Incremental Execution of Transformation Specifications

Ganesh Sittampalam  
Oxford University Computing Laboratory  
ganesh@comlab.ox.ac.uk

Oege de Moor  
Oxford University Computing Laboratory  
oege@comlab.ox.ac.uk

Ken Friis Larsen  
IT University of Copenhagen  
ken@friislarセン.net

Abstract

We aim to specify program transformations in a declarative style, and then to generate executable program transformers from such specifications. Many transformations require non-trivial program analysis to check their applicability, and it is prohibitively expensive to re-run such analyses after each transformation. It is desirable, therefore, that the analysis information is incrementally updated.

We achieve this by drawing on two pieces of previous work: first, Bernhard Steffen’s proposal to use model checking for certain analysis problems, and second, John Conway’s theory of language factors. The first allows the neat specification of transformations, while the second opens the way for an incremental implementation. The two ideas are linked by using regular patterns instead of Steffen’s modal logic: these patterns can be viewed as queries on the set of program paths.

Categories and Subject Descriptors: D.3.4 [Processors]: Optimization, Incremental compilers, Translator writing systems and compiler generators, D.3.3 [Language Constructs and Features]: Patterns, D.3.2 [Language Classifications]: Constraint and logic languages

General Terms: Algorithms, Languages, Theory.

Keywords: program transformation, transformation specification, incremental algorithm, program analysis, logic programming, constraints, language factors, residuation operators.

1 Background

Our aim is to create a toolkit for easy specification of program transformations, and for experimenting with their use. The motivating application of such a toolkit are domain-specific optimisations, typically written by the author of a software library [14, 34]. Further-
case the *else* branch can be safely disregarded, but the above pattern would fail if there was an assignment to \( X \) in the unreachable *else* branch. Constant propagation might be more often applicable if we rule out such infeasible paths, by taking the disjunction of the above pattern and

\[
\{ \} ; \\
\{ \text{assign}(X, C), \text{const}(C) \} ; \\
\{ \text{not}(\text{def}(X)) \} ; \\
( (\text{test}(X = C) ; \{ \text{else\_branch} \}) | | \\
(\text{test}(X \neq C) ; \{ \text{then\_branch} \}) ; \\
\} .
\]

The choice between two patterns is written \( \text{else}\_branch \). The predicate \( \text{else}\_branch \) matches a dummy statement at the beginning of each *else* branch; \( \text{then\_branch} \) is defined similarly. Note that this technique of ruling out infeasible paths is in fact quite generally applicable, not just to constant propagation; other common causes of infeasibility are also easily expressed.

The structure of the paper is as follows. Section 2 gives some further details of how we specify transformations. We then turn to the problem of incremental evaluation of regular path queries. In Section 3, we outline how a simplified version of the problem can be solved incrementally via Conway’s theory of language factors. It is then shown how this partial solution can be generalised to the full problem in Section 4. Next, the implementation is discussed in some detail in Section 5, and we report preliminary performance experiments. Finally, the results are evaluated in the context of related work in Section 6.

## 2 Detailed examples

Our system transforms programs in a subset of the .NET intermediate language (IL), which is an intermediate representation similar to Java bytecode [23]. Following the example of GRIMP in the SOOT toolkit [33] our transformations operate on a higher-level representation of IL, namely syntax trees where program structure (conditionals and loops) has been made explicit. The transformation engine itself is implemented in Standard ML.

The user specifies an individual transformation named \( \text{trans} \) as a relation \( \text{trans}(\text{In}, \text{Out}) \) in Prolog, where \( \text{In} \) is the original program fragment, and \( \text{Out} \) represents its transformed counterpart. To facilitate the task of defining transformations, we have augmented Prolog with facilities for incremental evaluation of regular path queries. Furthermore the Prolog programmer can query the results of other analyses that were directly implemented in ML.

The transformations can then be combined through a strategy script, which specifies how the source program is traversed, and how the transformations are sequenced. One can freely combine transformations specified in Prolog and primitive transformations written in Standard ML. Our strategy language has been borrowed from Stratego [36]. The difference is that in Stratego, the transformation relation is specified by pure rewrite rules.

The first two examples below are exactly as implemented in our system. In the third example, we have suppressed some awkward details of working with IL.

### 2.1 Atomic propagation

Atomic propagation is a generalisation of constant propagation that also propagates assignments of the form \( X := Y \), where \( Y \) is a variable. In our extended version of Prolog, it is expressed as follows:

\[
\text{atomic\_prop}(E, F) \\
\text{base\_instr}(E), \\
\text{fromentry} \\
\{\} ; \\
\{ \text{assign\_atomic}(V, A, T) \} ; \\
\{ \text{not}(\text{def}(V)) \} ; \\
\{ \text{use\_var\_type}(V, T) \} \\
\}
\]

\[
\text{sub}(A, \text{expr\_type}(\text{local\_var}(V), T), E, F) .
\]

The argument \( E \) is the statement that we wish to transform; \( F \) is its transformed counterpart. This optimisation will only be applied to atomic statements, and this is checked with the predicate \( \text{base\_instr}(E) \).

The keyword \( \text{fromentry} \) introduces a pattern that must be satisfied by all paths from program entry to the program point where \( \text{atomic\_prop} \) is being evaluated. This path query states that \( E \) is dominated by (possibly multiple occurrences of) the assignment \( V := A \), where \( A \) is a variable or a constant. Because the representation of IL is explicitly typed, we also record the type \( T \) of \( A \). The tick marks \( \text{'} \) in front of the constituent predicates represent a subtlety that we chose to ignore in the introduction. A predicate such as \( \text{use\_var\_type}(V, T, S) \) is in fact a property of a statement \( S \). The tick mark is a device to make the parameter \( S \) implicit, because in the path query there is nothing to bind it to: it only comes into play when the predicate is matched to a concrete atomic statement, during the evaluation of the path query. Predicates such as \( \text{def\_var}(X, S) \) can be defined by the user. A naive implementation would only match assignments \( S \) where \( X \) occurs on the left-hand side of an assignment. A more sophisticated definition queries the result of a may-alias analysis implemented in ML. The negation operator \( \text{not} \) negates a constraint, and it is thus subtly different from the \( \text{not} \) operator normally found in Prolog. The use of constraints is fully explained in Section 4.

Finally, if all the above conditions are satisfied, we create the new statement, using \( \text{sub}(S, T, E, F) \). This predicate replaces all occurrences of the subexpression \( T \) in \( E \) by \( S \), yielding the result \( F \). Note that we have not compromised the declarative nature of Prolog. The new program fragment \( F \) is created here, but the actual modification of the source program is done via a strategy script.

