order.

$$
\forall t \in cf_1 : \text{pseudo\_minimal\_ordering}(t') \ \text{ where } t' = \text{transformToCF2}(t) \quad (23)
$$

Discussion: Within transFormToCF2, the strategy rightThenLeft is responsible for establishing the pseudo-minimal ordering property. This property is heuristic-based and has been loosely defined. Thus, the burden of proof is light. Furthermore, it should be noted that the role of the strategy rightThenLeft in transFormToCF2 is one of control only. That is, rightThenLeft does not directly perform any rewrites that find their way into the actual term being transformed. Furthermore, given the symmetric nature of the strategies controlled by rightThenLeft, it is easy to argue that rightThenLeft does not jeopardize overall correctness.

The basic heuristic being employed here is as follows. Let $(primary_{left} \ bop \ primary_{right})$ denote the argument of a $\lambda$-application that we wish to transform. If $primary_{left} > primary_{right}$ then $primary_{left}$ should be processed first followed by $primary_{right}$ (and vice versa). In this context, the ordering relation $>$ is based on the number of $\lambda$-terms within $primary_{left}$ and $primary_{right}$ to which the rules expandLeftAddOp, expandLeftMultOp, expandRightAddOp, and expandRightMultOp can be applied. In the context of a $\lambda$-application of the form $(\lambda r_1.t_0)(primary_{left} \ bop primary_{right})$, the term $primary_{left}$ is processed by the application of the rules expandLeftAddOp and expandLeftMultOp. Similarly, the term $primary_{right}$ is processed by the application of the rules expandRightAddOp and expandRightMultOp.

The intuition behind this greedy heuristic is as follows. Consider the concatenation of the following three register-based computations:

$$
(1) \ \langle r_{i_1} \mapsto e_{i_1}, \ldots, r_{i_n} \mapsto e_{i_n} \rangle
$$
(2) \( \langle r_{2_1} \mapsto e_{2_1}, \ldots, r_{2_m} \mapsto e_{2_m} \rangle \)

(3) \( \langle r \mapsto r_1 \ bop \ r_{2_m} \rangle \)

Let us assume that the register name spaces of \( \langle r_{1_1} \mapsto e_{1_1}, \ldots, r_{1_n} \mapsto e_{1_n} \rangle \) and \( \langle r_{2_1} \mapsto e_{2_1}, \ldots, r_{2_m} \mapsto e_{2_m} \rangle \) are distinct. That is, \( \forall i, j : 1 \leq i \leq n \land 1 \leq j \leq m \Rightarrow r_{1_i} \neq r_{2_j} \). This assumption together with the dependency of \( r \) on \( r_{1_n} \) and \( r_{2_m} \) suggests two concatenation possibilities:

\begin{enumerate}
  \item \( s_{m.n.1} = \langle r_{2_1} \mapsto e_{2_1}, \ldots, r_{2_m} \mapsto e_{2_m}, r \mapsto r_{1_n} \ bop \ r_{2_m} \rangle \)
  \item \( s_{n.m.1} = \langle r_{1_1} \mapsto e_{1_1}, \ldots, r_{1_n} \mapsto e_{1_n}, r \mapsto r_{1_n} \ bop \ r_{2_m} \rangle \)
\end{enumerate}

From these two possibilities, the goal to minimize \( \text{total\_dist} \), as it is defined in Section 3 suggests (from a local perspective) that if \( n < m \), then the concatenation \( s_{m.n.1} \) is preferred; otherwise \( s_{n.m.1} \) is preferred. The reason for this preference is based on selecting the concatenation \( s \) that minimizes the sum:

\[ u_d\dist(r_{1_n}, s) + u_d\dist(r_{2_m}, s) \quad (24) \]

Specifically, we have:

**Case 1:** \( s = s_{m.n.1} \)

\begin{enumerate}
  \item \( u_d\dist(r_{1_n}, s) = 2 \)
  \item \( u_d\dist(r_{2_m}, s) = n + 2 \)
\end{enumerate}

**Case 2:** \( s = s_{n.m.1} \)

\begin{enumerate}
  \item \( u_d\dist(r_{2_m}, s) = 2 \)
  \item \( u_d\dist(r_{1_n}, s) = m + 2 \)
\end{enumerate}

If \( n < m \) then \( 2 + (n + 2) < 2 + (m + 2) \). A similar analysis holds for the case when \( \neg(n < m) \).

