Imperative Program Transformation by Rewriting

David Lacey and Oege de Moor

Oxford University Computing Laboratory

Abstract. We present a method of specifying standard imperative program optimisations as a rewrite system. To achieve this we have extended the idea of matching sub-terms in expressions with simple patterns to matching blocks in a control flow graph. In order to express the complex restrictions on the applicability of these rewrites we add temporal logic side conditions. The combination of these features allows a flexible, high level, yet executable specification of many of the transformations found in optimising compilers.

1 Introduction

Traditional rewrite systems are good for expressing transformations because:

- Rewrites tend to be very succinct in their specification.
- The represented transformation is intuitively expressed. Rewrite patterns express the required partial structure of transformable terms in an explicit way.
- There are existing approaches to combining rewrite sets via strategies and reasoning about their properties such as confluence, soundness and termination.

A lot of the success of expressing program transformations by rewriting has been for purely functional programs. However, rewriting imperative programs is problematic. One of the main reasons for this is that the conditions under which it is possible to apply a rewrite are hard to describe. Many rewrite systems work by transforming abstract syntax trees and to decide applicability one needs complex definitions to analyse these trees. The conditions are easier to specify if the program is represented by its control flow graph but it is not obvious how to specify rewrites on graphs.

This paper tackles these problems in two ways. Firstly, a pattern matching language is developed that detects patterns in programs represented by their control flow graph. Secondly, we use a language for specifying applicability conditions on rewrites using temporal logic constructs for reasoning over the graph. This leads to a language which is capable of expressing many common transformations on imperative programs found in optimising compilers. To give an impression of our approach, here is the specification (which will be described in detail during the paper) of the optimising transformation *constant propagation*.

R. Wilhelm (Ed.): CC 2001, LNCS 2027, pp. 52–68, 2001.
© Springer-Verlag Berlin Heidelberg 2001

It says that an assignment $x := v$ (where v is a variable) can be replaced by $x := c$ if the "last assignment" to v was $v := c$ (where c is a constant). The side condition formalises the notion of "last assignment", and will be explain later in the paper:

$$\begin{array}{l} \quad n : (x := v) \implies x := c \\ \text{if} \\ \quad n \vdash A^{\triangle}(\neg def(v)\ U\ def(v) \wedge stmt(v := c)) \\ \quad conlit(c) \end{array}$$

The rewrite language has several important properties:

- The specification is in the form of a rewrite system with the advantages of succinctness and intuitiveness mentioned above.
- The rewrite system works over a control flow graph representation of the program. It does this by identifying and manipulating *graph blocks* which are based on the idea of basic blocks but with finer granularity.
- The rewrites are executable. An implementation exists to automatically determine when the rewrite applies and to perform the transformation just from the specification.
- The relation between the conditions on the control flow graph and the operational semantics of the program seems to lend itself to formal reasoning about the transformation.

The paper is organised as follows. §2 covers earlier work in the area and provides the motivation for this work. §3 describes our method of rewriting over control graphs. §4 describes the form of side conditions for those rewrites. §5 gives three examples of common transformations and their application when given as rewrites. §6 discusses what has been achieved and possible applications of this work.

2 Background

Implementing optimising transformations is hard: building a good optimising compiler is a major effort. If a programmer wishes to adapt a compiler to a particular task, for example to improve the optimisation of certain library calls, intricate knowledge of the compiler internals is necessary. This contrasts with the description of such optimisations in textbooks [1,3,26], where they are often described in a few lines of informal English. It is not surprising, therefore, that the program transformation community has sought declarative ways of programming transformations, to enable experimentation without excessive implementation effort. The idea to describe program transformations by rewriting is almost as old as the subject itself. One early implementation can be found in the TAMPR system by Boyle, which has been under development since the early '70s [8,9]. TAMPR starts with a specification, which is translated to pure lambda calculus, and rewriting is performed on the pure 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 [24]. A yet more modern system in the same tradition is Stratego, built by Visser [32]. Stratego has sophisticated mechanisms for building transformers from a set of labeled, unconditional rewrite rules. Again the published applications of Stratego are mostly restricted to the transformation of functional programs. TrafoLa is another system able to specify sophisticated syntactics program patterns [20].