### 2.2 Dead assignment elimination

So far we have only considered conditions on all paths from program entry to the point of transformation. The converse direction, paths to the program exit, is also useful. An example in point is dead assignment elimination:

\[
\text{dead\_code}( \\
\text{instr}(\text{Labs}, E, \text{Annot}) , \\
\text{instr}( \text{Labs}, \text{exp}(\text{expr\_type}(\text{apply}(\text{nop, nil, void})))) , \\
\text{Annot}) \\
\text{not}(\text{nonnull}(E)) , \\
\text{toexit}( \{ \text{assign\_var}(V, \text{Exp} \_\_), \text{pure}(\text{Exp}) , \text{atnode}(N) \} ; \\
\{ \text{not}(\text{def\_var}(V)) ; \text{cnot}(\text{atnode}(N)) \} ; \\
\{ \text{e} \} \\
\{ \text{\{def\_var}(V) ; \text{cnot}(\text{use\_var\_type}(V)) \} ; \{ \} \} .
\]

This specifies when an assignment \( E = (V := \text{Exp}) \) at program point \( N \) can be replaced by a \( \text{skip} \) statement — in IL, this is represented
by the \texttt{nop} instruction. As an aside, note that the \textit{Annot} part of an instruction is typically used for storing results of ML analyses that can then be accessed by Prolog predicates.

Again this transformation only applies at the leaves of the abstract syntax tree. The \texttt{topout} pattern says that there are no uses of the variable \texttt{V} until we encounter the program exit (the \texttt{e} case) or another definition of \texttt{V} that does not use \texttt{V} itself. Because the assignment may occur in a loop, a use of \texttt{V} at \texttt{N} should be disregarded: otherwise the assignment could keep itself alive. This is checked using the primitive predicate \texttt{atnode}. The primitive connective \texttt{cor} computes the logical disjunction of its argument constraints.

### 2.3 Strength reduction

Consider a loop \( L = \text{while}(\text{Cond}, \text{Body}) \). Furthermore, assume that each execution of the \texttt{Body} increments \( I \) by one at least once, and that there are no other assignments to \( I \) except these increments. The key step of strength reduction is to identify an expression \( I \cdot C \) in the body, where \( C \) is not changed in the loop.

If these conditions are satisfied, we can replace the original program fragment by \( (X := I \cdot C; \text{while}(\text{Cond}, \text{Body}^p)) \). The new version \texttt{Body} of the loop body is the same as the original \texttt{Body}, except that each occurrence of \( I \cdot C \) has been replaced by \( X \). Furthermore, after every increment of \( I \) we insert the corresponding assignment \( X := X + C \). One might argue that this transformation would still be correct even if \( I \) is not changed at all in the loop. Without this condition, however, strength reduction is likely to interfere with invariant hoisting, which seeks to hoist maximal invariant expressions, and \( I \cdot C \) may be non-maximal.

Our intuitive understanding of strength reduction can be translated into the specification language as follows:

\[
\text{strength}\_\text{red}(\text{while}(\text{Cond}, \text{Body}), \text{seq}(\text{Init}, \text{while}(\text{Cond}, \text{Body}^p)))
\]

\[
\begin{align*}
\text{paths}&(\text{Body}, \{\text{cnot}(\text{def} \cdot \text{var}(I)) \text{ cor} \cdot \text{incr}(I))^*; \\
& \{\text{incr}(I)\}; \\
& \{\text{cnot}(\text{def} \cdot \text{var}(I)) \text{ cor} \cdot \text{incr}(I)^*\})^*,
\end{align*}
\]

\[
\begin{align*}
\text{paths}&(\text{Cond}, \{\text{cnot}(\text{def} \cdot \text{var}(I))\})^*, \\
\text{times}(\text{E}, \text{I}, \text{C}), \\
\text{occurs}(\text{E}, \text{Body}), \text{pare}(\text{C}), \\
\text{paths}(\text{Body}, \{\text{transparent}(\text{C})\})^*, \\
\text{paths}(\text{Cond}, \{\text{transparent}(\text{C})\})^*, \\
\text{fresh}\_\text{var}(\text{int} \cdot \text{X}), \text{subst}(\text{X}, \text{E}, \text{Body}, \text{Body}^p), \\
\text{incr}(\text{I}, \text{S}), \text{plus}(\text{PC}, \text{X}, \text{C}), \text{assign}\_\text{var}(\text{X}, \text{PC}, \text{int}, \text{XPC}), \\
\text{subst}(\text{seq}(\text{S}, \text{XPC}), \text{S}, \text{Body}^1, \text{Body}^p), \\
\text{assign}\_\text{var}(\text{X}, \text{E}, \text{int}, \text{Init})
\end{align*}
\]

This transformation relation maps a \texttt{while} statement to an initialisation followed by a transformed \texttt{while} statement. The primitive paths\((B, P)\) states that all paths through \( B \) satisfy the pattern \( P \). Here we check that \( I \) is incremented by one on every path through the body, and that such increments are the only assignments to \( I \). Next, the predicate \texttt{times}(\text{E}, \text{I}, \text{C}) binds \texttt{E} to an expression that is the multiplication of \( I \) and \( C \). Given such an \( E \), we look for occurrences of \( E \) in the \texttt{Body}. At this stage in the query, \( C \) is bound to a ground term, and we can verify that it is pure (free of side effects). Next it is checked that all paths through the loop are transparent to \( C \), in other words that the value of \( C \) does not change throughout the loop.

When all these conditions are satisfied, a new variable name \( X \) is generated, and all occurrences of \( E = I \cdot C \) in the \texttt{Body} are replaced by \( X \). Furthermore all increments to \( I \) are followed by an update of the value of \( X \), so that the invariant \( X = I \cdot C \) is maintained throughout the loop. Finally, we construct the initialisation, which sets \( X \) to \( I \cdot C \) at the beginning of the loop. The construction of the new code in is admittedly somewhat clumsy in Prolog, and we plan to remedy that using the mechanism of [35], which focuses on merging the concrete syntax of an object language into Prolog in a natural way.

Further example specifications in notations very close to this paper can be found in [12, 17, 18, 19, 21].

### 3 Chips and chops for incremental checking

We regard our transformations as rewrite rules that operate on the abstract syntax tree of a program. This deviates from the accepted practice in optimising compilers, where one transforms basic blocks in the flow graph. Each rewrite rule has an applicability condition, specified as a regular pattern. The problem of checking the applicability condition of a transformation boils down to checking a language containment, namely:

\[
\text{program}_\text{paths} \subseteq \text{pattern}
\]

The pattern may contain free meta-variables (as opposed to program variables) that we wish to solve for. The program paths include at least all sequences of atomic statements that may occur during program execution.

In order to obtain an accurate analysis, it is desirable to constrain the language of program paths as much as possible. We stipulate, therefore, that it is given by a context-free grammar, so that recursive procedures can be modelled accurately. The pattern, by contrast, is a regular expression.

To demonstrate the main ideas of our approach, we outline how the above problem can be solved incrementally. In doing so, we make three temporary simplifying assumptions. First, the alphabet of the pattern is the same as that of the program, consisting of atomic statements. Second, the pattern should not contain meta-variables. Finally, the object language is restricted to structured programs without procedures. In Section 4 it is shown how to process queries where the alphabet consists of propositions that may contain free variables: each such proposition describes a property of an atomic statement. The generalisation to interprocedural analysis is considered in Section 5.

