Program transformations using temporal logic side conditions

SARA KALVALA and RICHARD WARBURTON
University of Warwick
and
DAVID LACEY
XMOS Semiconductor

This paper describes an approach to program optimisation based on transformations, where temporal logic is used to specify side conditions, and strategies are created which expand the repertoire of transformations and provide a suitable level of abstraction. We demonstrate the power of this approach by developing a set of optimisations using our transformation language and showing how the transformations can be converted into a form which makes it easier to apply them, while maintaining trust in the resulting optimising steps. The approach is illustrated through a transformational case study where we apply several optimisations to a small program.

Categories and Subject Descriptors: D.3.1 [Software Engineering]: Programming Languages—General; D.1.4 [Software Engineering]: Programming Techniques—Sequential Programming; D.3.4 [Programming Languages]: Processors—Optimization

General Terms: Design, Languages
Additional Key Words and Phrases: Program Transformation, Optimizing Compilers, Rewriting, Temporal Logic.

1. INTRODUCTION

Significant effort in modern compiler development is spent implementing optimisations, which are often performed after an initial synthesis of low-level code [Aho et al. 2007; Appel 1998; Muchnick 1997]. Optimisations are often classified as either local or global. Local optimisations transform small segments of code, typically by pattern matching and simple rewriting, while global optimisations exploit complex chains of information and subtle patterns of execution. While such global optimisations are often difficult to identify in a program, the payback is sufficient to motivate development of general and robust techniques to incorporate them into working compilers. Furthermore, the complexity of the conditions at which some global optimisations apply implies that significant care has to be taken to ensure that they are not applied in a way that introduces errors in the program. Thus, a precise, formal language for specifying these global optimisations is a very useful addition to the arsenal of compiler designers.
The rewriting paradigm is particularly appropriate for describing optimisations, as rewrites express the transformation of the program and can usually be specified succinctly and intuitively, with side conditions that describe when the rewrite may be applied. In the case of local optimisations, side conditions can be relatively easy to specify as they are purely syntactic. For global optimisations the situation is more complex, particularly for programs written in imperative languages, as the conditions under which it is possible to apply rewrites are harder to describe—they often depend on capturing some or all the paths of execution through the program. Their specification needs to model semantic information, which is not always easy to track, particularly when dealing with the low-level representation of programs to which they are applied. The obvious solution is to employ a rigorous language for expressing optimisations that supports formal reasoning. However, such a language often raises the issue about its actual applicability to realistic tasks: ‘toy’ languages often carry the risk of not supporting the kinds of complex operations that practitioners would actually like to perform with them. An effective tool to aid optimisation needs to blend expressivity and rigour.

In this paper we propose a specification language for transformations capable of capturing many transformations found within optimising compilers for imperative languages. The transformations apply to control flow graphs, which represent the temporal nature of the program. We claim that the language is expressive enough to capture a variety of optimisations, and we support this claim by presenting a catalogue of optimisations expressed in the language described. The transformations described range from simple ones such as dead code elimination (which checks if the result of a computation will ever be used later in the program’s execution, see Section 8.1), and more complex ones such as lazy code motion (which moves a computation to a different point in the program to eliminate unnecessary calculations in loops, see Section 9.4). Other transformations described in our framework include constant propagation, strength reduction, branch elimination, skip elimination, loop fusion, partial redundancy elimination, and lazy strength reduction.

It is not enough to have a library of very well understood and trusted optimisations: practitioners often design complex, new optimisations to improve compilers for particular applications, and methodologies that can support such development are very useful. In Section 5 we show how new transformations can be assembled to implement special optimisations which arise while manipulating a specific piece of code. The formal underpinning of the language aids the verification of the soundness of transformations expressed in it.

Key to our approach is the blending of ideas from rewriting and temporal logic, which can aid reasoning about the soundness of the optimisations. We have based the language of side conditions on CTL, a well-understood and widely used temporal logic that allows side conditions to be checked mechanically against the representation of a program. Furthermore, as the program to be optimised is expressed in a language with a formal semantics, the soundness of the transformations can also be proved rigorously, as has been shown previously [Lacey et al. 2002; 2004]. Thus, the methodology described here supports the development of transformations, reasoning about the correctness of the transformations, and application of these transformations to programs to be optimised.
Fig. 1. Grammar for instructions in the $L_0$ programming language

The structure of the paper is as follows: Section 2 introduces a simple programming language over which the transformations operate and the representation of programs through control flow graphs. Section 3 explains the background to the design of the transformation specification language while Section 4 describes the syntax of the language. A case study showing how transformations can be formulated is discussed in Section 5, while Section 6 describes the semantics of the language in detail. We describe how TRANS can be used to identify loops within programs through an example of dominator analysis in Section 7. Transformation specifications are described in Section 8 and Section 9. We then show how the process of applying transformations can be mechanised in Section 10, and a discussion of the limitations of our approach is presented in Section 11. In Section 12 we survey related work and finally add some concluding remarks in Section 13.

2. THE SOURCE LANGUAGE

2.1 The $L_0$ programming language

The methodology for specifying transformations in this paper applies to a variety of languages; we introduce a simple imperative language $L_0$ to aid presentation. This toy language is standard in its behaviour; a formal semantics for $L_0$ can be found elsewhere [Lacey 2003]. The language is meant to exemplify compiler intermediate representations rather than programming languages, since that is the level at which optimisations are usually applied. Some features of even low-level programming languages, such as function calls, exceptions, input/output statements and pointers, have not been included in $L_0$. Control structures are written with goto statements instead of looping constructs. We assume the instructions are labelled; if a goto or conditional statement refers to an instruction label greater than the length of the program then control jumps to the last instruction.

An $L_0$ program consists of a sequence of instructions where an instruction can take one of five typical forms.

Definition 2.1 An instr is a single command of the form given in Fig. 1.

Definition 2.2 A program of length $n$ ($n \geq 1$) is a list $I_0, \ldots, I_{n-1}$ of instructions, where instruction $I_{n-1}$ is the only instruction of the form $\text{ret}(e)$.

An example program in $L_0$, corresponding to a small Pascal program which clears two lines of an array, is presented in Fig. 2.
4 Sara Kalvala et al

```pascal
var i : integer;
var a : array 10 of array 3 of integer;
begin
  i := 0;
  while i < 10 do
    a[i][1] := 0;
    a[i][2] := 0;
    i := i + 1
  end;
end.
```

Fig. 2. An example program in Pascal and its translation into $L_0$

2.2 The control flow graph

The transformation language introduced in this paper relies on the representation of programs as control flow graphs (CFGs), where each node corresponds to an individual instruction in the program, except for designated Entry and Exit nodes. The edges between nodes represent possible steps in the program between the instructions. In order to facilitate transformations, labels are added to edges of CFGs: edges that result from the condition in a conditional statement being true are labelled branch, and all other edges are labelled seq to signify default sequential execution.

Formally, a CFG is defined as a triple consisting of a set of nodes, an edge relation and a labelling function which labels each node with an instruction.

**Definition 2.3** A CFG for a program $I_0, \ldots, I_{n-1}$ is the tuple $\langle \text{Nodes}, \text{Edges} \subseteq \text{Nodes} \times \text{Nodes} \times \{\text{seq, branch}\}, I : \text{Nodes} \rightarrow \text{Instr}\rangle$ where:

- **Nodes** = \{Entry, $N_0, \ldots, N_{n-1}$, Exit\}
- **Edges** = \{(Entry, $N_0$, seq), ($N_{n-1}$, Exit, seq)$\} \cup$
  \{(\text{if } N_i = N_{i+1}, \text{seq}) \mid I(N_i) \neq \text{goto } \_\_ 0 \leq i < n - 1\} \cup
  \{(\text{if } N_i = N_j, \text{branch}) \mid 0 \leq i < n, 0 \leq j < n, I(N_i) = \text{if } e \text{ goto } j\} \cup
  \{(\text{if } N_i = N_{n-1}, \text{branch}) \mid 0 \leq i < n, j < 0 \lor j \geq n, I(N_i) = \text{if } e \text{ goto } j\} \cup
  \{(\text{goto } N_i, \text{seq}) \mid 0 \leq i < n, j < 0 \lor j \geq n, I(N_i) = \text{goto } \_\_ \} \cup
  \{(\text{goto } N_i, \text{seq}) \mid 0 \leq i < n, 0 \leq j < n, I(N_i) = \text{goto } N_j\}
- **$I(N)$** = \begin{cases} I_i & \text{if } N = N_i, \\ \text{skip} & \text{otherwise} \end{cases}

Fig. 3(b) shows an example CFG. All CFGs that are referred to within this paper have the additional property of recoverability. A CFG is said to be recoverable if:

- only one node is associated with a ret instruction and this is the only node whose successor is the Exit node;
- the Entry node has no predecessors;
- the Exit node has no successors;

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
0: i := 0
1: j := 0
2: if i = 10 goto 6
3: j := i+j
4: i := i+1
5: goto 2
6: ret(j)

Fig. 3. A program in $L_0$ and its control flow graph

—any node associated with a conditional has exactly two successors, one edge of type $seq$ and one of type $branch$;
—any node not associated with a conditional has exactly one successor connected by an edge of type $seq$ (apart from the $Exit$ node).

Recoverability is a useful property since the transformation methodology described here performs transformations on the CFG rather than the program itself, and one would like to recover a program in $L_0$ from the generated CFG. Recoverability simplifies the process of converting a transformed CFG back into an $L_0$ program. In our work recoverability is preserved by ensuring that a transformation is not applied if any of the above properties are violated by the resulting CFG.

2.3 Paths through a control flow graph

The transformations introduced in Section 4 use side conditions which describe properties about complete paths through a program. Complete paths from a specified point $n$ are sequences of connected nodes through a CFG from point $n$ to the $Exit$ point. Accordingly, complete paths can be extracted from a CFG using the following definition:

**Definition 2.4** For a CFG $G$, the set of complete paths of a system from a state $n_0$ is denoted $CPaths(n_0,G)$ and consists of all finite sequences $n_0 n_1 \ldots n_k$ such that $n_i \rightarrow n_{i+1}$, for all $n_i$ with $i < k$ and such that there does not exist a $n_k+1$ such that $n_k \rightarrow n_{k+1}$. The notation $x \rightarrow y$ is used when $\exists (n, m, \ldots) \in Edges | n = x, m = y$.

2.4 Limitations of $L_0$

We have restricted ourselves to a simple toy language because it allows a variety of low-level programs to be written and still has a simple enough semantics which allows proofs of soundness of the transformations to be performed (see [Lacey 2003]).
More language features could have been added, but this would only make the proofs of soundness more tedious. Others are more complex, such as pointers and exceptions. In Section 11 we present some ideas on how these issues could be resolved when porting our methodology to real programming languages.

3. DESIGNING A SPECIFICATION LANGUAGE FOR TRANSFORMATIONS

3.1 General principles

In our quest to develop a language to specify optimisations which could both be proved to be sound as well as be amenable to mechanical application on programs, we were driven by several requirements:

1. it must be expressive enough to specify a large range of common compiler optimisations;
2. it must be simple enough to facilitate formal analysis, and in particular proofs of soundness of the transformations;
3. it must allow transformation specifications to be automatically applied to programs without requiring any low level programming.

The approach we follow to satisfy these requirements is based on rewriting, temporal logic, and model checking. Transformations are expressed in a language that allows their rigorous proof of soundness, supports automatic checking of the applicability of an optimisation to a program, and automates generation of optimised programs.

3.2 Conditional rewriting

A rewrite typically consists of a left-hand side pattern that represents several abstract syntax trees that match the pattern, and may contain several variables which are given values by a particular match. These are then used to convert the right hand side pattern into a syntax tree. The rewrite is applied by replacing the tree that matches the left hand side with the tree created by the right hand side.

Rewriting is a paradigm that is naturally suitable for formal analysis. Compiler optimisations frequently find common patterns within programs, based on some static analysis or abstract interpretation and replace specific sequences of program instructions by other sequences. However, it is rare that part of a program should be rewritten solely based on a syntactic pattern; some more information may be needed. Often, quite complicated program analysis is required to check whether a particular transformation can apply. A rewrite rule can be extended by a condition that can be seen as a constraint on the possible matches that are allowed on the left hand side of the rewrite. The conditional rewriting paradigm is consequently appropriate for program transformation. In this case the condition of a conditional rewrite should specify what program analysis is needed to determine whether a transformation applies.

Conditional rewrites can be seen as having a left hand side pattern and a condition where both provide pre-conditions to performing a transformation and can therefore be merged. One can generalise this to a bipartite notion of transformation: an action that specifies how to transform a program in terms of some free variables and a condition that shows what must hold of these variables for the transformation
to apply. The condition specifies the program analysis, and binds the result of this analysis to variables. The rewrite rule then uses this binding to instantiate its replacement pattern. A useful transformation toolkit of the kind described here can be found in the Genesis system [Whitfield 1991] where optimising transformations are specified in a specialised language called GOSpeL. In this language one specifies patterns of code that can be optimised and the program analysis required to check that the transformation is correct.

3.3 Temporal logic

We make use of temporal logic as a specification language for the side conditions under which a transformation may apply. Temporal logics describe properties of a system relative to a point in time (or in a particular state of the computation). In our case, the points are nodes (or program points) on the CFGs. These are abstractions of the real state of the program to be run. So a logical judgement is of the form: $\phi @ n$ which states that the formula $\phi$ is satisfied at node $n$ of the CFG.

We base the language for expressing conditions on CTL (Computational Tree Logic, [Clarke et al. 1996]), which can express many optimisations while still being efficient to model check. LTL is also efficiently checkable, however, specifications of some optimisations, such as branch elimination and several forms of partial redundancy elimination (shown in Section 9), are frequently easier to express with branching quantification which LTL doesn’t allow.