It would be wrong, however, to suggest that rewriting cannot be applied in an imperative setting. For instance, the APTS system of Paige [27] 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. This is the norm in similar systems that have been constructed in the high-performance computing community, such as MT1 [7], which is a tool for restructuring Fortran programs. Other transformation systems that suffer the same drawback include Khepera [15] and Txl [11].

It is well known from the compiler literature that the program analyses necessary in the side conditions of transformations are best expressed in terms of the control flow graph of an imperative program. This has led a number of researchers, in particular Assman [4] and Whitfield and Soffa [33] to investigate graph transformation as a basis for optimising imperative programs. Whitfield and Soffa's system, which is called Genesis, allows the specification of transformations in a language named Gospel. Gospel has very neat declarative specifications of side conditions (referring to the flow graph), but modifications of the program are carried out in a procedural manner. Assmann's work, by contrast, is much more declarative in nature since it relies on a quite general notion of graph rewriting. One advantage of this approach is that certain conditions on the context can be encoded as syntactic patterns. It is impossible, however, to make assertions about program paths. This restriction is shared by Datalog-like systems expressing program analyses as logic programs [12].

A number of researchers have almost exclusively concentrated on elegant ways of expressing the side conditions of transformations, without specifying how the accompanying transformations are carried out. For example, Sharlit [30] is a tool for generating efficient data flow analyses in C. It is much more expressive than the limited notation of the tools discussed above, but the user has to supply flow functions in C, so the specifications are far from declarative. However, several systems (for example BANE [2] and PAG [25]) provide a more elegant system of automatically generating dataflow information from recursive sets of dataflow equations.

The work of Bernhard Steffen provides a new approach in the field. In a pioneering paper [28], Steffen showed how data flow analyses could be specified through formulae in temporal logic. Steffen's descriptions are extremely concise and intuitive. In a series of later papers, Steffen and his coworkers have further articulated the theory, and demonstrated its practical benefits [22,23,29].

The present paper builds on Steffen's ideas, combining it with a simple notion of rewriting on control flow graphs, and introducing a kind of logical variable so that the applicability conditions can be used to instantiate variables on the right-hand side of a rewrite rule.

3 Rewriting Flow Graphs

3.1 Rewrite Systems

Rewrites specify a tranformation between two objects. Usually these objects are tree-like expressions but they may also be more general graph structures. They consist of a left hand side pattern, a right hand side pattern and (optionally) a condition. We shall express this in the following manner:

$$LHS \Longrightarrow RHS \quad \text{if} \quad Condition$$

A pattern expresses a partial structure of an object. It will contain free variables, denoting parts of the structure that are unknown. For each pattern there will be a set of objects that have the same partial structure as the pattern. Objects of this set *match* the pattern. Any rewrite system needs a matching algorithm that determines whether an object matches a pattern. When a pattern matches then the free variables in the pattern will correspond to known parts of the matching object. So we can talk about the *matching substitution* that maps the free variables to these known parts.

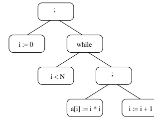
The rewrite is implemented by finding a sub-object of the object we are transforming which matches the left hand side pattern with matching substitution θ. The matching sub-object is replaced by the right hand side, where variables are instantiated by θ.

3.2 Representations of Programs

The imperative programs we wish to transform consist of program statements linked together by various control structures. To transform these programs we have a choice of representations. The program statements themselves are naturally expressed as syntax trees but this is not necessarily the best way to represent the control structure of a program. Figure 1 gives an example of a simple program and two possible representations.