### 3.1 Chip and chop

Recall from the introduction that our approach to incrementality is to devise a compositional analysis, so that when \( P_1, P_2 \) is analysed, and either of \( P_1 \) or \( P_2 \) is changed, the other one does not need to be re-analysed. More formally, we would like to have an algorithm \texttt{B} that satisfies the specification \( \texttt{Bpat}_{\text{prog}} = \text{prog} \subseteq \text{pat} \), and that is compositional in the following sense:

\[
\begin{align*}
\texttt{Bpat}(\text{prog}_1;\text{prog}_2) &= \texttt{Bpat}_{\text{prog}_1} \circ \texttt{Bpat}_{\text{prog}_2} \\
\texttt{Bpat}(\text{prog}_1|\text{prog}_2) &= \texttt{Bpat}_{\text{prog}_1} \oplus \texttt{Bpat}_{\text{prog}_2} \\
\texttt{Bpat}(\text{prog})* &= (\texttt{Bpat}_{\text{prog}})^*
\end{align*}
\]

for appropriate operators \( \circ, \oplus \) and \( * \). A moment’s reflection shows, however, that no such algorithm \texttt{B} can exist: the Boolean result does not carry enough information to validate the above equations.
For example, let us suppose that pat is $a{:}b$. Then, if \( \text{prog}_1 \) is $a$ and \( \text{prog}_2 \) is $b$ then $\text{false} \otimes \text{false}$ would be true. However, if \( \text{prog}_1 \) is $a$ and \( \text{prog}_2 \) is $c$ then $\text{false} \otimes \text{false}$ would be false.

To attain compositionality, it will be necessary to generalise from just a single pattern to all the "parts" of that pattern. If we have solved the problem for the parts, it would be possible to paste partial solutions together after each transformation, to obtain a solution for the whole pattern. We shall need a set of results, namely $\mathcal{B}$\( \text{pat} \)\( \text{prog} \) for each part $\text{pat}$ — such 'strengthening of the induction hypothesis' is very common when trying to achieve a compositional algorithm. But what is the appropriate notion of "part" for a regular language? To answer this question, we first introduce the chip and chop operators, and use these to give a definition of "part".

We define the chip operator $\setminus$ on languages $R$ and $S$ as follows:

$$\forall T: \quad T \subseteq R \setminus S \iff R \setminus T \subseteq S$$

Informally, one could think of the chip $R \setminus S$ as $R$ with $S$ chopped off from the front. This is illustrated by the diagram in Figure 1. Throughout this section, we shall use such diagrams to aid intuition, but the reader is warned not to use them as a formal basis for reasoning. For example, from the pictorial interpretation, one might think that $R \setminus R = \epsilon$ (the empty word), but in fact we have $a^* \setminus a^* = a^*$.

There is a dual operator to chip, which is called chop and denoted $/$. Chop is defined by the equivalence

$$\forall T: \quad T \subseteq S / R \iff T ; R \subseteq S$$

One could think of the chop $S / R$ as $S$ with $R$ chopped off from the back, as shown in Figure 2.

These operators were studied in the context of regular algebra by John Conway, who named them factors [9]. More generally they are known as residuation operators and they feature in many different areas of theoretical computer science, ranging from software specification, through non-commutative linear logic to Lambek grammars in computational linguistics. The authors first learned of these operators and their theory from Roland Backhouse [2, 3, 4].

For the present purposes, there are certain important facts about chip and chop. First, a language $S$ is regular if and only if its number of chops (that is the cardinality of \( \{ S/R : R \text{ is any language}\} \)) is finite.

Second, chips and chops naturally lead to the desired notion of "part" of a language. A part of $S$ is something that could occur as a contiguous segment in the middle of $S$, that is a language of the form $X \setminus S / Y$. As suggested by the diagram in Figure 3, chop and chip associate with each other:

$$X \setminus S / Y = X \setminus (S / Y)$$

To illustrate our definition of parts, the parts of the single word $S = \{ 0, \epsilon, a, ab, abc, b, bc, c, (a | b | c) \}$ has four chops, and eight distinct parts.

The following equation says that any such part of $S$ can be obtained as the chip of two chops:

$$\forall X, Y : \exists R_0, R_1: \quad X \setminus S / Y = (S / R_0) \setminus (S / R_1)$$

This equation is illustrated in Figure 4 — again, this is merely to build intuition, and a formal proof uses the fact that the operators $\setminus$ and $\setminus$ are Galois adjoints. Since we know the pattern has a finite number of chops, there are a finite number of parts of the pattern. In fact, if we were to list all pairs of chops, the chips of these pairs would be all the parts of the pattern.

Inspired by these facts, Conway defines the chip-chop matrix of a fixed regular language $S$. This matrix $M(S)$ is indexed by the finite set of chops of $S$, and the individual entries are given by

$$M(S)_{X , Y} = X \setminus Y$$

One could think of $M(S)$ as a systematic organisation of all the parts of $S$. In particular $S$ itself occurs as at least one of the entries of $M(S)$.

We are now ready to generalise from the original problem (which was to determine whether all program paths are in $S$) to a problem that is easier to solve compositionally, namely whether all program paths are in each part of $S$. That is, for a set of program paths $P$, we define a Boolean matching matrix $B(P)$, indexed by the chops of $S$:

$$B(P)_{X , Y} = P \subseteq M(S)_{X , Y}$$

Since $M(S)$ contains all the parts of $S$, this matrix answers the question whether $P \subseteq S'$ for any part $S'$ of $S$, in particular, for $S$ itself.

It turns out that $B$ can be computed as a homomorphism of grammars, that is the compositional form that we set out to achieve:

$$B(P_0 ; P_1) = B(P_0) \cdot B(P_1)$$
paths from program entry, and that we wish to check if the program entry up to and including $P$ can be calculated by taking the paths from program entry up to $P$ and attaching the paths that $P$ itself can take. Thus, the containment that needs to be checked is

$$\text{paths}(P) \subseteq S$$

In the preceding paragraphs, we outlined an algorithm where this containment can be checked while building up the left-hand side regular expression. Hence, following the shape of the attribute grammar in Figure 5, we can introduce three attributes $\text{paths}$, $\text{to}$, and $\text{from}$ for checking the relevant containments. Each of these attributes is a Boolean matrix.

To compute this information incrementally, we make one pass over the syntax tree prior to the transformation process, labelling each node with its $\text{paths}$ attribute.

Next, we apply the transformations. To attempt a transformation, we walk from the root to the node that we wish to transform, computing the inherited attributes $\text{to}$ and $\text{from}$ as we go along. Such tree walks are a natural part of a transformation system that operates on abstract syntax trees. In our system, the precise nature of the walk is specified in the strategy language (which is borrowed from Stratego [36]).

After each subtree replacement, we naturally have to re-compute the $\text{paths}$ information on any new nodes. Furthermore, it is necessary to recompute this attribute on all nodes to the root of the tree, but nowhere else. The new $\text{paths}$ values can then be used to compute new $\text{to}$ and $\text{from}$ information at a new potential transformation point, by walking down from the root.

### 3.3 Efficiency analysis

We now analyse the efficiency of the above procedures, assuming we are applying a simple transformation (such as constant propagation or dead code elimination) that has a single path query $S$ as the side condition. There are three stages to consider: the preprocessing, where the tree is decorated with $\text{paths}$ attributes; the search for a point to apply the transformation; and finally, the subtree replacement. Typically such a search is made by one sweep over the tree, applying the transformation wherever possible.

Let $N$ be the size of the program under consideration. The initial preprocessing pass over the syntax tree takes time $O(c(S) \cdot N)$, where $c(S)$ is the cost of an operation (multiplication, addition or closure) on Boolean matrices. This cost depends on $S$ because the dimension of the chip-chop matrix is the number of chops (which equals the number of chips). We shall consider an upperbound on $c(S)$ shortly.