We made some modifications to CTL in order to make it easier to express properties of programs, such as the inclusion of past temporal operators ($\neg E$ and $\neg A$). This allows one to intuitively specify classical compiler analyses – backward analyses use forward temporal operators, and forward analyses use backward temporal operators. The extensions for previous state operators are common and in fact [Steffen 1991] includes these extensions for the modal mu calculus. We also extend the next state operators ($EX$ and $AX$) so that one can specify what kind of edge they operate over. For example, the operators $EX_{seq}$ and $AX_{branch}$ stand for “there exists a next state via a seq edge” and “for all next states reached via a branch edge” respectively.

3.4 Model checking

Steffen has previously recognised [Steffen 1991; 1993a] that traditional dataflow analysis and model checking may be used to perform comparable types of analysis. We explore this analogy by using model checking as the technique for examining applicability of transformations.

In our framework, a transformation consists of two parts: a list of actions that change the program by adding or altering points in the CFG and a side condition which specifies when the actions can be applied, written in a language based on temporal logic. A transformation will be performed in the following way:

(1) a model checker is used to find a valuation $\sigma$ that causes the formula $\phi$ to be true of the CFG;
(2) this valuation is used to perform the changes to the graph specified in the list of actions.

There are four types of action: the replace action which replaces a node with
another sequence of nodes, the \textit{remove\_edge} and \textit{add\_edge} actions which add and remove edges and the \textit{split\_edge} action which inserts a node between two other nodes joined by an edge.

The side condition language is an extension of CTL where formulae are built up from basic predicates that describe properties of nodes. Two types of basic predicates are used to obtain information about a node in a CFG: \textit{node}(x) @ n holds for a valuation \( \sigma \) if \( \sigma(x) = \sigma(n) \), and \textit{stmt}(s) @ n holds whenever pattern s matches the statement at node n. Not every element of a predicate needs to be temporal in nature, some are global predicates that do not refer to any node in particular. For example, the formula \( \phi \) @ n \( \land \) \textit{conlit}(c) states that \( \phi \) holds at n (a local property) and c is a constant literal (a global property).

4. THE TRANSFORMATION LANGUAGE: TRANS

We have designed a language for expressing specifications, which we call TRANS, that captures the features described in the previous section. The syntax of TRANS is shown in Figure 4. The language is simple, yet can express many standard compiler optimisations, particularly with the introduction of \textit{strategies}, which combine transformations to create more complex transformations.

As an example of its use, the constant copy propagation transformation is written in TRANS as:

\[
\text{replace } n \text{ with } x := e[c] \\
\text{if} \\
\text{stmt}(x := e[v]) \land \neg \text{def}(v) \lor \text{stmt}(v := c) @ n \land \text{conlit}(c)
\]

In this transformation (described in more detail in Section 8.2), if the content at node n matches \textit{stmt}(x := e[v]), and the only definition of \( v \) that reaches \( n \) is \( v := c \), then \( v \) can be replaced by \( c \) in the expression \( e \) (where \( e \) can be a structured expression). This specification uses \textit{metavariables}, namely \( n, x, e, c \) and \( v \), which stand for elements of the program. Within the TRANS language \( e[c] \) denotes an expression containing \( c \) as an operand.

Rewriting is an important tool in this framework. The traditional way to write conditional rewrites (where a node can be replaced by a sequence of nodes) is as follows:

\[
\text{literal}:\text{pattern} \implies \text{pattern}_1;\text{pattern}_2;\ldots;\text{pattern}_n
\]

This notation is supported in our framework as syntactic sugar and mapped into a \textit{replace} action. For example, the conditional rewrite:

\[
\begin{align*}
n : p & \implies q_1; q_2; \ldots; q_m, \\
& A_1, A_2, \ldots, A_k \\
& \text{if} \quad \phi
\end{align*}
\]

where \( p \) is the pattern to be matched to the statement at \( n \), is an alternate syntax.
literal ::= metavar | num | exit | start | seq | branch

expr-pattern ::= literal
| expr-pattern op expr-pattern
| expr-pattern[expr-pattern]

pattern ::= if (expr-pattern)
goto expr-pattern
literal := expr-pattern
skip
| ret(expr-pattern)

node-condition ::= node-condition ∨ node-condition
| node-condition ∧ node-condition
| ¬ node-condition
| ∃ metavar . node-condition
| [EX | AX | ĒX | ĀX][literal](node-condition)
| [E | A | Ē | Ā](node-condition U node-condition)
| node(literal)
| stmt(pattern)

side-condition ::= side-condition ∨ side-condition
| side-condition ∧ side-condition
| ¬ side-condition
| ∃ metavar . side-condition
| node-condition @ literal
| pred (literal1,..,literaln)

action ::= replace literal with pattern1;pattern2;...;patternn
| remove_edge (literal,literal,literal)
| add_edge (literal,literal,edge-type)
| split_edge (literal,literal,pattern)

transform ::= action1,...,actionn if side-condition
| MATCH side-condition IN transform
| APPLY_ALL transform
| transform □ transform
| transform THEN transform

Fig. 4. The grammar of TRANS

for the action:

\[ \text{replace } n \text{ with } q_1; q_2; \ldots; q_m, \]
\[ A_1, A_2, \ldots, A_k \]
if
\[ \text{stmt}(p) \ @ n \land \phi \]

4.1 Macros

A macro definition provides a way to name commonly used formulae. Macros are of the following general form

\[ \text{let } p(\overline{x}) \triangleq \phi \]

The expression \( p(\overline{x}) \) in a formula represents the syntactic substitution \( \phi[\overline{x}/\overline{x}] \), where each variable in \( \overline{x} \) is replaced by the corresponding variable in \( \overline{x} \). This allows
one to specify formulae that will be used in several different transformations. Free
variables are used in some macros when all the uses of the variables have specific
denotations, such as loop head or loop tail.

As examples, two macros which match nodes that are connected and strongly
connected, respectively, to \( m \) can be written as: e.g.

\[
\text{let connected to } (m) \triangleq E(True \ U \ node(m))
\]

\[
\text{let strongly connected to } (m) \triangleq E(True \ U \ node(m)) \land \overline{E(True \ U \ node(m))}
\]

Macros support the definition of temporal operators \textit{eventually} (\( F \)) and \textit{forever} (\( G \)) in terms of \textit{until} operators in the standard way:

\[
\text{let } EF(\phi) \triangleq E(true \ U \ \phi)
\]

\[
\text{let } AF(\phi) \triangleq A(true \ U \ \phi)
\]

\[
\text{let } EG(\phi) \triangleq \neg AF(\neg \phi)
\]

\[
\text{let } AG(\phi) \triangleq \neg EF(\neg \phi)
\]

Before presenting the semantics of this language and further constructs which aid
specification of transformations, it may be useful to examine a simple exercise in
optimisation, which relies on some well-known optimisations as well as some which
are hand-crafted to suit the purpose in hand. This will give a flavour of the system
and justify the design of the language.

5. AN INTERACTIVE OPTIMISATION EXERCISE

We present a worked example where \textsc{TRANS} has been used to interactively optimise
a program and illustrate how transformations can be written in \textsc{TRANS} that capture
the steps one may wish to apply to optimise a program. The starting point is the
code from Fig. 2, which has been generated from Pascal source code. We add the
syntax \( M[x] \) to \( L_0 \) to represent array/memory access.

Some standard, local transformations, such as constant folding and copy propa-
gation, are first applied to the code, as well as dead code elimination. The resulting
starting point for further global optimisations is shown in Fig. 5(a).

The computation spread over lines 3 and 4 of Fig. 5(a) can be simplified, as the
end result is to assign the value of \( \text{temp2} + 2 \) to \( \text{temp4} \). This form of algebraic
simplification can be generalised to look for an addition between a variable and a
constant where the reaching definition of that variable is also an addition between
a variable and a constant. Standard libraries of transformations may not capture
this special case, but it can be easily written in \textsc{TRANS}, as shown in Fig. 6. The
assignment at node \( n \) is rewritten if the right-hand expression is a sum of a variable
and a constant and the only definition of that variable is a sum between a constant
and a variable. Applying this transformation repeatedly simplifies lines 3 and 8
and allows more dead code elimination, resulting in the code shown in Fig. 5(b).

There are now several transformations that can be applied. In Section 4 we
present several applicable standard transformations such as loop strengthening,
lazy code motion, lazy strength reduction and common subexpression elimination
defined in \textsc{TRANS}. Here, strength reduction replaces the calculations of \( i \times 3 \) in
lines 2 and 5 with pre-calculated values (stored in a new variable \( h \)). Further
applications of copy propagation, constant folding and dead code elimination results
in the code of Fig. 5(c).

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
Fig. 5. Steps in optimising the example program

\begin{align*}
0: & \quad i := 0 \quad 0: \quad i := 0 \quad 0: \quad i := 0 \\
1: & \quad \text{if } (i < 10) \text{ goto } 12 \quad 1: \quad \text{if } (i < 10) \text{ goto } 10 \quad 1: \quad h := 0 \\
2: & \quad \text{temp2} := i * 3 \quad 2: \quad \text{temp2} := i * 3 \quad 2: \quad \text{if } (i < 10) \text{ goto } 10 \\
3: & \quad \text{temp3} := 1 + \text{temp2} \quad 3: \quad \text{temp4} := \text{temp2} + 2 \quad 3: \quad \text{temp4} := h + 2 \\
4: & \quad \text{temp4} := \text{temp3} + 1 \quad 4: \quad M[\text{temp4}] := 0 \quad 4: \quad M[\text{temp4}] := 0 \\
5: & \quad M[\text{temp4}] := 0 \quad 5: \quad \text{temp6} := i * 3 \quad 5: \quad \text{temp8} := h + 3 \\
6: & \quad \text{temp6} := i * 3 \quad 6: \quad \text{temp8} := \text{temp6} + 3 \quad 6: \quad M[\text{temp8}] := 0 \\
7: & \quad \text{temp7} := 1 + \text{temp6} \quad 7: \quad \text{M[\text{temp8}] := 0 } \quad 7: \quad i := i + 1 \\
8: & \quad \text{temp8} := \text{temp7} + 2 \quad 8: \quad i := i + 1 \quad 8: \quad h := h + 3 \\
9: & \quad M[\text{temp8}] := 0 \quad 9: \quad \text{goto 1} \quad 9: \quad \text{goto 1} \\
10: & \quad i := i + 1 \\
11: & \quad \text{goto 1} \\
12: & \quad \text{ret 0}
\end{align*}

(a) \hspace{2cm} (b) \hspace{2cm} (c)

\begin{align*}
0: & \quad h := 0 \quad 0: \quad h := 2 \quad 0: \quad h := 2 \\
1: & \quad \text{if } (h < 30) \text{ goto } 8 \quad 1: \quad \text{if } (h < 32) \text{ goto } 7 \quad 1: \quad \text{if } (h < 32) \text{ goto } 7 \\
2: & \quad \text{temp4} := h + 2 \quad 2: \quad M[h] := 0 \quad 2: \quad M[h] := 0 \\
3: & \quad M[\text{temp4}] := 0 \quad 3: \quad \text{temp8} := h + 1 \quad 3: \quad h := h + 1 \\
4: & \quad \text{temp8} := h + 3 \quad 4: \quad M[\text{temp8}] := 0 \quad 4: \quad M[h] := 0 \\
5: & \quad M[\text{temp8}] := 0 \quad 5: \quad h := h + 3 \quad 5: \quad h := h + 2 \\
6: & \quad h := h + 3 \quad 6: \quad \text{goto 1} \quad 6: \quad \text{goto 1} \\
7: & \quad \text{goto 1} \quad 7: \quad \text{ret 0} \quad 7: \quad \text{ret 0} \\
8: & \quad \text{ret 0}
\end{align*}

(d) \hspace{2cm} (e) \hspace{2cm} (f)

Fig. 6. A variant case of constant propagation

\begin{align*}
n : a := i + c & \Longrightarrow a := j + k \\
\text{if } & \quad \overline{A(\neg \text{def}(i) \wedge \text{def}(j) \wedge \text{node}(n) \wedge \text{stmt}(i := j + d))} \otimes n \wedge \text{conlit}(c) \wedge \text{conlit}(d) \wedge \text{varlit}(j) \wedge \text{varlit}(i) \wedge k \text{ is } c + d
\end{align*}

Fig. 7. Transformation to apply strength reduction

At this point, the variable $i$ is only used to indicate when to exit the loop. However, this information is also contained in $h$, so the variable $i$ could be eliminated. The key to the transformation is identifying two induction variables whose initialisation step have a fixed ratio $r$. The transformation shown in Fig. 7 changes the exit condition of the loop accordingly. Note that this specification uses the macros defining loops and induction variables described in Section 7. This transformation, along with more dead code elimination, results in the code shown in Fig. 5(d).
let simple_use(v) ≜ \exists c. \exists x. (stmt(x := v + c) \land conflit(c))

MATCH
loop(p, h, b, t) \land induction_var(v, c, u) \land
\neg (\neg def(v) U node(m) \land stmt(v := f)) \& h \&
\neg (\neg use(v) \lor simple_use(v) U out_loop) \& h \&
\neg (\neg use(v) U def(v) \lor exit) \& t
IN
replace n with v := i + k if true
THEN
APPLY_ALL
m : (x := v + t_1) \implies x := v + t_2
if
\quad\quad t_2 = t_1 - k \& \neg node(u) \& m

Fig. 8. A transformation to alter an induction variable by a constant

m : (x := v + c) \implies v := v + c,
\quad\quad n : (y := M[x]) \implies y := M[v]; v := v - c
if
\neg (\neg def(x) U node(m)) \& m \&
\neg (\neg use(v) U node(n)) \& m \&
\neg (\neg use(x) \lor node(n) U exit \lor def(z)) \& m \&
\neg (\neg use(v) U \exists k. stmt(v := v + k) \land conflit(k)) \& n

Fig. 9. A transformation to replace an unnecessary induction variable

One may observe that all the uses of h here are of a simple form i.e. in additions to constants. Changing h so it always has some constant offset could eliminate one of these additions. As the desired transformation is quite complex, we introduce strategies, described formally in Section 6.6, which ease the development of complex transformations. The transformation that implements a general case of replacing one constant by another in these situations is shown in Fig. 8. Applying this transformation (and more dead code elimination) results in the code of Fig. 5(e).