Expression Trees. Expression trees are the most common objects used for rewriting. The advantage is that sub-term replacement is a simple operation. However, reasoning about dataflow properties is difficult in this representation.

```
i := 0;
while (i ¡ N ) do
 a[i] := i*i ;
 i := i + 1
end while
```



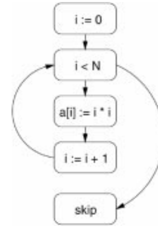


Fig. 1. A simple program and two representations

Control Flow Graphs. The control flow graph treats statements as nodes and edges as possible control flow of the program. If the edges are labelled for conditional jumps it can characterise the program. This allows side conditions to be expressed more simply but patterns over graphs are more complicated. For many structured transformations these pattern matching problems can be overcome however, so this is the representation we chose to work with.

Often the control flow graph is used as an intermediate representation in compilers on which optimising transformations are performed. This is another reason for choosing the CFG as our representation.

The representation chosen here is over a fairly simple and naive language. In particular there is no notion of pointers or aliasing.

3.3 Graph Blocks and Graph Rewriting

We concentrate on the notion of single-entry-single-exit regions as our objects of rewriting. For convenience we will refer to these regions as *graph blocks*. The term *hammocks* is sometimes used for this, but the use of this terminology is inconsistent, as pointed out in [21]. Often analysis in compilers is in terms of *basic blocks*, these are also graph blocks but tend to be specified as not containing cycles and maximal with respect to this property. We do not put this restriction on graph blocks since we need a finer granularity to specify transformations.

Any single node is a graph block, as is an entire program. Also, for many structured programs (not containing arbitrary goto statements), any sequence of statements also corresponds to a block in the control flow graph.

The restriction of objects for rewriting from general graphs to graph blocks allows us to easily specify a language for expressing patterns over graph blocks. The most elementary pattern is just a free variable representing any graph block and we will represent these variables by lower case Roman letters.

Any single node is a graph block, so one would like a way of specifying a single node. The important property we wish to match for a node is its associated program statement. Since this is just a term we can create patterns for these in the usual way. We can then specify in block patterns that a statement pattern matches any graph block consisting of a single node that matches the statement

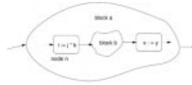


Fig. 2. A graph block pattern

pattern[1]. Sometimes, it is useful to have a free variable which gives a name to such a block, in which case the statement pattern is prefixed by a free variable and a colon. Here are a couple of examples:

$$x := i$$
$$n : (y := z * x)$$

The simplest way in which blocks are related is by sequencing together. This is expressed by the ; operator in the pattern language. So a pattern $a; b$ will match the union of a block which matches a and a block which matches b, providing the exit of a has a sequential successor that is the entry to b.

The final operator we used in our matching language is that of a *context*. This is used to express the idea of a block that contains another block. So the pattern $a[b]$ will match a block that matches a, and futhermore contains a sub-block matching b.

Overall, the grammar of the matching language is:

```
<block pattern> ::= <var>
                  | <var>:<statement pattern>
                  | <var>[<block pattern>]
                  | <block pattern> ; <block pattern>
```

To illustrate, Figure 2 pictorially shows the pattern:

$$a[n : (i := j * k); b; x := y]$$

During rewriting a pattern may match with an empty subgraph. In this case we see this as being equivalent to matching a single node with performs no operation. This is required for construction of the new graph created by the rewrite.

4 Side Conditions

The applicability conditions of a rewrite are expressed as side conditions which place restrictions on what objects can be matched to the free variables in the left hand side pattern of the rewrite. The language of conditions presented here is not claimed in any way to be complete, but it suffices to specify many of the standard transformations found in optimising compilers.

[1] There is an ambiguity here as a statement pattern could also be just a single free variable, but we stipulate that this is a block pattern and not a statement pattern.