Next one needs to walk the tree in search of places to apply the transformation. Each move (up, down or to a sibling) takes $O(c(S))$ Boolean operations, and there are $O(N)$ such moves in a complete sweep. It follows that the total time spent in searching for subtrees to transform is bounded by $O(c(S) \cdot N)$.

When the transformation is applied, one subtree is replaced by another. The new subtree needs to be attributed with fresh $\text{paths}$ attributes, and in the worst case this will take time $O(c(S) \cdot Q)$, where $Q$ is the number of new nodes in the subtree. All the $\text{paths}$ at-
tributes on the path from the subtree to the root need to be recomputed as well, for a total of $O(c(S) \cdot (Q + D))$ where $D$ is the depth at which the subtree replacement occurred. Note that it is not necessary to re-start the tree walk to identify the location of the next transformation: we can continue from the point of the last rewrite. In this case one needs to walk down again, along the path from the root, to the new subtree, computing $S \delta_0$ and $S \delta_1$ from attributes along the way. The cost of this is $O(c(S) \cdot D)$, so there is no increase in asymptotic complexity.

It remains to estimate an upper bound on $c(S)$. Note that $c(S)$ is a quantity that depends on the pattern $S$, in fact on the number of chaps $C$. The chip-chop matrix has $C^2$ entries. It is clear that the implementation of $\langle \cdot \rangle$ is $O(C^3)$ (using naive matrix multiplication) and $\langle \cdot \rangle$ takes $O(C^2)$ steps. Let $P$ be a program (a set of sequences of atomic statements). To establish a bound on the closure operator $P^*$, define

$$F(T) = B(\varepsilon) \wedge B(P) \cdot T$$

We compute the greatest fixpoint of $F$ by iterating from $B(\varepsilon)$. The question is how many iterations suffice to reach that fixpoint. To answer that, consider a part $X$ of $S$. After $n$ steps of the above iteration, the matrix entry corresponding to $X$ will tell us whether

$$\varepsilon \mid P \mid P^2 \mid P^3 \mid \ldots \mid P^n \subseteq X$$

Another way of thinking about this is that we determined what states of the deterministic automaton $A$ for $X$ are reachable via powers of $P$ up to $n+1$, and then whether all such reachable states are final in $A$. In the worst case, each iteration of the fixpoint computation adds only one more state to those that were reachable before: the number of iterations to reach the correct entry at the position corresponding to part $X$ in $P^*$ is thus bounded by the number of states in $A$. Because each such state corresponds to a chip $w \setminus X$ of $S$ where $w$ is a word (via the well-known derivatives of Brzozowski [7]), the number of states in $A$ is at most the number of chips of $X$, say $C_X$. Since each part of $X$ is also a part of $S$, we have $C_X \leq C$. It thus follows that $C$ iterations suffice to reach a fixpoint. As each iteration takes $O(C^3)$ steps, we have

$$c(S) = O(C^4)$$

One may now wonder whether there is a bound on $C$ given $S$. It is rather hard to give a tight bound. Let $k$ be the size of the minimal deterministic automaton $A$ for $S$. Above we remarked that $k \leq C$. Conway proves that in the worst case $C = 2^k$, but this only happens in pathological examples. In practice the patterns that arise in program transformation seem to have $C = O(k)$.

Summarising the above discussion, the number of Boolean operations for $P$ applications of a transformation (each of which introduces at most $Q$ new nodes in the tree), during one sweep of a program of size $N$ whose syntax tree has depth $D$ is

$$O(C^4 \cdot (N + P \cdot (Q + D)))$$

Here $C$ is the number of chaps of the regular pattern. With regard to space complexity, every node in the syntax tree needs to be decorated with a matrix, so the space requirements are $O(C^2 \cdot N)$.

It is interesting to compare the above time complexity bound with the best known non-incremental algorithm [22]. That algorithm takes $O(k \cdot E + x \cdot N)$ steps, where $k$ is the number of states in deterministic automaton for the pattern, $E$ the number of flow edges in the program, $x$ the number of transitions in the pattern, and $N$ the number of program statements. An estimate for the application of transformations via the non-incremental method is therefore

$$O(P \cdot (k \cdot E + x \cdot N))$$

We conclude that the incremental algorithm will outperform the straightforward method on all but the smallest instances.

4 Solving for propositions and variables

Thus far, we have developed an incremental method for getting a true or false answer to the language containment problem:

$$paths \subseteq pattern$$

While this is closely related to the problem we started off with, it is a simplification in at least two ways. First, in our program analysis examples, the alphabet of the pattern consists of propositions. Unlike traditional symbols in formal language theory, two of these propositions can match a program statement simultaneously. Second, the pattern may contain free variables, and thus in general we wish to compute a constraint on the values of these variables, instead of just a Boolean answer.

In this section, we shall show how this more general problem can be formalised. Next, we shall discover that the generality is only superficial: in fact, the general problem can be reduced to an instance of the simple containment, by suitable manipulation of the pattern.

4.1 Formal specification

To formalise the general problem, we give symbolic names to the propositions in the pattern. For example, recall the pattern for (the simplest form of) constant propagation:

$$\{ \cdot \} : \{ \text{assign}(X, C) \cdot \text{const}(C) \} ; \{ \text{not}(\text{def}(X)) \} \cdot \{ \text{use}(X) \}$$

There are four propositions in this pattern:

$$p_0 = \text{true}$$

$$p_1 = \text{assign}(X, C) \wedge \text{const}(C)$$

$$p_2 = \text{not}(\text{def}(X))$$

$$p_3 = \text{use}(X)$$

The set of names of propositions is called $prop$. For this particular pattern, we have $prop = \{p_0, p_1, p_2, p_3\}$. In what follows the pattern will be interpreted as a set of strings over $prop$, that is (using ML notation for types) an element of $proplistset$.

The validity of a proposition can be expressed as a constraint. A constraint is an equation $X = t$, where $X$ is a meta-variable and $t$ is a program term, or a negation of such an equation. Furthermore constraints may be combined with logical conjunction and disjunction. The set of all constraints is denoted $cons$.

An atomic statement is modelled by a function of type $prop \rightarrow cons$, which returns the weakest constraint that validates the proposition at that statement. For instance, the assignment $a := 3$ is modelled by the function

$$p_0 \mapsto \text{true}$$

$$p_1 \mapsto (X = a \wedge C = 3)$$

$$p_2 \mapsto X \neq a$$

$$p_3 \mapsto \text{false}$$
We thus make the type definition
\[
\text{stat} = \text{prop} \to \text{cons}
\]
According to this definition, a set of program paths is a set of lists of such statements. The program itself can now be modelled as an element of \text{stat list set}.