A final optimisation would be to eliminate the temp8 variable and use h instead. There are many ways to generalise this step. One simple way is to split it into two transformations. The first replaces the use of temp8 with h and alters h directly beforehand and afterwards. This first step can be specified in TRANS as shown in Fig. 9. Applying this optimisation followed by algebraic simplification and dead code elimination produces the optimised program shown in Fig. 5(f).

We have demonstrated how TRANS can be used to systematically improve a program, with a series of transformations which could have been conceived by a compiler developer to implement particular, subtle optimisations. The fact that these are written in a formal language which supports proof of soundness, however, differentiates this practice from directly implementing ad-hoc optimisations.

We now explain the semantics of TRANS and show the specification of a wide range of transformations, including ones developed for loop optimisations.
6. SEMANTICS

In this section we describe the semantics of TRANS. First the semantics of the side conditions will be described, then the semantics of the actions and finally the semantics of a complete transformation.

6.1 Semantic objects and valuations

Transformations are based on the interpretation of free variables which refer to objects in the program. There are several different sets of objects relating to a program.

Definition 6.1 The semantics TRANS refers to the objects which are manipulated in programs in $L_0$, the main ones being:

- $Instr$ The set of possible instructions
- $Expr$ The set of possible expressions
- $Var$ The set of program variables
- $Num$ The set of numbers used in the program
- $Op$ The set of operators on elements of $Expr$
- $SynFunc$ The set of syntactic functions

Note that $Var \subseteq Expr$ and $Num \subseteq Expr$. We use the symbol $\oplus$ to denote members of $Op$. Members of $SynFunc$ are functions of type $Expr \rightarrow Expr$ which denote simple syntactic substitution, for example $\lambda x.x + y$ where $y$ and $+$ are elements of $L_0$. A restriction that the bound variable only occurs once in the body of the function is made, to ensure each function picks out only one part of a syntax tree.

Definition 6.2 The set $\mathcal{O}$ of objects of the program is defined as

$$\mathcal{O} = \text{Nodes}(\mathcal{G}) \cup \text{Edges}(\mathcal{G}) \cup Instr \cup Expr \cup Op \cup SynFunc.$$ 

Transformations are defined through the use of free variables. We use $MetaVar$ as the type of metavariables, and we use $a, b, \ldots$ as metavariables. The type $MetaVar$ can bind to values within the set $\mathcal{O}$, defined in 6.2. The type $MetaVar$ is specifically used in the translation of side conditions into Binary Decision Diagrams described in Section 10.

The semantics of TRANS is given in terms of a valuation function for objects of $L_0$.

Definition 6.3 A valuation is a mapping from $MetaVar$ to $\mathcal{O}$.

Let $Valuation$ be the type of all valuations. A valuation can convert a Pattern (i.e. the $expr$-pattern non-terminal of TRANS in Fig. 4) into a semantic object. This is, in effect, “substituting” into a pattern containing free variables.

Definition 6.4 The partial function $\text{subst} : Valuation \times Pattern \rightarrow \mathcal{O}$ is defined by

$$\text{subst}(\sigma,x) = \sigma(x) \quad \text{if } \sigma(x) \in Expr$$
$$\text{subst}(\sigma,x \circ y) = \text{subst}(\sigma,x) \circ \sigma(o) \quad \text{if } \sigma(o) \in Op$$
$$\text{subst}(\sigma,e[d]) = \sigma(e)[\text{subst}(\sigma,d)] \quad \text{if } \sigma(e) \in SynFunc$$
6.2 Side conditions
The basis of the side conditions in TRANS are first order CTL formulae. We define
the type NodeCondition that intuitively corresponds to these connectives, and
formally to the language defined by the node-condition non-terminal of TRANS.
These are connected together using first order logic connectives. The truth of these
formulae depend on a valuation, a node in the CFG and the CFG itself. Accordingly,
we introduce a semantic function \([\cdot]\) which maps a NodeCondition to its meaning:

\[
[\cdot] : \text{NodeCondition} \rightarrow ((\text{Valuation} \times \text{Node} \times \text{FlowGraph}) \rightarrow \text{Bool})
\]

The definition of \([\cdot]\) follows the semantics of CTL. For convenience we define \(\overline{G}\)
which is identical to the graph \(G\) but with direction on every edge inverted.

**Definition 6.5** The semantic function for node conditions is defined by

\[
\begin{align*}
[\phi \lor \psi](\sigma, n, G) &= [\phi](\sigma, n, G) \text{ or } [\psi](\sigma, n, G) \\
[\phi \land \psi](\sigma, n, G) &= [\phi](\sigma, n, G) \text{ and } [\psi](\sigma, n, G) \\
[\neg \phi](\sigma, n, G) &= \text{it is not the case that } [\phi](\sigma, n, G) \\
\exists x.\phi(\sigma, n, G) &= \exists v \in O : [\phi](\sigma[x \mapsto v], n, G) \\
\text{node}(m)(\sigma, n, G) &= \sigma(m) = n \\
\text{stmt}(p)(\sigma, n, G) &= I(n) \bowtie_p p \\
\text{EX}_I(\phi)(\sigma, n, G) &= \exists m, n : (m, m, l) \in \text{Edges}(\overline{G}) : [\phi](\sigma, m, G) \\
\text{AX}_I(\phi)(\sigma, n, G) &= \forall m, n : (m, m, l) \in \text{Edges}(\overline{G}) : [\phi](\sigma, m, G) \\
\text{E(\phi U \psi)}(\sigma, n, G) &= \exists p : p \in CPaths(n, G) : \text{Until}(p, \phi, \psi) \\
\text{A(\phi U \psi)}(\sigma, n, G) &= \forall p : p \in CPaths(n, G) : \text{Until}(p, \phi, \psi) \\
\text{EX}_I(\phi)(\sigma, n, G) &= \exists m, n : (m, m, l) \in \text{Edges}(\overline{G}) : [\phi](\sigma, m, G) \\
\text{AX}_I(\phi)(\sigma, n, G) &= \forall m, n : (m, m, l) \in \text{Edges}(\overline{G}) : [\phi](\sigma, m, G) \\
\text{E(\phi U \psi)}(\sigma, n, G) &= \exists p : p \in CPaths(n, G) : \text{Until}(p, \phi, \psi) \\
\text{A(\phi U \psi)}(\sigma, n, G) &= \forall p : p \in CPaths(n, G) : \text{Until}(p, \phi, \psi)
\end{align*}
\]

The definition of the next operators (EX, AX etc.) includes the optional parameter \(I\),
which indicates whether an edge in the graph is a branch or seq edge. If the
statement holds true regardless of which type of branch is used, the parameter
may be omitted.

We define \text{Until} in the following manner.

**Definition 6.6** Consider a path \(p \in \text{CPaths}(n, G)\), for some \(n\), such that \(p = n_0n_1 \ldots n_k\).
The predicate \(\text{Until}(p, \phi, \psi)\) holds if:

\[
\exists j : 0 \leq j \leq k : [\psi](\sigma, n_j, G) \land \forall 0 \leq i < j : [\phi](\sigma, n_i, G)
\]

In order to capture pattern matching, the specification of the side condition semantics makes use of the relation \(\bowtie_{\sigma}\) which is a subset of \text{Instr} \times \text{Pattern} and defined by:

\[
\begin{align*}
I \bowtie_{\sigma} x := e &= I = \sigma(x) := \text{subst}(\sigma, e) \\
I \bowtie_{\sigma} \text{if } (e) &= \exists n. I = \text{if } \text{subst}(\sigma, e) \text{ goto } n \\
I \bowtie_{\sigma} \text{skip} &= I = \text{skip} \text{ or } \exists n. I = \text{goto } n \\
I \bowtie_{\sigma} \text{ret}(e) &= I = \text{ret}((\text{subst}(\sigma, e))
\end{align*}
\]
The temporal logic formulae can be combined using logical connectives and the \@ operator, creating side conditions of the type \emph{Condition} that do not depend on a particular node in the CFG. We define the semantics of these conditions by overloading the semantic function \([.]\):

\[
[.] : \text{Condition} \rightarrow (\text{Valuation} \times \text{FlowGraph} \rightarrow \text{Bool}) .
\]

The definition of this function is straightforward:

\begin{definition}
The semantic function for side conditions is defined by
\[
\begin{align*}
[\phi \lor \psi](\sigma, G) &= [\phi](\sigma, G) \lor [\psi](\sigma, G) \\
[\phi \land \psi](\sigma, G) &= [\phi](\sigma, G) \land [\psi](\sigma, G) \\
[-\phi](\sigma, G) &= \text{it is not the case that } [\psi](\sigma, G) \\
[\exists x. \phi](\sigma, G) &= \exists \tau : \tau \in O : [\phi](\sigma[x \mapsto \tau], G) \\
[\psi \@ m](\sigma, G) &= [\phi](\sigma, \sigma(m), G) \\
[p(x)](\sigma, G) &= \hat{p}(\sigma(x), \sigma)
\end{align*}
\end{definition}

Here, the logical combination of node conditions is standard. The last clause above refers to global predicates, described next.

### 6.3 Basic Predicates

The \textsc{TRANS} language allows a wide range of \emph{predicates} to be defined, some of which are fundamental predicates, and some of which are simply commonly used macros. We make the distinction between those predicates that are global, and those that are parameterised by node.

#### 6.3.1 Global predicates

Global conditions (such as \textit{conlit}(c), which states that \(c\) is a literal constant) are defined by a family of global predicates (each in a type \textit{GlobalPred}\_\(n\) where \(n\) is the arity of the predicate). The semantics of these expressions varies from predicate to predicate and is stipulated by a family of functions \(\hat{\cdot}\), such that for each arity \(n\) there is a function:

\[
\hat{\cdot} : \text{GlobalPred}_n \rightarrow (O^n \times \text{Valuation} \rightarrow \text{Bool}) .
\]

In the example transformations described in this chapter, the following basic global predicates on \(L_0\) are used:

- \textit{conlit}(x) \quad x \text{ is a constant literal}
- \textit{varlit}(x) \quad x \text{ is a variable literal}
- \textit{freevar}(x, e) \quad x \text{ is a free variable of the expression } e
- \textit{is}(x, e) \quad \text{the expression } e \text{ can be statically determined and evaluates to } x

Usually, we write “\(x \text{ is } e\)” instead of \textit{is}(x, e), in the style of Prolog. The formal definition of the above predicates is as follows

\[
\begin{align*}
\hat{\text{conlit}}(x)\sigma &= \sigma(x) \in \text{Num} \\
\hat{\text{varlit}}(x)\sigma &= \sigma(x) \in \text{Var} \\
\hat{\text{freevar}}(v, e)\sigma &= \sigma(v) \in \text{FV}\left(\text{subst}(\sigma, e)\right) \\
\hat{\text{is}}(x, e)\sigma &= \sigma(x) = \text{evalC}(\sigma, \text{subst}(\sigma, e))
\end{align*}
\]

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
The predicates return false if the function subst or evalC is undefined when called. The definition of the is\((x, e)\) predicate uses the following auxiliary function:

\[
\begin{align*}
\text{evalC}(\sigma, x) &= \sigma(x) & \text{if } \sigma(x) \in \text{Num} \\
\text{evalC}(\sigma, x \oplus y) &= \text{evalC}(\sigma, x) \oplus \text{evalC}(\sigma, y) & \text{if } \sigma(\oplus) \in \text{Op}
\end{align*}
\]

Within this definition \([\sigma]\) is the application of \(\sigma\) to the surrounding arguments. It is also useful to have one global predicate fresh, which succeeds when its argument is bound to the next new variable i.e. a variable that has not been mentioned in the program previously.

### 6.3.2 Local predicates

Local predicates describe conditions at particular nodes during execution. The following local predicates are useful for the transformations presented in this paper:

- \(\text{def}(x) \oplus n\) the statement at node \(n\) assigns to variable \(x\)
- \(\text{use}(x) \oplus n\) the statement at node \(n\) is an assignment, whose expression contains the sub-expression \(x\)
- \(\text{trans}(e) \oplus n\) the statement at node \(n\) does not assign to any of the free variables in the expression \(e\)

These predicates are defined in TRANS as macros.

- \(\text{let def}(x) \triangleq \exists e. \text{stmt}(x := e)\)
- \(\text{let use}(x) \triangleq \exists v, e. \text{stmt}(v := e[x]) \lor \text{stmt}(\text{if } (e[x]) \lor \text{stmt}(\text{ret}(e[x]))\)
- \(\text{let trans}(e) \triangleq \neg \exists v, d. \text{stmt}(v := d) \land \text{freevar}(v, e)\)

### 6.4 Actions

Side conditions stipulate which valuations allow the transformation to apply to a program (namely the valuations \(\sigma\) such that \(\phi(\sigma, G)\) holds). Given such a valuation, an action specifies how to alter the program. An action is defined as a function on flow graphs—a partial function, since it could still fail due to a type mismatch. Accordingly, the semantics of different actions have type:

\([\cdot] : \text{Action} \to (\text{Valuation} \times \text{FlowGraph} \rightarrow \text{FlowGraph})\)

There are four types of actions in TRANS: combinations of these actions allow nodes to be added or deleted at any position in the graph.