The strategy \texttt{rightThenLeft} orders concatenations according to the criteria described above. Operationally, \texttt{rightThenLeft} behaves as follows: The strategy \texttt{countLambda} is a second-order rule that when applied to a \( \lambda \)-abstraction will return the strategy \texttt{transient(trip)}. (Recall that according to the grammar fragment given in Figure 10, all \( \lambda \)-abstractions are derivable from the nonterminal \texttt{lambdaAbstraction}.) The expression \texttt{lcond\_tdl[countLambda][primaryLeft]} represents an application of the higher-order strategy \texttt{lcond\_tdl[countLambda]} to the term \texttt{primaryLeft}. The traversal \texttt{lcond\_tdl} is a top-down left-to-right higher-order traversal that composes its results using the combinator \(<+\). Thus, the result of the application will be a strategy \( s_{1..n} \) of the form:

\[ s_1 <+ \cdots <+ s_n \quad \text{where} \quad s_i = \text{transient(trip)} \quad (25) \]

In the strategy \( s_{1..n} \), \( n \) corresponds to the number of \( \lambda \)-abstractions in \texttt{primaryLeft}. Let us now consider what happens when the strategy
\( s_1..n \leftarrow \text{trip} \) is applied to \( \text{primary}_{right} \). Case 1: if the number of \( \lambda \)-abstractions in \( \text{primary}_{left} \) is greater-than or equal-to the number of \( \lambda \)-abstractions in \( \text{primary}_{right} \), then the strategy \( \text{trip} \) will never be successfully applied (i.e., the flow of control will not reach \( \text{trip} \)). Case 2: if the number of \( \lambda \)-abstractions in \( \text{primary}_{left} \) is less-than the number of \( \lambda \)-abstractions in \( \text{primary}_{right} \), then \( \text{trip} \) will be successfully applied. So, the observation of the successful application of \( \text{trip} \) can be used to order the processing of order of \( \text{primary}_{left} \) and \( \text{primary}_{right} \).

Due to the semantics of the \( \text{hide} \) combinator, the application of the strategy \( \text{hide}(s_1..n \leftarrow \text{trip}) \) cannot be observed (see Section 7.2.4). However, within the scope of a \( \text{hide} \), the \( \text{lift} \) combinator can be used to expose the successful application of the strategy to which it is applied. In particular, the strategy \( \text{hide}(s_1..n \leftarrow \text{lift}(	ext{trip})) \) is only observed as applying successfully when the sub-strategy \( \text{lift}(	ext{trip}) \) is successfully applied. Thus, the following strategic expression can be used as a condition to determine whether \( \text{primary}_{right} \) is larger than \( \text{primary}_{left} \).

\[
\text{TDL}\{\text{hide}(lcond_tdl[\text{countLambda}][\text{primary}_{left}] \leftarrow \text{primary}_{right})\}
\]

(26)

It should be noted that strategies like \text{rightThenLeft} realize, in a strategic idiom, what essentially amounts to a size comparison between terms – an approach that may not be considered ideal. While this might be theoretically interesting it does suggest that extending pure strategic frameworks with additional computational and analysis capabilities, such as the ability to perform basic mathematical operations and type analysis, would be quite useful for implementing more sophisticated strategies.

\[\Box\]

**Theorem 8.5** The results of the second transformation phase are correct.

\[
\forall t \in cf_0 : t \equiv \text{transformToCF2}(\text{transformToCF1}(t))
\]

(27)

**Proof:** There are only two kinds of rewrites taking place. The first kind is based directly on the \( \lambda \)-Equivalences 5.1 – 5.4 discussed in Section 5.1. The second kind of rewrite takes place in the context of common subexpression elimination. The correctness of these types of rewrites is almost immediate from the semantics of the \( \lambda \)-calculus. \[\Box\]

We would like to mention that one of the primary motivations behind our research into \( \lambda \)-based compilation is the simple nature of such correctness proofs. Note that the optimizations performed only effect the ordering of
\(\text{lambda}\)-expression and are very much “aligned” with the overall correctness argument.