The natural way to compare a program path and a string in the original pattern is to use pointwise application followed by conjunction. We therefore define an operator \((\odot) : \text{stat list} \to \text{prop list} \to \text{cons} by
\[
[s_1, s_2, \ldots, s_n] \odot [p_1, p_2, \ldots, p_m] =
\begin{cases}
  s_1[p_1] \land s_2[p_2] \land \ldots \land s_n[p_n], & \text{if } n = m \\
  \text{false}, & \text{otherwise}
\end{cases}
\]

Given a program (a set of lists of statements) \(\text{prog}\) and a pattern (a set of lists of propositions) \(\text{pat}\), we wish to find the weakest constraint that implies that each path in the program validates some string in the pattern. To wit, our task is to compute the constraint which implies that each path in the program validates some string in the original pattern. To be precise, we shall use the chip-chop matrix of the lifted pattern, \(M(\text{pat}')\), and define the matrix \(A(\text{pat}, \text{prog})\) by:
\[
A(\text{pat}, \text{prog})_{ij} = \land s : s \in \text{prog} : (\forall ps : ps \in \text{pat}' : s \odot ps)
\]

It is worthwhile to pause at this point, and check that the above is indeed a generalisation of our earlier language containment problem. Given a set of symbols \(\Sigma\), and a symbol \(s \in \Sigma\), define \(\text{fun}[s] : \Sigma \to \text{bool}\ by
\[
\text{fun}[s](p) = (s = p)
\]
This operation is lifted to sets of strings over \(\Sigma\) in the obvious way. Now we have
\[
X \subseteq Y \equiv \text{C(fun}(X), Y)
\]
In other words, the new specification is a generalisation of the one we considered before. Our task is now to show that this generality is superficial, and that in fact the new problem is an instance of the old one.

### 4.2 Reduction to containment

Intuitively, the main problem is to deal with the fact that multiple propositions can be true of a single statement simultaneously: unlike the usual case, the ‘symbols’ that are propositions can overlap, when multiple propositions are true of the same statement. Our solution is to change the alphabet of the pattern from single propositions to sets of propositions.

Formally, this lifted pattern may be constructed as follows. Given \(\text{pat}\), of type \(\text{prop list set}\), we construct a new pattern \(\text{pat}'\) (which has type \(\text{prop set list set}\) by:
\[
\text{pat}' = \{ xs \mid \exists x \in \text{pat} : x \in \text{xs} \}
\]
where \(\in\) is pointwise membership on lists of sets, \(i.e.:
\[
\{ x_1, x_2, \ldots, x_n \} \in [x_1, x_2, \ldots, x_m] =
\begin{cases}
  x_1 \in x_1 \land x_2 \in x_2 \land \ldots \land x_n \in x_n, & \text{if } n = m \\
  \text{false}, & \text{otherwise}
\end{cases}
\]

In other words, a list of proposition sets is in the lifted pattern if a list of propositions in the original pattern can be made by taking one element from each set.

To parallel the above definition of lifted patterns, we define \(\bar{\circ}\), an analogue to \(\circ\), to compare a program path and a lifted pattern:
\[
[s_1, s_2, \ldots, s_n] \bar{\circ} [p_1, p_2, \ldots, p_m] =
\begin{cases}
  s_1[p_1] \land s_2[p_2] \land \ldots \land s_n[p_n], & \text{if } n = m \\
  \text{false}, & \text{otherwise}
\end{cases}
\]

The application \(s_i(p_s)\), where \(s_i\) is a statement and \(p_s\) is a \(\text{prop set}\), is defined by:
\[
s([p_1, \ldots, p_n]) = s(p_1) \land \ldots \land s(p_n)
\]
In other words, a path in a lifted pattern matches a path in the program if each statement in the program path satisfies all the propositions in the corresponding set in the pattern path.

Some elementary reasoning now shows that
\[
\land p : p \in s \circ p \equiv \forall ps : ps \in s(p) : s \odot ps
\]
As expected, to perform this computation, we shall use the chip-chop matrix of the lifted pattern, \(M(\text{pat}')\), and define the matrix \(A(\text{pat}, \text{prog})\) by:
\[
A(\text{pat}, \text{prog})_{ij} = \land s : s \in \text{prog} : (\forall ps : ps \in M(\text{pat}')_{ij} : s \odot ps)
\]
The element of \(A(\text{pat}, \text{prog})\) corresponding to the appearance of \(\text{pat}'\) in \(M(\text{pat}')\) will be precisely \(C(\text{pat}, \text{prog})\).

We now proceed to outline how \(A(\text{pat}, \text{prog})\) is very closely related to the matching matrix \(B(\text{prog})\) (where \(\text{pat}'\) is the implicit pattern) that was introduced in Section 2. In fact, modulo the use of free variables (and hence of substitutions), the two are identical. It is thus the case that the construction of the lifted pattern \(\text{pat}'\) reduces the general problem to an instance of the simple case that we have solved already. To make this conjecture precise, we first discuss substitutions.

A constraint can be viewed as a set of substitutions, one for each possible valuation of the constraint. The above equality for \(C(\text{pat}, \text{prog})\) is thus equivalent to saying that
\[
\Phi(C(\text{pat}, \text{prog})) =
\Phi(\land s : s \in \text{prog} : (\forall ps : ps \in \text{pat}' : s \odot ps))
\]
for any substitution \(\Phi\). Here we use \(\Phi(\bar{\text{w}})\) to denote the application of \(\Phi\) to the constraint \(C\) (i.e. \(\Phi(C)\) is true if \(\Phi\) is a possible valuation of \(C\)). Note that \(\Phi(\bar{\text{w}})\) distributes over conjunction and disjunction.

We can lift the application of substitutions to statements, thus turning a function from propositions to constraints into one from propositions to Booleans, which we shall choose to instead view as sets of propositions:
\[
\Phi(s) = \{ p \mid \Phi(s(p)) \} 
\]
This can be extended pointwise to lists of statements:
\[ \phi([s_1, \ldots, s_n]) = [\phi(s_1), \ldots, \phi(s_n)] \]
and from there to programs:
\[ \phi(prog) = \{ \phi(s) \mid s \in prog \} \]
Note that since this definition is pointwise, it distributes through sequential composition:
\[ \phi(prog_1; prog_2) = \phi(prog_1); \phi(prog_2) \]
One can now show that
\[ \phi(A(pat, prog)_{i,j}) = \phi(prog) \subseteq M(pat^t)_{i,j} \]
This exposes the correspondence between \( A(pat, \omega) \) and the matching matrix \( B \) of Section 2, and we can use the same algorithm to compute \( A(pat, \omega) \). The only difference is that the logical operations manipulate constraints rather than Boolean values.

4.3 Efficiency analysis
Above it was established that the algorithm of Section 3 works for patterns over propositions, and in the presence of free variables. The complexity analysis that we conducted earlier therefore applies, but there are a number of additional considerations.

First, the factor \( c(S') \) now refers to the lifted version of \( S \). It is easy to construct \( S' \) from an automaton for \( S \), but the resulting automaton is non-deterministic; in general the number of states in the minimal automaton for \( S' \) is higher than that for \( S \). This is not a problem in practice, however: for atomic propagation, dead code elimination and common subexpression elimination the size of the lifted automaton is 5. Furthermore, the number of chips is also 5.

Second, all the logical operations are now applied to constraints, rather than to Boolean values. There is thus an additional factor to consider, which is the cost of these logical operations — they can no longer be regarded as constant-time. Below we shall discuss how the constraints can be represented via binary decision diagrams, which leads to a worst-case bound of \( O(F^V) \), where \( F \) is the size of the finite domain, and \( V \) the number of variables. This is the same as the additional factor to handle free variables in the non-incremental algorithm of [22]. In particular, our previous conclusion that the incremental algorithm is asymptotically better remains valid for patterns that consist of propositions containing free variables.