These restrictions are specified by propositions containing the free variables found in the rewrite which must hold of the objects matching those variables. There are simple restrictions one may make about expressions used by statements such as whether they consist of just a constant or variable literal. These primitive conditions are listed in Figure 3. For example, one may specify the following rewrite which performs a limited form of constant propagation:

$$x := c; y := x \Longrightarrow x := c; y := c \quad \text{if} \quad conlit(c)$$

These basic conditions can be combined with standard propositional logic operators ($\wedge, \vee, \neg$) to form more complex propositions. For example, here is the same limited constant propagation rewrite that only propagates something that is a constant literal and has type $ShortInt$:

$$x := c; y := x \Longrightarrow x := c; y := c \quad \text{if} \quad conlit(c) \wedge type(c, ShortInt)$$

$True$	Always holds
$False$	Never holds
$conlit(x)$	Holds if x is a constant literal
$varlit(x)$	Holds if x is a variable literal
$type(x, t)$	Holds if x is of type t

Fig. 3. Basic conditions

These conditions allow us to specify restrictions on statements in the control flow graph but not on how the nodes in the graph relate to each other. The paper by Steffen [29] shows that temporal logic is a very succinct way of specifying dataflow properties. This is the approach we take here.

The restriction on nodes in the control flow graph are expressed as a sequent. This is a formula of the form:

$$n \vdash TempForm$$

Where $TempForm$ is a temporal formula whose syntax is described below. A sequent can be read as saying that a formula "holds at" or "is satisfied by" a particular node.

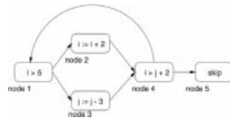


Fig. 4. A sample control flow graph

$def(x)$	The variable x is defined at this node.
$use(x)$	The variable x is used at this node.
$node(n)$	This node is n
$stmt(p)$	The statement associated with this node matches pattern p

Fig. 5. Basic temporal conditions

The most basic things we want to know about a node are its identity and the associated program statement. The syntax for the latter is:

$$n \vdash stmt(StmtPattern)$$

So the following formula would hold in Figure 4 with the substitution $\{n \mapsto node\ 2, x \mapsto \mathtt{i}, a \mapsto \mathtt{i}, b \mapsto 2\}$:

$$n \vdash stmt(x := a + b)$$

A couple of useful predicates derived from *stmt* are $def(x)$ and $use(x)$, these are described in Figure 5.

Identity is specified using the temporal predicate *node*. For example:

$$n \vdash node(m)$$

This formula holds for any substitution that maps n and m to the same node. This facility is useful when combined with the other temporal constructors.

We need ways to relate nodes to each other. This is done via the four temporal constructors described in Figure 6. The first and third, EX and AX specify the relation between a node and its immediate successors. For example, the formula below states that node n has a successor that is node m, thus restricting n and m to be related as such in the control flow graph:

$$n \vdash EX(node(m))$$

In Figure 4, a substitution satisfying this formula is $\{m \mapsto node\ 4, n \mapsto node\ 3\}$.

We may not be interested in only successors to a node but predecessors also. Putting a $^{\triangle}$ next to a constructor specifies predecessors instead of successors. For example, the same relation as above can be expressed as:

$$m \vdash EX^{\triangle}(node(n))$$

These constructors express relations between *immediate* successors or predecessors. However, many relations are between nodes which have paths of multiple edges between them. To express these relations we appeal to the *until* operators of computational tree logic $(A(\ldots U \ldots), E(\ldots U \ldots))$ [10]. These are predicates on paths in the control flow graph. A path is a (possibly infinite) sequence of

$EX(f_1)$	There exists a successor of this node such that f_1 is satisfied at that successor.
$E(f_1 U f_2)$	There exists a path from this node to a node n such that every node on that path up to but not including n satisfies f_1 and n satisfies f_2.
$AX(f_1)$	All successors of this node satisfy f_1
$A(f_1 U f_2)$	Every path from this node is either infinite with all the nodes on the path satisfying f_1, or finite such that f_1 is satisfied on every node until a node that satisfies f_2.