The add_{edge} action adds an edge of a particular type between two nodes. If an edge of the correct type already exists between the two nodes then the action has no effect.

**Definition 6.8** The action add_{edge}(n, m, e) is defined as:

\[
[\text{add}_{edge}(n, m, e)](\sigma, (N, E, I)) = (N, E \cup \{(\sigma(n), \sigma(m), \sigma(e))\}, I)
\]

The remove_{edge} action removes an edge between two nodes. If no such edge exists then the action has no effect.

**Definition 6.9** The action remove_{edge}(n, m, e) is defined as:

\[
[\text{remove}_{edge}(n, m, e)](\sigma, (N, E, I)) = (N, E \setminus \{(\sigma(n), \sigma(m), \sigma(e))\}, I)
\]
The *replace* action replaces a node with a sequence of nodes. This is illustrated in Fig. 10(a). The action changes all three components of the graph. It adds the nodes necessary to create the new sequence of instructions, to be connected by a series of *seq* edges. The successor edges of the node being replaced are moved to the end of the block and the labelling function of the graph is altered to map the correct values of the instructions on the nodes in the replacement block.

**Definition 6.10** The action replace \( n \) with \( i_1; \ldots; i_k \) is defined as:

\[
\left[ \text{replace } n \text{ with } i_1; \ldots; i_k \right](\sigma, (N, E, I)) = \\
\langle N \cup \{n_2, \ldots, n_k\}, E', I \triangleright [n_1 \mapsto \text{subst}(\sigma, i_1), \ldots, n_k \mapsto \text{subst}(\sigma, i_k)] \rangle
\]

where \( n_1 = \sigma(n) \) and \( n_2, \ldots, n_k \) are nodes not occurring in the original graph and \( \triangleright \) alters the labelling function \( I \) with new bindings. The edge relation \( E' \) is defined as:

\[
E' = \{ \text{remap\_succ}(n_1, n_k, e) \mid e \in E \} \cup \{(\sigma(n_i), \sigma(n_{i+1}), \text{seq}) \mid 1 < i < k \}
\]

and

\[
\text{remap\_succ}(x, y, e) = \begin{cases} 
(m, s, t) & \text{if } x = n, y = m, e = (n, s, t) \\
(e) & \text{otherwise}
\end{cases}
\]

The *split_edge* action alters a graph by inserting a node between two existing nodes joined by a particular edge. This action is useful because it specifies which particular edge is used as placement for a node, in the case where its successor node has several predecessors. This is illustrated in Fig. 10(b), which uses the same labelling function. In this definition \( n \) and \( m \) both represent nodes in the graph, whilst \( e \) is the edge between them that the transformation splits, and \( i \) is the new instruction to be inserted in the split graph.

**Definition 6.11** The action split_edge\((n, m, e, i)\) is defined as:

\[
\left[ \text{split\_edge}(n, m, e, i) \right](\sigma, (N, E, I)) = \\
\langle N \cup \{q\}, \\
(E \setminus \{(\sigma(n), \sigma(m), \sigma(e))\}) \cup \{(\sigma(n), q, \sigma(e)), (q, \sigma(m), \sigma(e))\}, \\
I \triangleright [q \mapsto \text{subst}(\sigma, i)] \rangle
\]

where \( q \) is a new node not occurring in the original CFG and \( \triangleright \) overwrites a map with a new entry.
It is possible to compose several actions in sequence. For example the action
\[ \text{replace } n \text{ with } x := 4, \]
\[ \text{split } \text{edge}(n, m, e, y := x) \]
will first perform the node replacement and then perform the edge split. It is straightforward to define how these actions are performed in sequence:
\[
\begin{align*}
[A_1, A_2, \ldots, A_k](\sigma, G) &= [A_2, \ldots, A_k](\sigma, [A_1](\sigma, G)) \\
\sigma' &= G
\end{align*}
\]
This completes the definition of the semantics of the action component of transformations. This formalisation, along with the semantics of the side conditions in the previous section, allows one to define the meaning of a complete transformation.

### 6.5 Transformations

This section presents the semantics of a transformation, drawing on the previously defined semantic functions. The overall form of a transformation is:

\[
A_1, A_2, \ldots, A_n \quad \text{if} \quad \phi
\]

where each \( A_i \) is some action and \( \phi \) is the side condition. The transformation attempts to compute a valuation \( \sigma \) such that \( [\phi](\sigma, G) \) is true. It then uses this valuation to perform the actions \( A_1 \) to \( A_n \) in left to right order.

Since a side condition can be true under many different valuations, the meaning of a transformation is given by a set of functions that transform graphs. The meaning of a transformation depends on a partial valuation. A partial valuation is a partial function from metavariables to objects. The semantic function \( \semantics{\cdot} \) for transformations will be of type:

\[
\semantics{\cdot} : \text{Transformation} \rightarrow (\text{PartValuation} \times \text{FlowGraph} \rightarrow \mathbb{P}(\text{FlowGraph} \rightarrow \text{FlowGraph}))
\]

We then define a transformation in \( \text{TRANS} \) to be the set of transformations we get courtesy of valuations that are compatible with the given partial valuation and make the side condition true.

**Definition 6.12** The semantic function on transformations is defined as:

\[
[\ldots, A_k \text{ if } \phi](\tau, G) = \lambda x. [A_1, \ldots, A_k](\sigma, x) \mid [\phi](\sigma, G) \text{ holds and } \sigma \upharpoonright dom(\tau) = \tau
\]

where \( f \upharpoonright D \) denotes the function \( f \) restricted to the domain \( D \).

This definition of the semantics as a set of functions between CFGs makes explicit the non-determinism that stems from the fact that many different substitutions may satisfy the applicability condition of the transformation. This can be seen to correspond to the fact that the transformation may apply to many different parts of the CFG. The intended application for these transformations resolves the non-determinism by choosing one correct valuation, i.e. one local place to transform.
6.6 Strategies

In order to succinctly specify more complex transformations, we introduce strategies, which are operators that act upon transformations. In this section we describe four strategies which will be used to define optimisations in Section 9.

6.6.1 Matching Free Variables. The MATCH ... IN strategy executes a transformation restricted to a valuation that satisfies a particular formula. This strategy is particularly useful in combination with other strategies, as in such cases the desired valuations cannot be simply added to the side condition. For example, the transformation

\[ \text{MATCH stmt}(x := e) \oplus n \text{ IN } T \]

specifies that the transformation \( T \) should be used only when a substitution that makes the formula \( \text{stmt}(x := e) \oplus n \) true is found. Only the variables \( n, x \) and \( e \) are restricted; any other free variables in \( T \) are unaffected. Often \( T \) is a complex transformation made of strategies.

The MATCH ... IN strategy is defined in terms of the semantic function for transformations.

**Definition 6.13** The MATCH ... IN strategy is defined by the following extension to the semantic function on transformations:

\[ \lceil \text{MATCH } \phi \text{ IN } T \rceil(\tau, G) = \{ f | \lceil \phi \rceil(\sigma, G) \text{ holds}, \sigma \upharpoonright \text{dom}(\tau) = \tau, f \in [T](\tau \cup (\sigma \upharpoonright \text{FV}(\phi)), G) \} \]

where \( \text{FV}(\phi) \) is the set of free variables occurring in the formula \( \phi \).

6.6.2 Global Transformation. With Definition 6.12 of the semantics of transformations as a set of functions between CFGs, the natural solution is to choose one particular instance to solve the non-determinism. An alternative which is required for some program transformations is a global transformation that applies in every place, i.e. for every valuation that satisfies the side condition. This is achieved with the APPLY_ALL strategy.

**Definition 6.14** The APPLY_ALL strategy is defined by:

\[ \lceil \text{APPLY_ALL}(T) \rceil(\tau, G) = \{ f_1 \circ f_2 \circ \ldots \circ f_n | f_i \in [T](\tau, G) \setminus \{ f_1 \ldots f_{i-1} \} \} \]

where \( n \) is the (finite) number of elements of \([T](\tau, G)\).

In other words, the APPLY_ALL strategy applies all of the possible transformations in any order.

6.6.3 Nondeterminism. In certain cases it is useful to extend the non-determinism of transformations when combining them. The operator \( \sqcup \) on transformations corresponds to a nondeterministic choice which is made available for when valuations are matched.

**Definition 6.15** The \( \sqcup \) operator on transformations is defined by the following extension to the semantic function on transformations:

\[ [T_1 \sqcup T_2](\tau, G) = [T_1](\tau, G) \cup [T_2](\tau, G) \]

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
6.6.4 Composition. Sequential composition involves performing one transformation directly after another. The composed transformation will only succeed if both component transformations succeed. Composition is done with the $T_1 \text{ THEN } T_2$ strategy.

**Definition 6.16** The THEN strategy is defined by:

$$[T_1 \text{ THEN } T_2](\tau, G) = \{ f \circ g \mid f \in [T_1](\tau, G), g \in [T_2](\tau, G)\}$$

This strategy makes most sense intuitively if both $T_1$ and $T_2$ are deterministic i.e. both $[T_1]$ and $[T_2]$ contain only one element.

Four strategies have now been defined and although they are simple they allow transformations to be combined in a very flexible manner. Section 8 describes simpler transformations, whilst Section 9 presents more sophisticated transformations that include strategies. But before introducing these examples, we examine how loops can be recognised within our methodology.

### 7. IDENTIFYING LOOPS VIA DOMINATORS

Since our system operates over the control flow graph of the program, it does not naturally have information about the existence and positioning of loop structures. In this section, we show how loops can be recovered from unstructured graphs, and we define the macro `loop` which is used extensively in our specification of loop optimisations in Section 9.

The key concept is that of dominance. A node $n$ is said to dominate a node $m$ if every path from the start of the program to node $m$ must pass through node $n$. This can be expressed as a temporal logic formula, by stating that at the start node there does not exist a path that satisfies $\neg \text{node}(n)$ until the path reaches $m$.

$$\text{let } \text{dom}(n,m) \triangleq \neg E(\neg \text{node}(n) \ U \ \text{node}(m)) \ @ \ \text{start}$$

Conversely, we can define post dominance, when every backward path from the exit node to $m$ must pass through $n$.

$$\text{let } \text{pdom}(n,m) \triangleq \neg E(\neg \text{node}(n) \ U \ \text{node}(m)) \ @ \ \text{exit}$$

For the purpose of this paper, the general pattern for a reducible loop is depicted in Fig. 11. It is generally the case that optimising compilers don’t deal with irreducible loops, and we consequently don’t account for this case [Tarjan 1973]. Some loops test their condition after their loop bodys, for example do-while loops within the Java programming language, we don’t account for this case since it can be easily converted into the form we look for by copying the loop body to a sequence of nodes ahead of the loop header.

Loops are characterised by certain key nodes: the pre-header (`phead`), the head, the tail and the break nodes. We specify the order of the key nodes in the loop with dominance relations. The pre-header must dominate the head (`dom(phead, head)` must hold) and the break node must post-dominate the tail (`pdom(break, tail)` must hold). In addition, the pre-header node must be the immediate predecessor of the head node:

$$\text{AX}(\text{node(head)}) \ @ \ phead$$
Fig. 11. General pattern for a loop

A loop is identified by the existence of a backedge between head and tail. The back edge relation can be specified as an edge between two nodes whose source can reach its target in a backwards direction:

\[
\text{let } \text{back-edge}(\text{tail}, \text{head}) \triangleq (\text{EX node}(\text{head})) \land (\overline{\text{EF node}(\text{head})}) @ \text{tail}
\]

It is worth noting that the back-edge predicate may hold true at positions other than those within the context of loop analysis (since we allow unrestricted goto with \(L_0\)). This is acceptable for our purposes, since the loop-related transformations refer to the loop definition, defined later in this section, which specifically binds the back-edge definition to the loop tail and head nodes. We can now define the relation between the four key nodes that define a loop

\[
\text{let } \text{loop}(p\text{head}, \text{head}, \text{break}, \text{tail}) \triangleq \text{dom}(p\text{head}, \text{head}) \land \text{pdom}(\text{break}, \text{tail}) \land \text{AX(node(head))} @ p\text{head} \land \text{back-edge(tail, head)}
\]

Some optimisations such as our version of strength reduction apply only to well-structured loops, whose only entry is the head node and only exit is the break node. To identify such loops, nodes inside the loop and outside the loop are distinguished. Execution of nodes inside the loop lead eventually to the break node before either re-entering the loop or exiting the program. So all paths from nodes outside the loop reach either the exit or pre-header without going through the break

\[
\text{let } \text{out_loop} \triangleq \text{A} (\neg \text{node(break)} \cup \text{node(phead)} \lor \text{exit})
\]

An illegal jump out of the loop requires the existence of a node that is not the break node but has a successor outside the loop.

\[
\text{let } \text{out_jump} \triangleq \neg \text{node(break)} \land \text{EX(out_loop)}
\]

An illegal jump into the loop can be defined in a similar manner.

\[
\text{let } \text{in_jump} \triangleq \neg \text{node(head)} \land \overline{\text{EX(out_loop)}}
\]

These predicates allow us to define well-structured loops:

\[
\text{let } \text{wsloop}(p\text{head}, \text{head}, \text{break}, \text{tail}) \triangleq \text{loop(phead, head, break, tail)} \land
\]

\[
\text{let } \text{wsloop}(p\text{head}, \text{head}, \text{break}, \text{tail}) \triangleq \text{A}(\neg \text{out_jump U out_loop}) @ \text{head} \land
\]

\[
\text{A}(\neg \text{in_jump U out_loop}) @ \text{head}
\]
This definition of loops is used within the strength reduction and loop fusion transformations described in the following two sections. Intuitively, it captures loops that have some block of sequential instructions up to a tail node, from which an edge goes back to the loop header. Somewhere within this loop there is a node that has a branch that allows one to break out of the loop, called the break node. A pre-head node is also matched, to support the hoisting of instructions to the place immediately preceding the loop.