5 Implementation
Having established the theoretical foundations, we now turn to the details of implementing these ideas for a real programming language — one with procedures, expressions and local variables.

5.1 BDDs to represent constraints
The usual problem with machine representation of constraints is that any representation that can be efficiently queried for solutions is subject to exponential blowup in the size of the representation if an unfortunate series of operations is applied. Binary decision diagrams (BDDs) [6] are the standard representation (they are used for a wide range of applications) that tend to minimise the occurrence of such blowups in practice.

In its purest form, a BDD is a logical formula on Boolean variables. Each BDD is conceptually a decision tree with Boolean variables for nodes, true and false for edges, and 0 and 1 for the leaves. If following the tree for a particular valuation for the Boolean variables leads to a 1, then the formula is true for that valuation, and otherwise it is false. Each decision tree is actually stored as a pointer into a single global directed acyclic graph; this graph is subject to the invariants that variables must always occur in the same order no matter what path is taken through the graph, and that no node can have the same child for both its false and its true edges. Amongst other things, these invariants guarantee canonicity of representation, so that logical equivalence of BDDs can be tested just by comparing the pointers.

Of course, our meta-variables are not Boolean; however, we can treat them as having a finite domain and make a conservative estimate of the maximum number of elements that domain will contain. Suppose this maximum is \( F \); then we number the elements from 0 up to \( F - 1 \), and represent each meta-variable by a vector of \( \log_2(F) \) Boolean variables. The atomic constraint \( X = t \) can then be represented by matching up the vector of Boolean variables for \( X \) against the binary representation of the number corresponding to \( t \). It follows that for \( V \) variables, the maximum size of a BDD representing a constraint is \( 2^{\log_2(F^V)} = F^V \). Since all operations take time at most proportional to this maximum size, we obtain the \( O(F^V) \) bound promised in Section 4.3.

Although we do not know all the elements of the domain in advance, since transformations might cause new elements to be created, we can simply give elements numbers as they appear. In fact, since distinct meta-variables may have distinct domains (some may refer to literal values only, some only to program variables and some to expressions, for example), it makes sense to maintain separate mappings (“value tables”) for each meta-variable. This allows the size of the vector of Boolean variables to be reduced, which improves the performance of the BDD operations. Even better would be to allow explicit type declarations for meta-variables and to use one value table per type. We shall return to this point in Section 6.

As we have already mentioned, our transformations are specified as Prolog relations, and it may seem that there is a clash between Prolog’s use of substitutions and our use of more general constraints. Our implementation integrates the two by switching back and forth where necessary; moving from the constraint world to Prolog substitutions may of course cause backtracking over the possible valuations of a constraint (in practice this does not affect performance — the cost of the BDD operations dominates the time spent in the Prolog interpreter). Of course, we must also be careful to write Prolog programs that ensure that the appropriate variables are bound to ground terms before they will be needed in the constraint world.

5.2 A special case: transparency
One problem with our compositional approach is that general solutions for each atomic proposition must be computed independently for each program statement. So for example, the proposition \( \text{not}(\text{def}[X]) \) at the statement \( a := 3 \) is solved by the constraint \( X \neq a \), and we must use this solution whether or not the program variable \( a \) currently occurs elsewhere in the program, since a future transformation may introduce it.

This requirement becomes problematic when we move on to checking transparency conditions. At a particular program statement \( s \), the predicate \( \text{transparent}(E) \) should be true iff the value of the pro-
gram expression in the meta-variable $E$ cannot be affected by $s$. We take the obvious (but slightly conservative) approach of defining this to mean that none of the program variables occurring in the expression are changed by $s$, so for example $a := 3$ is transparent to $b + c$, but not to $a + c$ (ignoring aliasing effects). Thus, for the predicate transparent($E$) and the statement $a := 3$, we need to generate a constraint for $E$ that allows it to take on all possible program expressions that do not contain $a$. Clearly this is not (directly) possible in a constraints language whose atomic conditions only allow us to express equality (and hence inequality) over finite domains.

However, although the set of all possible expressions is infinite, the set of all program expressions that have so far been seen when we come to apply a transformation is finite; indeed, it can be found simply by inspecting the value table for $E$. If we can somehow delay the computation of the constraint until the point at which we actually apply a transformation, our problem is solved.

We achieve this as follows. For each pair $(E,a)$ of a meta-variable $E$ and a program variable $a$, we dynamically allocate a single Boolean variable that indicates “$a$ occurs in $E$” (we write this as $E \triangleright a$). This Boolean variable is used at constraint generation time (usually in negated form). When it comes to applying a transformation, the set of expressions that have been seen so far is inspected and the set \{$e_1, e_2, \ldots, e_k$\} of those containing $a$ is generated. We then compute the constraint

$$E \triangleright a \equiv (E = e_1 \lor E = e_2 \lor \ldots \lor E = e_k)$$

We repeat this procedure for each pair $(E,a)$, and compute the conjunction of these constraints with the constraint resulting from the path query in question. Note that the constraints decorating the syntax tree are left unchanged, so that an accurate result can still be obtained if they are used in future transformations.

It might seem that this procedure is quite expensive. In fact, we can maintain the constraint $E = e_1 \lor E = e_2 \lor \ldots \lor E = e_k$ incrementally, adding a new disjunct each time an expression containing $a$ is added to the value table for $E$, so the required constraint can be computed cheaply.

### 5.3 Local variables

If we have a block containing a local variable declaration, then analysis information that propagates outside that block should not contain mention of that variable; if it were to do so then we might apply a transformation that made a reference to the variable outside its scope, or we might find that a transformation involving a different variable of the same name was incorrectly disabled. The latter problem will particularly be an issue for recursive procedures, where we will not be able to circumvent it by renaming variables.

One option would be to analyse each block twice; once to produce analysis results that can be used inside that block, and once to produce analysis results for outside the block. However, this solution doubles the cost of our analysis, and does not sit very well with our compositional approach. It would be preferable to find a means of transforming the analysis information obtained from inside the block to remove mention of local variables. More precisely, if $C$ is an entry of the matching matrix $A_{\text{pat},B}$ for the entire block $B$, and $X$ is a meta-variable with an expression $e$ in its domain that contains an out-of-scope local variable, then we would like to transform $C$ such that all occurrences of the atomic constraint $X = e$ in $C$ are replaced by $\text{false}$.

Some thought suggests that this is unlikely to be straightforward; the BDD representation does not give us direct access to the atomic equality conditions from which a constraint was built up. Indeed, suppose our domain consists of just two variables, $p$ and $q$, where $p$ is local to a block and $q$ is not local to the block. Then the constraints $X = p$ and $X \neq q$ will be represented by exactly the same BDD; but an analysis that wished to ignore this $p$ should generate the differently represented constraints $\text{false}$ and $X \neq q$ instead.