Fig. 6. Temporal constructors

nodes $< n_1, n_2, \ldots >$ such that each consecutive pair (n_i, n_{i+1}) is an edge in the control flow graph. The formula $n \vdash E(f_1 U f_2)$ holds if a path exists, starting at n such that f_1 is true on this path until f_2 is true on the final node. That is, we have a finite non-empty path $< n_1, n_2, \ldots, n_N >$ such that $n_1 = n$, for all $1 \leq i \leq N-1$: $n_i \vdash f_1$ and $n_N \vdash f_2$. For example the following formula holds if there is a path from node n to node m such that every node on that path does not define x[2] :

$$n \vdash E(\neg def(x)\ U\ node(m))$$

For example, this formula would match the graph in Figure 4 with substitution $\{n \mapsto node\ 3, x \mapsto \mathtt{i}, m \mapsto node\ 1\}$[3].

The universal until operator: $n \vdash A(f_1\ U\ f_2)$, says that every path starting at n is either infinite with every node satisfying f_1 or finite with a prefix that satisfies f_1 until a point where f_2 is satisfied. Note that here the logic deviates slightly for standard CTL in that in the universal case we use the weak version of the until operator.

As with the EX/AX constructors we can look at paths that follow predecessor links instead of successor links. So $E^{\triangle}(\ldots U \ldots)$ and $A^{\triangle}(\ldots U \ldots)$ look at paths running backwards through the control flow graph.

The temporal operators find paths over the entire control flow graph. Sometimes it is necessary to restrict their scope. For example, the following formula says that all paths, *whose nodes all lie within the graph block* b, satisfy $True$ until $def(x)$:

$$A[b](True\ U\ def(x))$$

This would hold in Figure 4 for substitution $\{a \mapsto nodes\ 1\ to\ 4, x \mapsto \mathtt{i}\}$.

Sequents of the form $n \vdash TempForm$ are good for reasoning about nodes in the graphs. However, the block patterns described in section 3 can result in free

[2] Note here that the U operator has weakest precedence amongst the logical operators.
[3] This is not the only substitution that will provide a match for that formula on that graph.

variables standing for graph blocks. We extend the idea of sequents to deal with graph blocks in the following four ways:

$$\begin{array}{ll} all(a) & \vdash TempForm \\ exists(a) & \vdash TempForm \\ entry(a) & \vdash TempForm \\ exit(a) & \vdash TempForm \end{array}$$

Respectively, these correspond to the statements "every node in the block satisfies ...", "there exists a node in the block satisfying ...", "the entry of the block satisfies ..." and "the exit of the block satisfies ...".

Fresh Variables. In many rewrite systems the only free variables that can occur in a rewrite are on the left hand side of the rewrite. However, the free variables in a condition may be so constrained that they can be used to construct the right hand side without appearing in the left hand side. For example, the following rewrite has a side condition that all predecessors are of the form $v := c$. This will restrict the value of c so it can be used in the right hand side:

$$x := v \implies x := c \quad \text{if} \quad AX^{\triangle}(stmt(v := c))$$

Our use of free variables in logical predicates (where there may be several possible satisfying substitutions) is similar to logic programming. It is a very important extension to the rewrite language, allowing conditions to interact more directly with the rewrite.

In line with the similarity to logic programming it is useful to have some evaluating predicates in the side conditions. In particular, the condition below states that x must equal the value of y multiplied by z:

$$x \texttt{ is } y \times z$$

This predicate is only meaningful when x,y and z match constant numeric literals.

5 Examples

5.1 Constant Propagation

Constant propagation is a transformation where the use of a variable can be replaced with the use of a constant known before the program is run (i.e. at compile time).

The standard method of finding out if the use of a variable is equivalent to the use of 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.

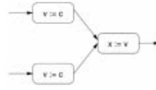


Fig. 7. Possibly transformable code snippet

The rewrite itself is simple[4]:

$$n : (x := v) \implies x := c$$

Evidently we need restrictions on v and c. The idea is that if we follow all paths of computation backwards from node n, then the first definition of v we come to must be of the form $v := c$. The paths of computation backwards from node n are just the backwards paths from n in the control flow graph. We want to check all of them so the $A^{\triangle}(..U..)$ constructor is appropriate. To fit into the "until" path structure we can note that requiring the first definition on a path to fulfill 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. This gives us the condition:

$$n : (x := v) \Longrightarrow x := c \text{ if } n \vdash A^{\triangle}(\neg def(v)\ U\ def(v) \wedge stmt(v := c)) \\ conlit(c)$$