8. EXAMPLE TRANSFORMATIONS

This section provides examples of common optimising transformations that can be specified in \textit{TRANS}. These transformations are amongst transformations that are found in the optimising phase of many compilers.

Examples presented in this paper may differ slightly from standard presentations in the manner described above. However, combining the transformations produces similar results. In fact, specifying the transformations in this modular way supports experimenting with application of transformations in different orders to increase efficiency. The \textit{TRANS} language also makes the transformation more amenable to formal analysis, for example proving that the transformation is semantics preserving.

8.1 Dead code elimination

Dead code elimination removes a definition of a variable if it is not used in the future. The rewrite simply removes the definition:

$$ n : (x := e) \implies \text{skip} $$

The side condition on this transformation is that all future paths of computation should not use this definition of $x$ or, more precisely, there does not exist a path with a node that uses $x$ without a different instruction re-assigning to $x$ first. This can be specified using the $E(\ldots U \ldots)$ construct, noting that $x$ could be used at node $n$ itself. So the paths that should be identified are those not using $x$ until the formula $\text{use}(x) \land \neg \text{node}(n)$ holds. The final specification of dead code elimination is shown in Fig. 12.

8.2 Constant propagation

Constant propagation is a transformation where the use of a variable is replaced with the use of a constant known before the program is run (\textit{i.e.} at compile time). The standard method of finding out if the use of a variable is equivalent to the use of a constant is to find all the possible statements where the variable could have been defined, and check that in all of these statements, the variable is assigned the
same constant value. The rewrite itself is simple:

\[ n : (x := e[v]) \implies x := e[c] \]

This rewrite uses the term \( e[v] \) to find an expression \( e \) with sub-expression \( v \). Note that, as per Definition 6.1, a variable \( v \) matches against one occurrence of the sub-expression in \( e \). The rewrite then replaces the occurrence that has been matched. So if \( e \) matches the expression \( x + (x - 2) \), with \( v \) matching the left hand \( x \) and \( c \) matching 3, then the rewrite inserts the expression \( 3 + (x - 2) \).

For the rewrite to be correct \( v \) and \( c \) must be restricted so that \( v \) necessarily equals \( c \) at the given point. The idea is that if all backward paths from node \( n \) are followed, then the first definition of \( v \) encountered must be of the form \( v := c \).

As all such paths must be checked, the \( \text{← − A}(...) \) constructor is appropriate. To fit into the “until” path structure we can note that requiring the first definition on a path to fulfil a property is the same as saying the path satisfies non-definition until a point where it is at a definition and the property holds. The full specification of constant propagation is given in Fig. 13.

This transformation is equivalent to standard constant propagation as found in existing compilers. Sometimes it is useful to propagate other entities such as variables; the transformation specifications in these cases are almost identical.

There are several extensions to constant propagation that are quite specialised and require complex algebraic reasoning; a survey of their computational complexity can be found in [Müller-Olm and Rüthing 2001]. Conditional constant propagation is presented in [Wegman and Zadeck 1991]. These extensions expose a current limitation of TRANS: it doesn’t allow recursive pattern matching facilities, therefore expressions of arbitrary complexity cannot be folded. It only takes account of expressions of the form of constant op constant. Further extensions to TRANS could be used to capture these special cases.

8.3 Strength reduction

Strength reduction is a transformation that replaces multiplications within a loop structure with additions that compute the same value. This is beneficial since the computational cost of multiplication is usually greater than that of addition.

For strength reduction to apply, a basic induction variable and a derived induction variable dependent on it within the loop must be identified. Within a loop, a basic induction variable is one that has only one assignment of the form \( v := v + c \). A derived induction variable of the loop is a variable that is linearly dependent on a basic induction variable \( v \). The optimisation is illustrated in Fig. 14. We use \( n \) to denote the node at which the induction variable increments, and \( m \) to denote the node at which the derived induction variable is assigned to within the loop.

The following relation states that \( v \) is an induction variable that is incremented
at program point \( m \) with increment \( c \):

\[
\text{let basic_induction_var}(v, c, m) \triangleq -\text{out_loop} \land (\text{stmt}(v := v + c) \circledast m \land A(\text{node}(m) \lor -\text{def}(v) \cup \text{out_loop})) \circledast \text{head}
\]

A linearly derived induction variable is one that has only one assignment in the loop assigning it to a linear function of the variable it depends on. This can be detected using the predicate \textit{dependent_var}:

\[
\text{let linear_def}(w, v, k, d) \triangleq (\text{stmt}(w := v \ast k) \land d \text{ is } 0) \lor \text{stmt}(w := v \ast k + d)
\]

\[
\text{let dependent_var}(w, v, k, d) \triangleq -\text{out_loop} \land (\text{linear_def}(w, v, k, d) \circledast n \land A(\text{node}(n) \lor -\text{def}(w) \cup \text{out_loop})) \circledast \text{head}
\]

Finally, the dependent variable \( w \) is initialised before the start of the loop in the transformed program. In the first iteration of the loop the value of \( w \) between the head of the loop and its first definition will be different in the transformed program compared to the original program. To ensure the value difference does not matter the following invariant must be preserved:

\[
A(\neg \text{use}(w) \cup \text{def}(w) \lor \text{exit}) \circledast \text{head}
\]

In other words, either the variable \( w \) cannot be used between the head of the loop and its first definition within the loop or control flow leaves the loop if the variable \( w \) is not live.

When these conditions are satisfied, the strength reduction transformation introduces a fresh variable \( w' \) and replaces the assignment \( w := v \ast k \) with \( w := w' \). In order to maintain the correct value of \( w' \), it is necessary to add the assignment \( w := w + \text{step} \), where \( \text{step} \) is matched to the value of \( c \ast k \) (using the \textit{is} predicate) after execution of the assignment \( v := v + c \). In addition \( w \) must be initialised to the correct value just after the pre-header of the loop. The overall transformation is specified as in Fig. 15.

\textit{Variations.} The above transformation introduces a new variable and on its own only adds new calculations and does not make code any faster. This is a case where cleaning up transformations are needed. In particular, repeatedly applying
\[ phead : s \implies s; w' := v \cdot k + d, \]
\[ n : (w := v \cdot k) \implies w := w', \]
\[ m : (v := v + c) \implies v := v + c; w' := w' + \text{step} \]

if

\[ \text{loop}(\text{phead}, \text{head}, \text{break}, \text{tail}) \land \]
\[ \text{basic}_{\text{induction\_var}}(v, c, m) \land \text{dependent\_var}(w, v, k, d) \land \]
\[ A(\neg \text{use}(w) \lor \text{def}(w) \lor \text{exit}) \land \text{head} \land \]
\[ \text{conlit}(k) \land \text{conlit}(c) \land \text{step is } k \cdot c \land \text{fresh}(w') \]

Fig. 15. Specification of strength reduction

variable propagation and dead code elimination removes the calculations involving the original dependent variable \( w \).

Some loop strengthening algorithms (including the one in [Aho et al. 2007]) find more types of dependent variables. Specifically, dependent variables may exist that are a linear function of an induction variable but their definition is in terms of other dependent variables. Detecting this kind of dependent variable is appropriate when all loop strengthening is done in one monolithic transformation. In our framework, where small transformations are iterated, the complex approach is not necessary since repeated strengthening along with variable propagation and algebraic simplification will eventually strengthen variables such as \( w \) above.

There are even more sophisticated induction and dependent variable detection techniques that are beyond the scope of both this paper and the TRANS language. For example see [van Engelen 2001] where some advanced algebraic reasoning is required. A version that combines strength reduction with code motion [Knoop et al.] is described in Section 9.3.

8.4 Branch elimination

A jump statement such as depicted in Fig. 16(a) where two branches of a conditional lead to the same node can be rewritten to a \texttt{skip}; the two conditional edges are replaced by a sequential edge. The specification of this transformation is given in Fig. 17(a).

Branch elimination can also disconnect an unused branch of a conditional, as shown in Fig. 16(b). A pattern like this might occur after other transformations. The optimisation is based upon two transformations: one for recognising ‘always true’ conditions and one for recognising ‘always false’ conditions. The ‘always-true’ case is specified in Fig. 17(b).
9. EXAMPLE TRANSFORMATIONS USING STRATEGIES

Strategies, as described in Section 6.6, provide a way of exploiting the non-determinism of matching within the side conditions of transformations. Here we describe transformations that use strategies to alter the control flow of a program.

9.1 Skip elimination

Some transformations, such as dead code elimination, may leave skip instructions in the program being optimised. As illustrated in Fig. 18, these are unnecessary and can be removed.

A skip statement may have several predecessors but has only one successor. To remove the skip statement, it is necessary to remove all edges connected to it and add an edge between each predecessor to the successor. The strategy language is very suitable for such manipulations: a MATCH strategy locates a skip instruction and its unique successor, and the APPLY_ALL strategy performs the edge removal. The resulting specification of skip elimination is shown in Fig. 19. It uses an action remove_generic which removes possibly existing edges between two nodes independently of their type (i.e. branch, seq). Recall that an APPLY_ALL strategy executes the transformation for every matching substitution of free variables (given that n and s are bound in the MATCH strategy).

9.2 Loop fusion

Loop fusion, as illustrated in Fig. 20, is a control flow transformation which fuses two consecutive indexed loops into one. This often makes the code more time-efficient since it reduces the number of increment instructions to i and allows more opportunity for instruction scheduling.

The specification of fusion must identify two loops. To make the transformation

---

1However, this transformation may not produce faster code, especially as it can introduce worse cache behaviour.

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
let \( \text{remove}_{\text{generic}}(n,m) \triangleq \text{remove}_{\text{edge}}(n,m,\text{seq}); \text{remove}_{\text{edge}}(n,m,\text{branch}) \).

MATCH
\[ \text{stmt}(\text{skip}) \land AX(\text{node}(s)) @ n \]
IN
\[ \text{APPLY}_{\text{ALL}} \]
\[ \text{remove}_{\text{edge}}(p,n,e) \]
\[ \text{add}_{\text{edge}}(p,s,e) \]
if
\[ \neg EX_e(\text{node}(p)) @ n \]
THEN
\[ \text{remove}_{\text{generic}}(n,s) \]

Fig. 19. Specification of skip elimination

simpler, the specification shown here is restricted to loops that follow a for-like pattern, i.e. where the break node is the same as the header node and whose tail node increments the induction variable of the loop. The macro for identifying loops introduced in Section 8.3 is used twice here to verify that \( \text{loop}(p_1,h_1,h_1,t_1) \) and \( \text{loop}(p_2,h_2,h_2,t_2) \). These definitions specify the header and break nodes as the same node (since they are bound to the same meta-variable). Loops can be identified as consecutive by checking that all the second loop's pre-header's immediate predecessors are the break node of the first loop, i.e.

\[ AX(\text{node}(h_1)) @ p_2 \]

The node \( \text{cont} \) that follows the break in the second loop (i.e. where control leaves the loop) is detected with the following predicate:

\[ EX(\text{node}(\text{cont}) \land \text{out}_{\text{loop}}(p_2) @ h_2 \]

The loops must be indexed in the same manner. Firstly, both the pre-header nodes (initialising the induction variable) and the break nodes must have the same instruction:

\[ \text{same}_{\text{instr}}(p_1,p_2) \land \text{same}_{\text{instr}}(h_1,h_2) \]
where \( \text{same instr} \) is defined by:

\[
\text{let } \text{same instr}(n, m) \triangleq \exists s. \text{stmt}(s) @ n \land \text{stmt}(s) @ m.
\]

Furthermore, both loops must have a common induction variable \( x \) (in the following formula \( \text{basic induction var}_i \) is defined as in Section 8.3 for either the first or second loop):

\[
\text{basic induction var}_1(x, c, t_1) \land \text{basic induction var}_2(x, c, t_2)
\]

Since the transformation is being applied to for-pattern loops, the specification stipulates that the basic induction variable is incremented at the tails of the loops. Finally, both exit conditions of the loops must leave the loop by a seq edge:

\[
\text{EX}_\text{seq}(\text{out loop} 1) @ h_1 \land \text{EX}_\text{seq}(\text{out loop} 2) @ h_2
\]

It is also necessary for initialisation, increment and exit values (\( i, c, \) and \( k \) respectively) to be unchanged between the two loops. In the transformation presented below they are restricted to be constant literals. For the loops to be successfully fused, we need to ensure that the statements in the second loop do not depend on the first loop. To this end, the predicate \( \text{ind expr} \) holds of an expression whose components are not defined within the first loop:

\[
\text{let ind expr}(e) \triangleq \neg \exists E(\text{trans}(e) \lor \neg \text{trans}(e) \land \neg \text{out loop} 1)
\]

The predicate \( \text{independent} \) holds for nodes that do not depend on the first loop using the definition \( \text{ind expr}(e) \).

\[
\text{let independent} \triangleq \text{skip} \lor \exists e. \text{ind expr}(e) \land (\text{stmt}(\text{if } (e)) \lor \text{stmt}(\text{:= } e) \lor \text{stmt}(\text{ret}(e)))
\]

This expression can then be used to state that all the statements in the second loop are independent:

\[
\text{A(independent } U \text{ out loop} 2) @ h_2
\]

If these conditions are satisfied, the transformation will fuse the loops by connecting the tail of the first loop to the head of the second loop, the tail of the second loop to the head of the first loop and the break of the first loop to the cont node. The macro \( \text{move edge} \) is used to perform these connections as defined by:

\[
\text{let move edge}(a, b, c) \triangleq \text{remove edge}(a, b, \text{seq}), \text{add edge}(a, c, \text{seq})
\]

The increment of the induction variable from the first loop and the initialisation and break from the second loop must be removed. However, it is simpler to replace the break in the second loop with the constant condition \( \text{if } (\text{True}) \) and let branch elimination remove the edges later.