However, this problem can be overcome by adding a special value, which we shall label $\star$, to our domain, and by then making use of a certain operation that acts on the internal structure of BDDs. The significance of this special value is that it will never be mentioned in the constraints arising from the solutions of atomic propositions at statements, which ensures that the constraints for $X = p$ and $X \neq q$ will always have different representations. In a sense, the $\star$ value acts as a placeholder for the set of all variables (or other expressions) not visible within the scope of the code a constraint applies to; if a given constraint allows $X$ to take on the value $\star$, then it should also allow $X$ to take on any variable not visible in that scope, and vice-versa.

The special operation on BDDs which we referred to is a slightly peculiar form of existential quantification. If the constraint $C$ is satisfied by a substitution $\varphi$, then $\text{ExQ}(X,C)$ is satisfied by $\varphi$ overridden with any possible assignment for $X$. In other words, all mention of $X$ has been removed; if $C$ was true for a certain value of $X$ and the other meta-variables it mentions, then $\text{ExQ}(X,C)$ is true for any value of $X$ with the same values for the other meta-variables. This operation is particularly natural because of the structure of BDDs described above.

To remove any bindings $X = p$ from the constraint $C$, we define

$$C' = \begin{cases} \text{if } X = p \text{ then } \text{ExQ}(X,C \land X = \star) \text{ else } C \end{cases}$$

To see how this works, consider a language of constraints inductively built up from equality conditions (of the form $X = p, X = q, Y = p$, etc), conjunction, disjunction and negation. If this was our representation, then the required translation could be achieved by syntactically replacing occurrences of $X = p$ with $\text{false}$. It can be shown (by induction on the structure of this language) that if $C$ is the equivalent BDD for such a constraint, then $C'$ as defined above is the equivalent BDD for the translated version of that constraint, provided that none of the equality conditions in the original constraint were of the form $X = \star$. Note that since $C'$ is equivalent to the translated constraint, it also satisfies the requirement of being equivalent to a constraint that does not have $X = \star$ as an equality condition, and thus we can safely apply the same procedure at another block boundary without requiring a different $\star$ value for that block.

Of course we do not only want to hide constraints of the form $X = p$; we need to remove all $X = e$ where $e$ contains $p$. This is achieved by replacing $X = p$ in the above definition of $C'$ by \{$X = e_1 \lor X = e_2 \lor \ldots \lor X = e_k$\} where \{$e_1, e_2, \ldots, e_k$\} is the set of expressions containing $p$; this constraint can be maintained incrementally as we described in the previous section.

Finally, we also need to remove occurrences of $E \triangleright p$; to do this we simply define

$$C' = \text{ExQ}(E \triangleright p, C\land \neg(E \triangleright p))$$

Repetition of these two procedures for each $(X,p)$ pair of meta-variable and local variable and for each $E \triangleright p$ where $p$ is a local variable now gives us the required translation.
### 5.4 Procedures

A major advantage of our compositional technique is that it can easily be extended to interprocedural analysis. In particular, we can compute the analysis matrix just once for each procedure body, and reuse it at each call site without loss of precision in our analysis. Note that this in fact yields a context-sensitive analysis, since it is equivalent to inlining the procedure body at each call site.

If a procedure $P$ has a value parameter $f$ and body $P_{body}$, then we can model a call site $P(a)$ (where $a$ is any expression) by the fragment

$$\{ \text{var } f := a; P_{body} \}$$

Similarly result parameters can be modelled by the fragment (here $a$ must of course be an lvalue):

$$\{ \text{var } P_{body}; a := f \}$$

Finally, the obvious combination can be used to model value-result parameters. Return values can be handled by adding an extra result parameter to the procedure and treating return statements inside the procedure body as assignments to that parameter; of course, a knowledge of the expression evaluation semantics of the language being transformed will be required to ensure that the analysis matrix is correctly inserted into the calling procedure.

To compute solutions for recursive procedures, we simply compute a greatest fixed point by iteration (using the matrix in which every element is the constraint $true$ as the starting value). This can be extended in the usual fashion to mutually recursive groups.

We have not yet generalised our complexity bounds to the interprocedural case.

### 5.5 Performance experiments

The key concern with regard to our efficiency analysis is that individual operations on constraints may prove to be very expensive, thus causing the time per transformation to grow out of control. In particular, if the theoretical upper limit of $O(F^V)$ ($F$ the size of the domain, $V$ the number of meta-variables) on the time per operation were to be reached in practice, then our algorithm would scale extremely badly, since in principle $F$ could be in proportion to the size of the program.

We have implemented our algorithm, currently only for intra-procedural analyses. Our implementation is in Standard ML, and makes use of BuDDy, a C implementation of BDDs (the interface between C and ML is is handled by the MuDDy package). Our object language is the verifiable subset of Microsoft’s .NET intermediate language (which is roughly equivalent to the Java bytecode language). We are working towards covering the whole of verifiable IL, but have not yet tackled its object-oriented features. Our implementation also does not yet handle alias analysis; one possibility is to integrate the BDD-based flow-insensitive analysis of Berndl et al [5]. As well as being convenient to use because of our existing use of BDDs, this analysis is incremental with respect to the addition of new “points-to” relations; if one is removed then we must either accept an inaccurate (but safe) result or restart the analysis from scratch.

We constructed a series of test cases in which the number of program variables grows linearly with the size of the program (in order to ensure that $F$ really is in proportion to the size of the program), and used these to measure the time taken for each transformation as the program size grows. We also constructed a series of test cases in which the number of program variables remains constant as the program size grows.

Since we do not yet handle objects or alias analysis, our test cases were constructed by gluing together short fragments of imperative code compiled from C# we wrote ourselves, rather than from “real-world” benchmarks.

The transformations we applied were atomic propagation, dead code elimination and unique uses propagation (a transformation that propagates $V := E$ where $E$ is any expression to a use site of $V$ so long as this is safe and there is precisely one such use site; the path query for this transformation requires verifying transparency with respect to $E$ in the same way as for common subexpression elimination).

Our experiments were performed using the MLton optimising compiler (we ported MuDDy to it for this purpose) on a 1.8GHz AMD Athlon machine with 2GB of RAM. In all cases the program was just a single method, since we have not yet implemented the interprocedural aspects of our analysis.

We observed that the time taken per transformation applied grew approximately linearly when the domain grew with the program, but remained approximately constant as the program grew if the domain did not grow — see Figures 6 and 7. Recall that the asymptotic analysis shows that there should be a logarithmic factor due to the need to recompute attributes on the path to the root; however in practice much of this computation is likely to be repeated work, and caching of the results of BDD operations inside BuDDy may therefore mask this factor. (In fact, the domain of certain meta-variables,
such as that of node identifiers, did grow with the size of the program, but the way in which these meta-variables are used meant that this had no impact on the cost of transformations.)

The linear growth in relation to $F$ is better than the potential worst-case behaviour of $O(F^2)$ (for a variable), but still slightly disappointing, as we had hoped that the sharing in the BDD representation would result in something closer to logarithmic behaviour. In contrast, some experiments with an implementation of a non-incremental solver for our queries suggest that the cost of the BDD operations in this case grows much more slowly.

However, in practice we expect that domains will grow only slowly with program size; in particular, it is not generally the case that new local variables are introduced every few lines.

It has been observed in other applications that use BDDs that their performance is very sensitive to the variable ordering chosen; however, we have not observed such sensitivity for our analysis.