The language we use to specify rewrites is quite powerful and, unsurprisingly, one can specify transforms in different ways which have subtle variations in behaviour. Another way of looking at constant propagations is that it rewrites a program consisting of a constant assignment to a variable which is itself used after some intermediate block of code. This leads us to form a rewrite thus:

$$v := c; a; x := v \Longrightarrow v := c; a; x := c$$

Here the condition needed is that v is not redefined in block a. This is simple to express as:

$$v := c; a; x := v \Longrightarrow v := c; a; x := c \text{ if } all(a) \vdash \neg def(v) \\ conlit(c)$$

However, this formulation will not transform the program fragment shown in Figure 7 (in that the graph does not match the LHS of the rewrite) whereas the former formulation would. This shows that different specifications can vary in quite subtle ways and reasoning about equivalence probably requires a more formal treatment.

---

[4] This rewrite tackles one kind of use of the variable v. Other similar rewrites can be formed for other uses (e.g. as operands in a complex assignment)

5.2 Dead Code Elimination

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

$$n : (x := e) \implies skip$$

The condition on this rewrite is that all future paths of computation do not use x. Another way of looking at this property is to say that there does not exist a path that can find a node that uses x. This can be specified using the $E(\ldots U \ldots)$ construct. However, care is required, since we do not care if x is used at node n. So we can specify that for all successors of n, there is no path of computation that uses x:

$$n : (x := e) \implies skip \text{ if } n \vdash AX(\neg E(True\ U\ use(x)))$$

Note that for this rule and the one above, while providing illustrative examples, may not perform all the constant propagations or dead code elimination one may want. In particular, if a variable is not truly dead but *faint* [16]. These optimisations would require more complicated rules combined with other transformations such as copy propagation.

5.3 Strength Reduction

Strength reduction is a transformation that replaces multiplications within a loop structure into additions that compute the same value. This will be beneficial if the computational cost of multiplication is greater than that of addition.

For example, the following code:

```
i := 0;
while (i < N) do
   j := i * 3;
   a[i] := j;
   i := i + 1;
end while
```

Could be transformed to:

```
i := 0;
j := 0;
while (i < N) do
   a[i] := j;
   i := i + 1;
   j := j + 3;
end while
```

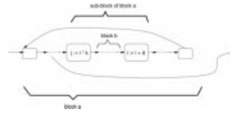


Fig. 8. Pattern for loop strengthening

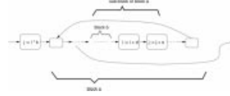


Fig. 9. After the strengthening transformation

The pattern here is that a variable `i` is incremented by a constant each time in the loop and the variable `j` is calculated by multiplying `i` by a constant value. The pattern as it would occur in a control flow graph is shown in Figure 8. It can be captured using our graph matching language by:

$$a[n : (j := i * k); b; m : (i := i + d)]$$

The transformed code will be of the form given in Figure 9. This can be specified as:

$$j := i * k; a[b; i := i + d; j := j + step]$$

Obviously appropriate conditions are needed that restrict the value of *step*. This can be calculated as such:

$$step \texttt{ is } k \times d$$

For the rewrite to be valid we stipulate both k and d to be constant literals. Therefore we add the condition:

$$conlit(k) \wedge conlit(d)$$

The transformation depends on the fact that the node m and node n are the only points in the loop that define i or j respectively. This can be stipulated by saying that for every node in graph block a either we do not define i, or we are at node m and either we do not define j, or we are at node n:

$$all(a) \vdash \neg def(i) \vee node(m) \quad \wedge \quad all(a) \vdash \neg def(j) \vee node(n)$$