The complete specification of loop fusion is shown in Fig. 21. Note that this version implements a restricted version of fusion, as the definition of \( \text{independent} \) does not capture all independent uses within the second loop.

### 9.3 Partial redundancy elimination

Partial redundancy elimination transforms cases like the one shown in Fig. 22. The calculation of the expression \( a + b \) at node \( n \) will have already been computed if

\[ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.\]
one path is taken but not if the other path is taken. The transformation adds the calculation of the expression to the other path as well, making the calculation at node \( n \) fully redundant. The idea is that after partial redundancy elimination, common subexpression elimination can remove the calculation at node \( n \) thus improving performance of the left branch of computation.

Expression \( e \) is said to be *available* at point \( p \) if there is some point on every program path to \( p \) where the expression is calculated and if the same expression were evaluated at \( p \) it would result in the same value. This is captured by specifying that for every path backwards from that point \( p \) a calculation of the expression is reached before any of the constituents of that expression is reached. This concept of availability, as well as the notion of an expression being available on some path, are captured by the definitions:

\[
\text{let } \text{avail}(e) \triangleq \overrightarrow{A}(\text{trans}(e) \cup \text{use}(e))
\]

\[
\text{let } \text{avail_one}(e) \triangleq \overrightarrow{E}(\text{trans}(e) \cup \text{use}(e))
\]

These two notions can be combined to specify *partial availability*, defined as a point where on some paths the expression is available but not on all paths—the situation to be eliminated:

\[
\text{let } \text{partial}_\text{avail}(e) \triangleq \text{avail_one}(e) \land \neg \text{avail}(e)
\]
To eliminate the partial redundancy, calculations of an expression are placed at the point where they become unavailable—a point that has predecessors where the expression is available and predecessors where it is unavailable:

$$EX(\text{avail}(e)) \land \overline{EX(\text{avail}(e))}$$

However, these calculations should be placed where they will not cause extra calculations on other paths. Fig. 23 illustrates a case where a calculation should not be moved, as there is a chance the result will never be used. A safe place to add a computation of an expression to eliminate a redundancy at node $n$ can be identified by showing that all paths at that place must lead to node $n$, not altering any of the constituents of $e$ along the way. This extra property finalises the definition of a possible placement ($pp$) of a computation of expression $e$ to eliminate a partial redundancy at node $n$:

$$\text{let } pp(n, e) \triangleq \text{avail}(e) \land \overline{EX(\text{avail}(e))} \land \text{A}(\text{trans}(e) \cup \text{node}(n))$$

Partial redundancies are eliminated at points where an expression is calculated, that expression is partially available and every backward path either leads to a point where the expression is available or could be made available with a possible placement:

$$\text{partial_avail}(e) \land \overline{\text{A}(\text{trans}(e) \cup \text{pp}(n, e) \lor \text{avail}(e))} \otimes n$$

If there is such a node, the transformation shown in Fig. 24 places the computation of $e$ between each possible placement and each of its predecessors for which $e$ is not available. This leads to conditions where common sub-expression elimination and dead code elimination are applied to remove the calculation at node $n$. Partial redundancy elimination optimisations generally perform what is known as critical edge splitting [Knoop et al. 1992] which increases applicability of an optimisation. We assume edge splitting has been performed before this transformation is applied.

9.4 Lazy code motion

Lazy code motion is another form of partial redundancy elimination, with a more global view of moving around the calculation of an expression. Rather than just finding one partial redundancy to eliminate, it finds the “best” places to calculate any expression calculated in the code. The transformation only moves the expression as far away from the original computation as needed to remove redundancies.
MATCH
(\text{use}(e) \land \text{partialavail}(e) \land \overrightarrow{A}(\text{trans}(e) \lor \text{pp}(n, e) \lor \text{avail}(e))) @ n

fresh(h)

\text{IN}
\text{APPLY}\_\text{ALL}
\text{split}_{\text{edge}}(p, m, z, (h := e))
\text{if}
\text{pp}(n, e) @ m \land \overrightarrow{E}(\text{node}(p) \land \neg \text{avail}(e)) @ m

Fig. 24. Specification for partial redundancy elimination

This reduces any harm to the performance of register allocation on the program. Our formulation of Lazy Code Motion very closely follows that of Steffen and Knoop [Knoop et al. 1992].

The first property of interest to this transformation is \textit{down safety}. A program point is \textit{down-safe} with respect to an expression $e$ if all paths from that point reach a calculation of $e$ without redefining any of the constituents of $e$.

$$\text{let } \text{d_safe}(e) \triangleq \overrightarrow{A}(\text{trans}(e) \lor \text{use}(e))$$

Computations can be placed at down-safe points. The earliest computation point at which $e$ must be computed is one at which there exists a path backwards that has no down-safe points until it reaches a point where one of the constituents of $e$ is redefined, if it exists:

$$\text{let } \text{earliest}(e) \triangleq \overrightarrow{E}(\neg \text{d_safe}(e) \lor \text{start})$$

The points that are both down-safe and earliest provide sufficient criteria for where to place computations of $e$ to eliminate partial redundancy, but ideally the computation should be placed as close before the point of redundancy as possible. We can define places we could safely place a computation of $e$ after an earliest placement. A later placement is one that on all backward paths from that point can find a down-safe and earliest place without going through a computation of $e$ (in which case we have not gone too far).

$$\text{let } \text{later}(e) \triangleq \overrightarrow{A}(\neg \text{use}(e) \lor \text{d_safe}(e) \land \text{earliest}(e))$$

The latest placement point is now defined as being a later placement point that either computes $e$ or does not have later placement points on all of its successors.

$$\text{let } \text{latest}(e) \triangleq \text{later}(e) \land (\text{use}(e) \lor \neg \overrightarrow{A}(\text{later}(e)))$$

Computations of $e$ are inserted at points that satisfy this latest predicate. These points cover all the computations of $e$ but are as close to the original computations as possible. The transformation inserts the calculation of $e$ at this point and store the result in some new variable $h$. All the other computations of $e$ will then just use the result stored in this variable. However, this is not always ideal since a node that satisfies latest may calculate the result and put it in $h$ only to use it straight away and never later on in the program. There is no need to introduce the new calculation when the only place it would be used is the node it was introduced.
MATCH
\text{partial\_avail}(e) \land n
\text{fresh}(h)
\text{IN}
\text{APPLY\_ALL} (\text{insert}(h,e) \Box \text{remove}(h,e))

Fig. 25. Specification of lazy code motion

avoid this situation, the \textit{isolated} predicate identifies when a node will not pass on the use of a computation of \(e\):

$$\text{let isolated}(e) \triangleq AX((\neg \text{use}(e) \cup \text{latest}(e)))$$

The transformation \text{insert}(h,e) inserts a calculation of \(e\) (storing it in variable \(h\)) after nodes that satisfy \text{latest}(e) but not \text{isolated}(e):

$$\text{let insert}(h,e) \triangleq m : s \implies s' ; h := e \text{ if } (\text{latest}(e) \land \neg \text{isolated}(e)) \Box m$$

Calculations of \(e\) can be removed at program points that satisfy \text{redundant}(e), that is points that are neither \text{latest} or \text{isolated}.

$$\text{let redundant}(e) \triangleq \neg (\text{latest}(e) \lor \text{isolated}(e))$$

The removal transformation depends on where the redundant calculation occurs, so it can be written in three variations: one for when the calculation occurs in an assignment, one for when it occurs in a conditional and one for when it occurs in a return statement.

$$\text{rem assign}(h,e) \triangleq m : x := c[e] \implies x := c[h] \text{ if } \text{redundant}(e) \Box m$$
$$\text{rem branch}(h,e) \triangleq m : \text{if } (c[e]) \implies \text{if } (c[h]) \text{ if } \text{redundant}(e) \Box m$$
$$\text{rem return}(h,e) \triangleq m : \text{ret}(c[e]) \implies \text{ret}(c[h]) \text{ if } \text{redundant}(e) \Box m$$

Then \text{remove}(h,e) is the transformation that removes a redundant use of \(e\) (by using the variable \(h\) instead).

$$\text{let remove}(h,e) \triangleq \text{rem assign}(h,e) \Box \text{rem branch}(h,e) \Box \text{rem return}(h,e)$$

The complete transformation, shown in Fig. 25, finds an expression that is partially redundant and then performs all applications of both \text{insert} and \text{remove}.

9.5 Lazy strength reduction

Lazy strength reduction is a transformation that combines strength reduction with code motion. Code motion recognises paths where an expression is available; with strength reduction the expression to be reduced is not directly available but can be made available by altering the paths leading up to its calculation.

The goal is to eliminate partially redundant calculations such as of the expression \(v \ast c\) where \(v\) is a variable literal and \(c\) is a constant literal. Variable \(v\) is said to be \textit{injured} at a node if \(v\) is redefined at this node solely by addition of a constant value. Formally:

$$\text{let injured}(v) \triangleq \exists d.\text{stmt}(v := v + d) \land \text{conlit}(d)$$
The value of the expression \( v \ast c \) after execution of a node at which \( v \) is injured can be found by adding on the constant \( c \ast d \). Using the notion of injured nodes the trans predicate can be redefined for the expression \( v \ast c \) to say that either none of the constituents in the expression are redefined or that \( v \) is merely injured.

\[
\text{let } \text{trans}_\ast(v, c) \triangleq \neg \text{def}(v) \lor \text{injured}(v)
\]

This adjusted \( \text{trans}_\ast \) predicate allows the definition of the \( d_{\text{safe}} \), \( \text{earliest}_\ast \), \( \text{latest}_\ast \) and \( \text{isolated}_\ast \) predicates, analogous to the predicates used for specifying lazy code motion:

\[
\begin{align*}
\text{let } d_{\text{safe}}(v, c) &\triangleq A(\text{trans}_\ast(v, c) \cup \text{use}(v \ast c)) \\
\text{let } \text{earliest}_\ast(v, c) &\triangleq \overline{\overline{E}(\neg d_{\text{safe}}(v, c) \lor \neg \text{trans}_\ast(v, c))} \\
\text{let } \text{latest}_\ast(v, c) &\triangleq \overline{A(\neg \text{use}(v \ast c) \cup d_{\text{safe}}(v, c) \lor \text{earliest}_\ast(v, c))} \\
\text{let } \text{isolated}_\ast(v, c) &\triangleq AX(A(\neg \text{use}(v \ast c) \cup \text{latest}_\ast(v, c)))
\end{align*}
\]

Transformations \( \text{insert}_\ast(v, c, h) \) and \( \text{remove}_\ast(v, c, h) \) are defined analogously to the transformations used in the specification of lazy code motion. In addition, any node that injures the value \( v \ast c \) must be altered to update the value \( h \). The optimiser should do this only on injured nodes that have a path to a node that will use \( h \) i.e. a node satisfying redundant\((v, c)\). The adjust transformation is therefore defined as:

\[
\begin{align*}
\text{let } \text{adjust}(v, c) &\triangleq \\
\text{if } \text{step is } d \ast c &\text{ then } \\
\text{injured}(v) \cup m &\text{ else if } \\
\text{injured}(v) \cup m &\text{ endif if } \\
E(\neg \text{latest}_\ast(v, c) \lor \text{isolated}_\ast(v, c) \lor \text{redundant}_\ast(v, c)) &\cup m
\end{align*}
\]

The specification of lazy strength reduction, shown in Fig. 26, is similar to lazy code motion but uses the new predicates and the additional adjust transformation.

This specification illustrates the strength of our approach, in which small transformations are defined independently—and checked for correctness—and then combined using strategies, resulting in the full optimisations. Each individual transformation may not result in improvement of code, but it is the composition, either through specific strategies or indeed an overall loop, which results in more efficient code. And each transformation can be re-used when devising new optimisations.

Of course, matching these complicated side conditions to programs can be very tedious. If at times the match isn’t made the code is not improved. In order to obtain as many matches between side conditions and nodes in CFGs as there are, and for this we apply model checking algorithms, as explained next.

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.
10. APPLYING TRANSFORMATIONS TO PROGRAMS

The TRANS methodology has been developed to enable transformations to be specified easily and in a format that is suitable for formal analysis, and also to apply the transformations so specified to actual programs. This is why we chose to support coding of the side conditions in a language which allows matching between side conditions and CFGs to be performed mechanically. In this section we present an algorithm that converts TRANS specifications into a variant called TRANS\(\mu\), which is used to semantically model check the control flow graph. Data structures are described that efficiently manipulate sets of valuations using Ordered Binary Decision Diagrams (OBDDs).