6 Related work

The relation of our work to that of Conway was discussed in Section 3. To our knowledge, the only other applications of Conway’s chip-chop matrices are those of Backhouse [2, 4]; in particular [2] describes an algorithm for computing these matrices. Apart from this theoretical background, we have also taken inspiration from many previous works on the specification of program transformations, and the most relevant of these sources are further described below.

6.1 Gospel and Genesis

Gospel was put forward by Whitfield and Soffa as a notation for the formal study of program transformations [38]. Furthermore, they also provided a tool called Genesis for implementing specifications that are expressed in Gospel.

A specification in Gospel consists of three parts: a declaration section where the meta-variables are declared; a precondition section that typically describes a code pattern, as well as a number of flow dependencies; and finally an action, where the transformation is applied to the program, provided the preconditions have been satisfied. A similar style of specification is also found in the APTS system of Paige [25].

In comparison to Gospel, our specifications do not declare the meta-variables and their domains. In retrospect, this was a mistake, and we plan to rectify it: knowing the precise domain of a variable (statement, expression, constant, ...) not only enlightens the specification, it also allows for a more efficient implementation of constraints via BDDs. We express the flow dependencies of Gospel via regular path queries. This is more formal, and it allows for more rigorous analysis of the transformations, for instance when proving their correctness [19]. Of course the use of regular patterns is also an essential prerequisite for the incremental implementation technique that we have presented here.

A final difference is the fact that we use Prolog as a meta-language, allowing the user to define new primitives of her own with a minimum of effort. Any non-syntactic processing, such as comparison of integers (for example for loop unrolling) or constant folding is done within the Prolog program. A number of other researchers have observed that logic programming is suitable for this kind of application, e.g. [27].

6.2 Metaframe

Our first attempt to put the use of Gospel on a more formal basis was inspired by the work of Rus and Van Wyk on parallelizing optimisations [29]. They use modal logic as the specification language, an idea pioneered by Steffen and his coworkers [32]. The attraction of using modal logic lies both in its expressivity and in the fact that applicability can be verified via a model checker.

In first instance, it might appear that the use of a model checker is prohibitively expensive. In fact it is possible to partially evaluate the model checker with respect to specifications, and hence generate bit-vector analyses. This is implemented in the Metaframe system. Steffen reports that the resulting analyses are extremely efficient, and in fact competitive with those written by hand. Steffen’s specifications do not contain free meta-variables, however, which simplifies the implementation considerably.

A major difference of our approach with Metaframe is the incremental solving algorithm. Sokolsky and Smolka have proposed an incremental model checker for the modal mu-calculus [31], and it would be interesting to see whether this can be used in the context of program analysis and transformation. Their algorithm is quite similar to traditional incremental dataflow algorithms that employ the technique of restarting iteration [26]. At present there seem to be no model checkers that employ the other dominant paradigm in incremental dataflow analysis, namely incremental elimination methods [8, 30]. For now it appears to us that the incremental algorithm we have presented here bears little relation to any previous incremental dataflow analyses.

6.3 Cobalt

Cobalt [21] is also a language for specifying program transformations, in fact inspired by earlier work in our research team to employ modal logic as a specification language [18, 19]. The purpose of Cobalt is to enable automatic proofs that the transformations are sound, and that work thus complements the present paper on efficient execution.

To enable such automatic proofs of transformations, specifications are restricted to the form

$$\text{fromEntry}([\cdot]; [P]; [Q]; [R]) \quad \text{or the dual}\quad \text{toInt}([P]; [Q]; [R]; [\cdot]).$$

(The syntax of Cobalt is of course different.) Quite a wide class of transformations can be expressed this way, including complex examples such as partial redundancy elimination. An example of a transformation that is not readily described in this format is strength reduction. Furthermore, it is difficult to filter out the infeasible paths as in our formulation of conditional constant propagation. This is admittedly less of a problem in the presence of the composition mechanisms of [20].

These composition mechanisms, which combine multiple transformations by speculatively applying them to loop bodies and then iterating until a fixed point is reached, allow transformations which are mutually dependent with the results of other transformations to be applied. For example, Wegman and Zadeck’s conditional constant propagation [37] can be obtained by using ordinary constant propagation and dead branch elimination as building blocks. Sim-
ilarly, it would be possible to eliminate a chain of mutually dependent assignments such as \( x := y; y := z; z := x \) inside a loop using a standard formulation of dead assignment elimination.

We expect that these compositions mechanisms can be easily applied within our framework (in [21] the verification that this is possible for Cobalt is also left to future work). Indeed, our strategy language should make the specification very natural, in a similar manner to current work in Stratego [24].

Of course, speculatively applying transformations and then iterating is likely to be relatively expensive. A cheaper, but less general, option might be to write patterns tailored for specific cases — for example, the elimination of precisely three mutually dependent assignments. We hope that it would be possible to generate such patterns automatically from the basic patterns for the transformations in question.

6.4 Optimix

An alternative to a transformation system based on term rewriting is one that employs the formal notion of graph rewriting. This is the basis of the Optimix system, designed by Åßmann [1]. It supports both edge addition rewrite systems (EARS, which have nice formal properties, in particular unique normal forms), and more general exhaustive graph rewrite systems (XRGs, which can be checked for termination). Typically the EARS are used to perform analysis, augmenting the original flow graph representing the program, and then the XRGs comes into play for actually modifying the flow graph. This is a significant improvement over Gospel in the sense that the preconditions and the modification part of the specification are both put on a solid semantic basis.

To encode dataflow analyses in Optimix, they are expressed as reachability problems. Many analyses can be so expressed, as first shown in a ground-breaking paper by Reps, Horwitz and Sagiv [28]. Efficient implementations are generated by the application of Dialog techniques. The recent work of Liu and Stoller [22] on generating efficient analyses from Datalog specifications is very closely related to that of Åßmann. We have discussed their algorithms for solving regular path queries earlier in this paper. This lays bare the connection to our work, as essentially we are also computing the solution to a reachability problem, applied to the product of the flow graph and the deterministic automaton that represents the regular path query. The ESP system [10] extends the reachability algorithm of [28] in a similar way, to solve regular path queries that encode safety properties of software. A less principled but similar approach is taken by the designers of Metal, which was designed for the dual purpose of specifying optimisations and bug identification [15]. The close similarity between our queries and those in ESP and Metal suggests that the incremental algorithm presented here might be applied to check safety properties on the fly in an interactive development environment.

In [27], Reps shows how demand-driven versions of certain interprocedural program analyses (expressed as reachability problems) can be obtained via the magic-sets transformation. He also notes that the same result can be obtained by directly executing the reachability definitions via tabled SLD resolution — this was the technique we employed in [11], and at least with that prototype implementation, the performance was not adequate in practice. Previous work of Duesterwald et al. [13] has indicated that demand-driven analysis often outperforms incremental implementations, but it appears that for the problem considered here, the situation is reversed.

7 Acknowledgements

We would like to thank the Programming Tools Group at Oxford for many enjoyable discussions about this work, especially David Lacey who suggested many improvements. Roland Backhouse, Jens Knoop, David Lacey, Sorin Lerner, Annie Liu, and Eric van Wyk provided valuable feedback on a draft of this paper, as did the anonymous POPL referees. Stephen Drape did some initial experiments with the implementation. Some of the early parts of this work were funded by a generous gift from Microsoft Research.

8 References