Finally, this rewrite can only produce a benefit if the block a is indeed a loop. To specify this we can use the observation that a is a loop if the entry to a has a predecessor in a. That property is conveniently phrased using a temporal operator that has reduced scope. If the entry to a has a predecessor within a then this predecessor must also be a descendant of the entry of a:

$$entry(a) \vdash EX^{\triangle}[a](True)$$

The condition is equivalent to saying that the entry to the block a is a loop header which has a back edge connected to it. Putting everything together, strength reduction is captured by:

$$\begin{array}{ll} & a[n : (j := i * k); b; m : (i := i + d)] \\ \Longrightarrow & \\ & j := i * k; a[b; i := i + d; j := j + step] \\ \text{if} & \\ & step \texttt{ is } k \times d \\ & conlit(k) \\ & conlit(d) \\ & all(a) \vdash \neg def(i) \vee node(m) \\ & all(a) \vdash \neg def(j) \vee node(n) \\ & entry(a) \vdash EX^{\triangle}[a](True) \end{array}$$

6 Discussion and Future Work

6.1 Summary

This paper has demonstrated with a few examples a method of specifying imperative program transformations as rewrites. It is the authors' belief that these specifications are clear and intuitive. The approach combines techniques from program optimisation, rewriting, logic programming and model checking.

6.2 Implementation

The rewrite language presented has been specifically designed to express executable specifications. A small prototype system has already been implemented covering some of the specification language to automatically execute rewrites on simple programs.

The implementation is built around a constraint solving system. A left hand pattern can be seen as a constraint on the free variables in the pattern. The side conditions can be viewed in the same way. The rewrite engine translates the left hand side pattern and the side conditions into one constraint set with is then resolved into normal form. The normal form for this constraint set provides a list of matching substitutions that obey the restrictions in the side conditions. An interesting aspect of the implementation is the path finding algorithm which implements model checking fixed point algorithms [10] raised to the level of constraints.

Currently, the implementation is not as efficient as if one had hand-coded the transformations. However, this is to be expected due to the general nature of our approach. Initial experimentation shows that the performance does not terminally degrade for non-trivial (∼400 lines) sized programs. We believe that a workable efficiency could be achieved with more efficient constraint representations and the addition of code that caches and incrementally updates information

obtained while checking/applying different rewrites to the code. In addition it may be useful in future to extend to language to let the user specify which common side conditions could be factored out of the rewrites to be calculated together. Greater detail of the implementation should appear in a future paper.

6.3 Future Work

Semantics. The semantics of the rewrites are well defined in terms of the control flow graph. The control flow graph has a well defined relation to the semantics of the program. By formalising these relations we hope to develop general methods for establishing certain properties of the transformations we express, such as soundness or when the transformation is performance improving.

Annotated/Active Libraries. The example transformations presented in this paper are general ones to be applied to any program. These are well known and implemented in many compilers. However, code that is specialised to some specific purpose or particular architecture can be particularly amenable to certain specialised optimisations. To implement these optimisations automatically in "traditional" compilers involves detailed knowledge of the compiler design and implementation which is not available to everyone. Active libraries [31] try and bridge the gap between the optimising compiler and the library writer. The idea is that the library is annotated with domain specific information to help the compiler perform optimisations. Engler and his team at Stanford have explored the idea in some depth, and illustrated the possibilities in on wide variety of examples [13,14].

The rewrite language we have presented seems an ideal language for expressing different optimisations as annotations to a library. We hope to experiment with different domains/architectures to see how useful it would be. An good starting point might be to compare our language of optimising annotations with that of the Broadway compiler constructed by Guyer and Lin [17,18,19]. That work shares our concern that optimisations should be specified in a simple declarative style.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1985.
2. A. Aiken, M. Fuhndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Second International Workshop on Types in Compilation (TIC '98)*, March 1998.
3. A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
4. U. Assmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In P. Fritzson, editor, *Compiler Construction 1996*, volume 1060 of *Lecture Notes in Computer Science*. Springer, 1996.