10.1 Modal Mu-calculus

We introduce a language based on TRANS, which we call TRANS\(\mu\), where side conditions are specified using modal mu-calculus [Kozen 1983] instead of CTL. This allows use of model-checking algorithms to find matches. TRANS\(\mu\) is obtained by adding to the TRANS grammar following syntax:

\[
\text{node-condition ::= } \mu \text{ condition-var. node-condition} \mid \text{condition-var}
\]

In order to tailor the system to our needs, there are some differences between standard modal mu-calculus and the version presented here. The modal mu-calculus is frequently defined to operate over a labelled transition system. Accordingly the ♦ and □ operators are frequently parameterised with action names, for example ⟨a⟩ or [b]. Since CTL formulae do not express formulae in terms of labels, the mu-calculus we use is defined simply in terms of the transition relation between states (→). This simplification affects neither decidability nor efficiency.

Standard modal mu-calculus with both least and greatest fixed point operators forces the model checking algorithm into exponential complexity. However, translation from CTL to modal mu-calculus and restricting the fragment of modal mu-calculus needed ensures we can use the fragment which only requires a least fixed point operator, as the formal monotonicity condition is satisfied if the number of negations between a fixed point operator and any instance of a bound variable is even. Monotonicity ensures the existence of a least fixed point, and in fact, alternation free modal mu-calculus is checkable within a linear time [Cleaveland and Steffen 1992]. While this result doesn’t imply that our application process is more or even as efficient as traditional dataflow analysis, it gives some reason to believe that it is possible to develop an optimiser that matches program nodes to side-conditions in a manner efficient enough to support practical use. Our implementation of this approach, while still experimental, reinforces this conclusion that performance can be made acceptable.

The refinement to mu-calculus therefore loses nothing in expressiveness compared to CTL, from which our temporal formulae are refined. It should be noted that for both the modalities and the \(\mu\) operator backwards equivalents exist.
10.2 Conversion to TRANS\(\mu\)

In order to translate a CTL formula into a corresponding mu-calculus formula the rewrites for past and future quantifiers below are applied exhaustively:

\[
AX(\phi) = \neg EX(\neg \phi) \\
A(\phi_1 U \phi_2) = \neg (E(\neg \phi_1 U EX(True)) \lor (\neg \phi_1 \land \neg \phi_2)) \\
E(\phi_1 U \phi_2) = \mu Y.(\phi_2 \land FP) \lor (\phi_1 \land EX(Y)) \\
\neg AX(\phi) = \neg EX(\neg \phi) \\
\neg A(\phi_1 U \phi_2) = \neg (E(\neg \phi_1 U \neg EX(True)) \lor (\neg \phi_1 \land \neg \phi_2)) \\
\neg E(\phi_1 U \phi_2) = \mu Y.(\phi_2 \land \neg BP) \lor (\phi_1 \land \neg EX(Y)) \\
FP = \mu Z.\neg EX(True) \lor EX(Z) \\
BP = \mu Z.\neg EX(True) \lor \neg EX(Z)
\]

10.3 Matching side conditions

The algorithm for applying a transformation specification \(T\) (with side condition \(\phi\)) to the CFG \(G\) of the program being optimised is based on finding valuations \(\sigma\) such that \([\phi](\sigma, G)\) is true, where \([\_]\) is the semantic function for side conditions from Definition 6.5. The \texttt{solve} function computes the set of valuations that satisfy \(\phi\) in graph \(G\), that is, \((\texttt{solve}(\phi, G))^* = \{\sigma | [\phi](\sigma, G)\}\).

\textbf{Definition 10.1}

\[
\begin{align*}
\texttt{solve}(\phi_1 \lor \phi_2, G) &= \texttt{solve}(\phi_1, G) \cup \texttt{solve}(\phi_2, G) \\
\texttt{solve}(\phi_1 \land \phi_2, G) &= \texttt{solve}(\phi_1, G) \cap \texttt{solve}(\phi_2, G) \\
\texttt{solve}(\neg \phi, G) &= \texttt{solve}(\phi, G)^c \\
\texttt{solve}(\exists x.\phi, G) &= \exists x. \texttt{solve}(\phi, G) \\
\texttt{solve}(p(\overline{X}), G) &= \{\sigma | p(\sigma(X))\} \\
\texttt{solve}(\phi \not\in n, G) &= \cup : n \in Nodes(G) : \{\sigma | \sigma(n) = m\} \cap (\texttt{solve}(\phi, m, G))
\end{align*}
\]

where

\[
\begin{align*}
\texttt{solve}'(\text{node}(m), n, G) &= \{\sigma | \sigma(n) = m\} \\
\texttt{solve}'(\text{stmt}(s), n, G) &= \texttt{match}(s, I(n)) \\
\texttt{solve}'(\phi_1 \lor \phi_2, n, G) &= \texttt{solve}'(\phi_1, n, G) \cup \texttt{solve}'(\phi_2, n, G) \\
\texttt{solve}'(\phi_1 \land \phi_2, n, G) &= \texttt{solve}'(\phi_1, n, G) \cap \texttt{solve}'(\phi_2, n, G) \\
\texttt{solve}'(\neg \phi, n, G) &= \texttt{solve}'(\phi, n, G)^c \\
\texttt{solve}'(EX(\phi), n, G) &= \cup : (n, n', e) \in Edges(G) : \texttt{solve}'(\phi, n', G) \\
\texttt{solve}'(\overline{EX}(\phi), n, G) &= \cup : (n', n, e) \in Edges(G) : \texttt{solve}'(\phi, n', G) \\
\texttt{solve}'(X, n, G) &= n(X) \\
\texttt{solve}'(\mu X.\phi, n, G) &= \text{LFP}(\lambda y. (\texttt{solve}'(\phi, n \triangleright (X \mapsto y), G) \{}))
\end{align*}
\]

The predicate \texttt{match} above holds true iff the statement \(s\) is the instruction at node \(n\) and \(n(X)\) refers to extracting the valuation bound to \(X\) at node \(n\). In the last rule, LFP is the least fixed point of a function and \(n \triangleright (X \mapsto y)\) refers to binding \(X\) to the value \(y\).

10.4 Set Representation

The above algorithm for matching a transformation to a CFG needs to represent large sets of valuations and manipulate them efficiently. \texttt{TRANS}\(\mu\) valuations are
represented through Binary Decision Diagrams [Akers 1978], which are based on boolean functions that can be used to test for set membership. By ordering and reducing BDDs canonical forms for each set are created. Standardised and efficient algorithms for computing conjunction, negation, disjunction and extensional quantification for reduced OBDDs are available [McMillan 1993; Bryant 1986]. This methodology, known as symbolic model checking, has been demonstrated to be efficient, and has also been applied recently to Alias Analysis [Whaley and Lam 2004; Berndl et al. 2003].

OBDDs can represent the set of valuations, but only within a known finite domain. This finite restriction could pose a problem if the domain were infinite, and in fact the set of objects \( \mathcal{O} \) from Definition 6.2 is infinite. In manipulating a particular CFG, however, the only elements that can be used in analyses are elements of the flow-graph of the program being analysed, so in practice we are restricted to a finite domain by only transforming one program, and correspondingly one flow-graph, on any given pass.

The one exception to this required finiteness arises with the is predicate from Section 6.3, as it ranges over all integers. This problem is resolved in the following way: rather than generalising predicate ‘\( x \) is e \& \& \( \phi \)’ to any expression, a set of valuations is found for \( \phi \) using the OBDD based system described above. Once this valuation has been identified the is formula can be checked. If the valuation is unbound due to a re-evaluation of \( \phi \) then another binding is found for \( x \). In practice most uses of the is predicate fall into the aforementioned form, including all of the uses in defining optimisations in Section 8 and Section 9. Thus, the restriction doesn’t limit the expressive power of TRANS in practice.

10.5 Implementation
We have implemented the methodology described above for translating TRANS specifications and matching against a program. We have generated the intermediate code from the LCC compiler [Fraser and Hanson 1991] (rather than writing programs in \( L_0 \)) and matched the side conditions of transformations. Visualisation of the CFGs and the points of match was generated (using the udraw package) to allow checking for accuracy of matching.

11. DISCUSSION
The transformations from Sections 5, 8, and 9 demonstrate that TRANS is flexible enough to describe a wide range of existing optimising transformations. However, the language is still quite lightweight and it is not surprising that some known transformations from the optimising compiler community cannot be expressed using TRANS. Some transformations cannot be expressed due to the action language not being expressive enough to capture features such as inlining and loop unrolling. These limitations could be resolved by extending the kinds of objects which can be matched in CFGs—for example, by matching against blocks of code. Such extensions would not fundamentally change the nature of TRANS; in particular the side condition language would be the same.

Some transformations, for example conditional constant propagation [Wegman and Zadeck 1991], cannot be expressed due to the limitations of the side condition language. In [Lerner et al. 2002] a method is proposed where analyses of
conditional constant propagation can be specified as combinations of other simpler transformations. This approach could possibly be added to TRANS by introducing new combination operators to the language. This is a promising future extension to our work.

In the worst case, the language is extensible in ad-hoc ways i.e. with specific predicates that perform certain analyses. For example the predicate my_analysis could be added with the specification:

\[
\text{my_analysis}(v, c, n) \equiv v \text{ can be replaced by } c \text{ at } n \text{ due to conditional constant propagation}
\]

provided that an algorithm can be written that returns a binary decision diagram representation of this relation. In this case we can specify a transformation by:

\[
n : (x := v) \Rightarrow x := c \text{ if } \text{my_analysis}(v, c, n)
\]

Throughout this paper we have referred to the language \( L_0 \), which lacks many of the features of modern programming languages. We introduce the concept of an over-approximation through which the concepts can be carried to a richer language. We say that a CTL formula \( P_2 \) is an over-approximation of a CTL formula \( P_1 \) if for every instance that \( P_1 \) holds, \( P_2 \) must hold, but not vice-versa. One corollary is that the semantic condition implied by the formula still holds true in different languages: an implementation of transformations over some language \( L \) is correct if the relevant formulae that are over-approximations in \( L_0 \) are also over-approximations in \( L \). The idea is that the user only has to think about the transformation in terms of a simple language like \( L_0 \) and if the transformation is correct in this simple language then it will be correct in a more complicated language that extends \( L_0 \).

One of the most significant differences between \( L_0 \) and common languages is aliasing, where variables and memory locations don’t necessarily have a bijective relationship. In order to account for the inability to statically compute the exact aliasing relationship under these conditions, many algorithms have been developed that trade off accuracy of computation, with cost of computations [Aho et al. 2007]. Therefore when computing alias sets or relations one differentiates between must and may alias concepts. Variables \( x \) and \( y \) must alias each other if on all paths through the program they refer to the same memory location at some program point. They may alias each other if there exists a path where they refer to the same memory location.

Within a TRANS implementation on a language with aliasing, the use and def predicates must be redefined to take account of aliasing. Occurrences of \( \text{use}(x)@n \) (where \( x \) and \( n \) are metavariables that bind to variables and nodes, respectively) must be refined to \( \text{mayuse}(x)@n \) or \( \text{mustuse}(x)@n \) depending on the aliasing conditions. This depends upon the polarity of the use predicate. If the number of negations preceding the predicate is even the polarity is positive and the occurrence can be replaced by \( \text{mustuse} \), otherwise it has negative polarity and should be replaced by \( \text{mayuse} \). This allows an over-approximation of the original use predicate and is thus sound. The same reasoning applies to the def predicate.

The ability to conservatively over-approximate is due to the modular nature of TRANS definitions, and is a comment on the powerful nature of the underlying system. An interesting extension will be to apply transformations in an
inter-procedural setting. A partial solution to this is in [Lacey 2003]; however the inter-procedural analysis provided there is not as effective as some inter-procedural dataflow analysis e.g. [Knoop 1998]. It may be possible to take some of the ideas from the literature on this subject and provide more powerful inter-procedural analysis for transformations specified in TRANS.

One of the key benefits of describing transformations as rewrite rules with temporal logic side conditions is that there is an implementation that automatically detects when and where these transformations apply to a particular program. However, for this to be truly useful in a real world program development environment detection of applicability must be more efficient than in Section 10. In particular, when applying several transformations in sequence, the efficiency of the transformation engine would be greatly improved if an algorithm was developed that incrementally executes the program analysis needed to decide which transformation applies (i.e. re-use previous calculations used when deciding that the last transformation applied). This seems viable since the transformations change the program in a highly stylised and local manner. In addition, others have already developed incremental model checking algorithms that could be adapted [Sokolsky and Smolka 1994] and an incremental algorithm for calculating regular path queries can be found in [de Moor et al. 2003]. Incremental data-flow analysis algorithms can be found in [Landi and Ryder 1991] and [Pollock and Soffa 1989].

The analysis of transformations could also be further investigated.

A method for proving the soundness of transformations has been described [Lacey et al. 2004]; however, this method becomes quite cumbersome for proving soundness of transformations of real programming languages. Research is needed into how machine assistance could be used to help prove soundness, perhaps using a semi-automated theorem prover. How much user-interaction would be required is unknown; it is as yet unknown whether determining the soundness of transformations specified in TRANS is decidable.

Another kind of formal analysis that can be applied to transformations specified in the manner described in this paper is that of interference analysis, i.e. statically determining whether applying one transformation will cause another transformation to either apply when it did not before or not apply when it did before. The general idea is that the logical side conditions and transformations can be combined (either algebraically or via automata) to describe the situation under which interference can happen. This combined description can then be checked for validity i.e. whether any program could possibly satisfy it. The free variables in the transformation descriptions mean that this is not a trivial problem. A method is presented in [Lacey 2003] that shows how to determine interference properties for a sub-language of TRANS.

[Lacey 2003] provides a general method for refining and then executing the transformations specified in TRANS over complicated language features. We are currently in the process of implementing such a system with the intention of automating soundness checking for real programs.
12. RELATED WORK

Several systems have been developed for describing and implementing program transformations from specifications. This section details some of these systems and compares them to the approach taken here.