**Corollary 8.6**

Let \(t_0 \in cf_0\) and let \(t_2 = \text{transformToCF2}(\text{transformToCF1}(t))\).

\[
t_2 \in cf_2 \land t_0 \equiv t_2 \land \text{pseudo}\_\text{optimal}(t_2) \land \text{pseudo}\_\text{minimal}\_\text{ordering}(t_2)
\] (28)

**9 An Example**

To illustrate our method let us consider eliminating the common subexpressions in the arithmetic expression \((b + (-a/2)) \ast (b + (-a/2))\).

![Abstract expression tree for \((b + (-a/2)) \ast (b + (-a/2))\)](image)

The strategy \(\text{transformToCF1}\) (see Figure 11) will apply strategy \(cf_1\) to the expression tree for our example (shown in Figure 13) using a bottom-up, left-to-right (BUL) traversal. Note that during this traversal each applied rewrite of the form \(e \rightarrow (\lambda x.e)\) in \(cf_1\) generates a new unique \(\lambda\)-variable. In practice this is accomplished by appending sequential unique integers (beginning with 1) to standard identifiers (i.e., 1 is appended to the first generated identifier, 2 to the second, and so on). This information enables the tracing of the application order of the rewrite rules; see Figure 14 for a listing of canonical form \(cf_1\) for our example.

Figure 15 shows the partial parse tree representation for \(cf_1\) (given the grammar presented in Figure 10) after applying \(\text{transformToCF1}\) to our example expression. Note that many interior nodes and \(\text{subtree}_2\) have been omitted due to limited space. However, with the exception of variable names, \(\text{subtree}_2\) (i.e., the \(\lambda r\_expr\_12\) expression) is identical to \(\text{subtree}_1\) (i.e., the \(\lambda r\_expr\_6\) expression).

The strategy \(\text{transFormToCF2}\) (see Figure 12) will perform a BUL traversal of form \(cf_1\) attempting to apply rule \(cf_2\) to each term encountered. The “expand” rules (\(\text{expandLeftAddOp}, \text{expandLeftMultOp}, \text{expandRightAddOp}, \text{and}\))
(λr_expr_13 . r_expr_13)(
    (λr_expr_6 . r_expr_6)
    (((λb_1 . b_1)(b)
      +
      (λr_expr_5 . r_expr_5)
      (((λr_sym_3 . (λa_2 . a_2)(−r_sym_3))(a)
        /
        (λr_lit_4 . r_lit_4)(2)))
    *
    (λr_expr_12 . r_expr_12)
    (((λb_7 . b_7)(b)
      +
      (λr_expr_11 . r_expr_11)
      (((λr_sym_9 . (λa_8 . a_8)(−r_sym_9))(a)
        /
        (λr_lit_10 . r_lit_10)(2))))))

Fig. 14. Form cf_1 for expression \((b + (−a/2)) \ast (b + (−a/2))\)

expandLeftMultOp) perform the scope capture operation discussed in Section 5.1—each λ-expression (call it expr_0) is “pulled out” over its enclosing expression while at the same time eliminating any and all common expressions equivalent to expr_0 that are “passed over”. In our example these transformations are first attempted on subtree_1, the left-most subtree.