5. Uwe Aßmann. On Edge Addition Rewrite Systems and Their Relevance to Program Analysis. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *5th Int. Workshop on Graph Grammars and Their Application To Computer Science, Williamsburg*, volume 1073 of *Lecture Notes in Computer Science*, pages 321–335, Heidelberg, November 1994. Springer.
6. Uwe Aßmann. OPTIMIX, A Tool for Rewriting and Optimizing Programs. In *Graph Grammar Handbook, Vol. II.* Chapman-Hall, 1999.
7. A. J. C Bik, P. J. Brinkhaus, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Transformation mechanisms in mt1. Technical report, Leiden Institute of Advanced Computer Science, 1998.
8. J. M. Boyle. A transformational component for programming languages grammar. Technical Report ANL-7690, Argonne National Laboratory, IL, 1970.
9. J. M. Boyle. Abstract programming and program transformation. In *Software Reusability Volume 1*, pages 361–413. Addison-Wesley, 1989.
10. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1996.
11. J. R. Cordy, I. H. Carmichael, and R. Halliday. The TXL programming language, version 8. Legasys Corporation, April 1995.
12. Steven Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems — A case study. *ACM SIGPLAN Notices*, 31(5):117–126, May 1996.
13. D. R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of the First Conference on Domain-Specific Languages*, pages 103–118. USENIX, 1987.
14. D. R. Engler. Interface compilation: Steps toward compiling program interfaces as languages. *IEEE Transactions on Software Engineering*, 25(3):387–400, 1999.
15. R. E. Faith, L. S. Nyland, and J. F. Prins. KHEPERA: A system for rapid implementation of domain-specific languages. In *Proceedings USENIX Conference on Domain-Specific Languages*, pages 243–255, 1997.
16. R. Giegerich, U. Moncke, and R. Wilhelm. Invariance of approximative semantics with respect to program transformations, 1981.
17. S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Second conference on Domain-Specific Languages*, pages 39–52. USENIX, 199.
18. S. Z. Guyer and C. Lin. Broadway: A software architecture for scientific computing. Proceedings of the IFIPS Working Group 2.5 Working Conference on Software Architectures for Scientific Computing Applications. (to appear) October, 2000., 2000.
19. S. Z. Guyer and C. Lin. Optimizing high performance software libraries. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing. August, 2000.*, 2000.
20. R. Heckmann. A functional language for the specification of complex tree transformations. In *ESOP '88*, Lecture Notes in Computer Science. Springer-Verlag, 1988.
21. R. Johnson, D. Pearson, and K. Pingali. Finding regions fast: Single entry single exit and control regions in linear time, 1993.
22. M. Klein, J. Knoop, D. Koschützski, and B. Steffen. DFA & OPT-METAFrame: a toolkit for program analysis and optimization. In *Proceedings of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 418–421. Springer, 1996.

23. J. Knoop, O. Rüthing, and B. Steffen. Towards a tool kit for the automatic generation of interprocedural data flow analyses. *Journal of Programming Languages*, 4:211–246, 1996.
24. P. Lipps, U. Mönke, and R. Wilhelm. OPTRAN – a language/system for the specification of program transformations: system overview and experiences. In *Proceedings 2nd Workshop on Compiler Compilers and High Speed Compilation*, volume 371 of *Lecture Notes in Computer Science*, pages 52–65, 1988.
25. Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
26. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.
27. R. Paige. Viewing a program transformation system at work. In *Proceedings Programming Language Implementation and Logic Programming (PLILP), and Algebraic and Logic Programming (ALP)*, volume 844 of *Lecture Notes in Computer Science*, pages 5–24. Springer, 1994.
28. B. Steffen. Data flow analysis as model checking. In *Proceedings of Theoretical Aspects of Computer Science*, pages 346–364, 1991.
29. B. Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, 21:115–139, 1993.
30. S. W. K. Tjiang and J. L. Henessy. Sharlit — a tool for building optimizers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.
31. Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98).* SIAM Press, 1998.
32. E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming '98*, ACM SigPlan, pages 13–26. ACM Press, 1998.
33. D. Whitfield and M. L. Soffa. An approach for exploring code-improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, 1997.