12.1 DFA\&OPT-Metaframe

The Metaframe system [Klein et al. 1996] is a toolkit for program analysis and transformation. The complete toolkit is a large-scale project that provides libraries and tools to aid the construction of industrial strength compilers. Part of this system is a transformation tool called the DFA\&OPT-Metaframe toolkit. This toolkit in some respects bears the closest relation to the system described here in that it also uses temporal logic as a specification language. Both systems stem from Steffen's work on data-flow analysis as model checking [Steffen 1993b], and in fact some of the transformations implemented by us have been catalogued by Steffen.

There are, however, differences between the two systems. Metaframe does not use first order temporal logic to specify properties and communicate the results of the analysis to the transformation. The analyses are specified in propositional temporal logic and the resulting program analysis functions are called from a domain specific imperative programming language similar to Pascal. The user writes a temporal logic formula that is automatically converted into a program analyser, which can then be used in the writing of a compiler. During the conversion of a temporal logic formula into an analysis function a formula is partially evaluated to a model checker and optimised to produce an analysis routine similar to that in hand-written compilers (in particular with no loss of speed over the hand-written versions).

The system based on \textsc{TRANS}, as it stands, is not as suitable for compiler construction as the approach above. Due to the high-level and general nature of \textsc{TRANS}, the implementation will not be obviously as fast as equivalent hand-written code or the more specific propositional temporal logic analysis found in Metaframe. Since Metaframe includes a full programming language, it is also more expressive. However, these disadvantages are a consequence of the higher level abstraction chosen by us, and can be remedied with further work. On the other hand, our approach has some advantages. Firstly, it supports simpler specifications than propositional temporal logic, with no need for imperative code. While the combination of \(L_0\) and \textsc{TRANS} is not as expressive as a system that includes a full programming language, it does allow for more concise side conditions. Secondly, having the side conditions and transformations combined in the same simple language with no imperative features allows us to perform formal analysis on the transformations. For example, previous work [Lacey et al. 2002; 2004] shows precisely how optimisation specifications can be proved to be sound, that is, semantics preserving.

Several aspects of the Metaframe system suggest extensions to the work presented here. In particular, the techniques of partial evaluation of model checkers with respect to a particular formula [Steffen et al. 1995] could be adapted to provide a more efficient implementation for \textsc{TRANS} than the prototype implementation which we are currently using, based on our description in Section 10 [Lacey 2003].
TYPE
Stmt: Si,Sj,Sl;

PRECOND
Code_Pattern /* Find a constant definition */
any Si: Si.opc == assign AND type(Si.opr.2) == const;

Depend /* Use of Si with no other definitions */
any (Sj,pos):flow_dep(Si,Sj,(=));
no (Sl,pos):flow_dep(Sl,Sj,(=)) AND (Si != Sl)
AND operand(Sj,pos) != operand(Sl,pos);

ACTION /* Change use of Si in Sj to be constant */
modify(operand(Sj,pos),Si.opr.2);

Fig. 27. Specification of constant propagation in GOSpeL

12.2 Genesis/GOSpeL

The Genesis system implements specifications in the transformation specification language GOSpeL, where optimisations are specified through an ACTION component (with operators for modifying, copying and removing statements of a program similar to TRANS) and a TYPE and PRECOND component which together are equivalent to the condition part of TRANS specifications.

Fig. 27 shows the specification of constant propagation in GOSpeL: we see that this specification can quite easily be converted into TRANS. The ACTION part of the specification states that the statement $S_j$ is modified by replacing a sub-term with a constant term, in TRANS this is specified as:

$$S_j : (y := e[x]) \implies y := e[c]$$

The Code_Pattern in the specification binds a statement to $Si$ which assigns a variable to a constant, which is simple to write in TRANS: $x := c @ Si \land conlit(c)$

The DEPEND part of the GOSpeL statement contains two parts, the first indicating that $S_j$ is flow dependent on $Si$, and the second that that $S_j$ is dependent on no statement other than $Si$. Flow dependence indicates whether a variable is used in a statement that has a defining instance at a point it is dependent on. It can be defined in TRANS in the following way:

let $flow_{dep}(x, n, m) \triangleq use(x) @ m \land \text{def}(x) \land E(\neg \text{def}(x) U node(m)) @ m$

The complete direct translation into TRANS of the GOSpeL specification is:

$$S_j : (y := e[x]) \implies y := e[c]$$

if

$x := c \land \text{conlit}(c) @ Si$

$flow_{dep}(x, Si, S_j)$

$\neg \exists Sl. flow_{dep}(x, Sl, S_j) \land \neg \text{node}(Sl) @ Sl$

This specification of constant propagation is quite different from the one developed in Section 8.2. The one above is slightly less applicable since all defining instances of the variable we are replacing must be defined at a single point ($Si$).
<table>
<thead>
<tr>
<th>Dependency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flow_dep(n,m)</td>
<td>a variable definition at ( m ) is used at ( n ).</td>
</tr>
<tr>
<td>anti_dep(n,m)</td>
<td>a use of a variable at ( n ) is re-defined at ( m ).</td>
</tr>
<tr>
<td>out_dep(n,m)</td>
<td>a variable definition at ( m ) is output at ( n ).</td>
</tr>
<tr>
<td>ctrl_dep(n,m)</td>
<td>( n ) is a conditional statement and ( m ) occurs after one of its branches.</td>
</tr>
</tbody>
</table>

Fig. 28. Dependencies in GOSpeL

In general, the GOSpeL language has features for matching patterns of code on individual statements and detecting four different types of flow dependencies between nodes (taken from [Padua and Wolfe 1986]), shown in the table in Fig. 28. Each of these can be expressed in TRANS, however GOSpeL also allows these dependencies to be altered with direction vectors. For example, the dependency ‘flow\_dep(n,m,\,<\,)’ states that an array element definition at \( n \) is indexed at a place before (under the order of some iterative loop) the index of an array element used at point \( m \). Such predicates cannot be written in TRANS and the language would have to be extended to handle inequality constraints. Another aspect of GOSpeL is that it can match patterns binding variables to whole blocks of code and then move and modify these to enable optimisations such as loop unrolling and inlining. Again, this is not currently possible in TRANS but could be with some conservative extensions of block matching operators (in fact, this is the approach taken by one of the authors in [Lacey and de Moor 2001]). Some of the transformations shown in this article are not investigated in GoSPeL, for example the partial redundancy elimination shown in Sections 9.3 and 9.4.

Overall, the philosophy of the TRANS approach is different from GOSpeL in that analyses in TRANS is broken down into smaller components, rather than development of specific (and potentially quite complicated) analyses. This has the advantage of increasing the expressiveness and allowing more uniform formal analysis. Nevertheless, some analysis has been done on transformations in GOSpeL, in particular an approach to prove (by pen-and-paper) that disabling interference does not occur between two transformations is provided in [Whitfield and Soffa 1997].

12.3 Optimix

Optimix is a graph rewriting system developed by Assmann [Assmann 1996; 1999]. It can be used for many purposes including specifying some of the transformations described in this paper. It bears a similarity to TRANS since it is based on modifying the control flow graph of a program. Optimix analyses a program by repeatedly applying small rewrites to its graph. Each rewrite extends the graph with nodes that do not represent part of the program but capture information about the program. By using repeated application, each individual rewrite can be quite simple and succinctly specified but combined rewrites propagate quite complex information around the graph. This information, represented as extra nodes or edges can then be used to mark where a program is to be transformed.

The use of a general method makes Optimix very expressive and useful for a variety of transformation and analysis problems. It differs from the TRANS system in that it uses the graph to store intermediate analysis information required for the transformations. The process of matching a temporal logic formula, on the other
hand, provides all this information in one step and abstracts away the detail of each step, from the point of view of the person guiding the optimisation.

12.4 Rewriting

There are numerous rewriting-based transformation systems in the field of functional programming. An early implementation of an automatic transformation system can be found in the TAMPR system, which has been under development since the early '70s [Boyle 1970; 1989]. TAMPR starts with a specification that is translated to pure lambda calculus, and rewriting is performed on the lambda expressions. Because programs are represented in a functional notation, there is no need for complex side conditions, and the transformations are all of a local nature.

OPTRAN is also based on rewriting, but it offers far more sophisticated pattern matching facilities [Lipps et al. 1988]. TrafoLa is another system able to specify sophisticated syntactic program patterns [Heckmann 1988]. The Glasgow Haskell compiler allows the programmer to add pragmas to code which allow extra rewrites to be performed on the program during the compilation process [Jones et al. 2001]. The system MAG [de Moor and Sittampalam 2001] provides similar functionality but with more advanced mechanisms for resolving side conditions that are functional equalities. TRANS differs from these systems in having side conditions. These allow a more global view of the program, and are of use when optimising imperative programs.

12.5 TTL

Kanade’s Temporal Transformation Logic (TTL) [Kanade et al. 2006; 2007] is a system similar to TRANS, that uses a CTL based proof technique. Kanade’s focus is automatic verification of the soundness of the transformations themselves. Accordingly, instead of using the generic rewriting technique that TRANS uses, TTL has a set of transformational primitives. Each primitive represents a common element used within compiler optimisations, for example replacing an expression with a variable. Each of the transformational primitives has an associated soundness condition that, if satisfied, implies the soundness of the transformation. The soundness of transformations within TTL can be proved using the PVS system, which also supports validation of a trace from an instrumented compiler.

The primary difference between TTL and TRANS is that transformation primitives in TTL are less general. Whilst Kanade is able to show that TTL allows automated soundness proofs of some sophisticated optimisations, such as optimal code placement, other common optimisations, for example constant propagation, have not yet been specified, due to the data structures used to implement TTL. The use of specific transformational primitives also raises questions about how general his system is, most notably over the need to introduce further primitives in future. TTL is also seen as a specification language, for other compiler implementations, whilst TRANS can be refined and executed as the optimisation stage of a compiler.

12.6 Cobalt and Rhodium

Lerner [Lerner et al. 2003] describes the Cobalt system, which supports automated provability and executable specifications. Rewrite rules are given with temporal conditions expressed in a more restrictive form compared to CTL, which allows the
proof, once and for all, of the basic inductive form of the proof technique which holds given sufficient optimisation specific conditions are met. The optimisation specific proof obligations can be discharged automatically using a theorem prover, since they require no inductive heuristics.

The specific nature of Cobalt’s temporal conditions, though common to many dataflow analysis, is limited compared to the generic model checking that TRANS performs with its CTL side conditions. This is one of the main motivations given for developing Rhodium [Lerner et al. 2005], which is another domain specific language for developing compiler optimisations. Rhodium consists of local rules that manipulate dataflow facts. This is a significant departure in approach from TRANS, since it uses more traditional, data flow analysis based specifications rather than temporal side conditions.

12.7 Other program transformation systems

The APTS system of Paige [Paige 1994] describes program transformations as rewrite rules, with side conditions expressed as boolean functions on the abstract syntax tree, and data obtained by program analyses. These analyses also have to be coded by hand. Other transformation systems that suffer the same drawback include Khepera [Faith et al. 1997] and Txl [Cordy et al. 1995]. Datalog-like systems express program analyses as logic programs [Dawson et al. 1996]. A more modern system is Stratego [Visser et al. 1998], which has sophisticated mechanisms for building transformers from a set of labelled, unconditional rewrite rules. The CTADEL system [van Engelen 1998] is a program transformation system that has been used, amongst other things, to transform code used in the mathematical modelling of meteorological phenomena. The Vista system [Zhao et al. 2002] is a system of interactive program transformation aimed at the optimisation of programs for embedded systems. It has a fixed selection of hard coded optimisations but has a very advanced user interface for iteratively transforming, viewing and testing code. The SHARLIT system [Tjiang and Hennessy 1992] provides a toolkit for writing dataflow analyses that generate C++ code. SHARLIT splits the specification of dataflow analyses into flow functions, action routines and simplifier rules, and is intended to provide a particularly good toolkit for global, interprocedural, analysis. The specification language, isn’t based on a temporal logic, but closely resembles custom dataflow analysis. It is evaluated with respect to paths on the CFG, but the system enables sophisticated simplification rules that allow one to specify a wide variety of analyses. This also allows it match information at every basic block and still provide an efficient solution to its data-flow equations. The solutions themselves are computed by an iterative algorithm. Whilst SHARLIT aims to provide an efficient system within which to develop dataflow analyses it doesn’t offer a system within which to verify the soundness of the optimisations.

13. CONCLUSION

This paper presents a language and framework for specifying and executing optimising transformations. A specification language for transformations (TRANS) is introduced that combines elements of rewriting and temporal logic, and this is paired with a model-checking approach which facilitates the application of optimisations to code reminiscent of compiled programs. Many example transformations,
some of which are quite sophisticated, were presented to show the flexibility of the language in specifying many useful compiler optimisations. The main thesis of this paper is that it is possible to use a formally defined language which supports rigorous analysis to specify and implement a wide range of realistic optimisations.

We have based this work on a simple imperative language, but we are in the process of porting the methodology into a more realistic setting and show that the transformations are effective. Usually, a lot of time in optimisations is spent in looking for situations where optimisations are valid and effective; the use of side conditions based on temporal logic provides an interesting solution to this issue.

14. ACKNOWLEDGEMENTS

David Lacey acknowledges the support and guidance from the Oxford University Programming Tools Group. Richard Warburton is funded by the EPSRC under grant EP/DO32466/1 “Verification of the optimising phase of a compiler”. We are grateful to the anonymous reviewers for their detailed and helpful suggestions.

REFERENCES


Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. IEE Transactions on Computers C-35 12, 1035–1044.


ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.


ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.


ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Month 20YY.