No expressions in our example will be eliminated, however, until each of the scopes of the λ-variables representing common subexpressions in subtree_1 are finally expanded over the

\[ \lambda r_{expr\_13} . r_{expr\_13}(expr_1 \ast expr_2) \]  \hspace{1cm} (29)

expression. For example, when the λb_1 expression captures the scope of expr_2 above by expanding it over the \(\lambda r_{expr\_13}\) expression, common subexpression λb_7 will be eliminated and all references to b_7 in expr_2 will be replaced with b_2. Since subtree_1 and subtree_2 are identical except for variable names, all expressions in subtree_2 will eventually be eliminated. This is illustrated in Figure 16, which shows a term \(t \in cf_2\) for our example expression; note that
Fig. 15. Partial parse tree of form \( cf_1 \) for expression \((b + (-a/2)) \times (b + (-a/2))\) only the subtree \(1\) “versions” of commoned subexpressions have survived.

10 Future Work

The overarching, long-term goal of this proof-of-concept research is to develop a compilation-by-transformation trusted compiler—a compiler that can be shown to correctly compile any program it receives. Towards this end our work continues along two parallel paths—strategy implementation and development of formal proofs.

Most common code optimizations (e.g. constant folding, strength reduction,
(λr_sym_3.
  (λa_2.
   (λr_lit_4.
    (λr_expr_5.
     (λb_1.
      (λr_expr_6.
       (λr_expr_13.r_expr_13)(r_expr_6 * r_expr_6)
       )b_1 + r_expr_5)
       )b
       )a_2/r_lit_4
       )2
     )(-r_sym_3)
   )a

Fig. 16. Form cf2 for expression \((b + (-a/2)) * (b + (-a/2))\)

and function inlining)(see [1]) are examples of the before-mentioned (Section 1) distributed data problem, i.e. they require the recognizing of static syntactic patterns and capturing of information at one point in the syntax tree that can then lead to optimization rewrites at different locations in the tree. As an example, performing constant folding on the expression \((\lambda x.e)2\) would involve the rewrite \((\lambda x.e)2 \rightarrow e\) followed by replacing all occurrences of \(x\) in \(e\) with the constant 2. Writing strategies to perform such transformations is straightforward in TL [9](see [8] for an example of function inlining in TL).

Moving towards a prototype trusted compiler that generates “reasonably good” code will also require the extension of our current optimization work to a global scope. Traditionally such optimizations require performing data flow analysis using a control flow graph [1]. Such flow graphs can be either explicitly created by defining canonical forms with embedded flow information, or implicitly by creating virtual control flow graphs [41,42] directly from syntax trees. And, as mentioned previously in Section 3, our canonical forms are essentially variations of SSA form; SSA form has been shown to be a practical basis for global code optimization [28].

Development of rewrite strategies for register allocation and instruction scheduling are also needed; current work in this area shows promise. Models facilitating rewriting of statements and higher level abstractions (subroutines) need
to be developed.

Work has begun on mechanizing the proofs presented in this paper and in our other ongoing, related work. Full assurance in the trustworthiness of a compiler comes only when one is convinced of its correctness by the development of a system of formal proofs.

11 Conclusion

The approach of eliminating common subexpressions using scope restructuring rewrites defined by the $\lambda$-calculus is attractive because its correctness argument is based directly upon well-known $\lambda$ equivalences. The strategic programming language TL provides a framework that allows a fairly direct realization of these rewrites, thus facilitating the correctness argument. This design has the inherent property that the primary effort in the correctness argument centers on completeness rather than soundness.

The control afforded by strategic combinators can be effectively used to ensure termination and to order rule application to achieve goals that are secondary to correctness. In this case, secondary goals are described by the properties pseudo_optimal and pseudo_minimal_ordering.

Generic traversals provide an elegant behavioral abstraction for visiting all sub-terms within a term. Higher-order strategies provide a native mechanism for moving data from one term to another as well as providing the ability to create strategies specifically targeted to a given input. As a result, we believe that the approach presented is a promising beginning or early step towards the eventual development of a compilation-by-transformation trusted compiler.

References


