URL: http://www.elsevier.nl/locate/entcs/volume70.html 20 pages

Strategies for Source-to-Source Constant Propagation

Karina Olmos and Eelco Visser

Institute of Information and Computing Sciences
Universiteit Utrecht, P.O. Box 80089
3508 TB Utrecht, The Netherlands.
karina@cs.uu.nl, visser@acm.org

Abstract

Data-flow optimizations are usually implemented on low-level intermediate representations. This is not appropriate for source-to-source optimizations, which reconstruct a source level program after transformation. In this paper we show how constant propagation, a well known data-flow optimization problem, can be implemented on abstract syntax trees in Stratego, a rewriting system extended with programmable rewriting strategies for the control over the application of rules and dynamic rewrite rules for the propagation of information.

1 Introduction

Optimizing compilers for imperative languages apply data-flow optimizations to improve the performance of programs [14,2]. Data-flow optimizations such as constant propagation, copy propagation, and dead code elimination transform or eliminate statements or expressions based on data flow information that is propagated along the control-flow paths of the program. For example, in constant propagation the assignment of a constant value to a variable is propagated to occurrences of the variable, which can then be replaced by the constant value.

Data-flow optimizations in compilers are usually performed on an intermediate representation (IR) in which control is expressed by means of labels and jumps, and memory access is expressed in terms of registers and memory stores and fetches [16,14,22], or in terms of stack operations. For the implementation of optimizations, the flat list of instructions is turned into a control-flow graph, which guides the propagation of information. Implementing optimizations on a low-level intermediate representation has the advantage that the implementation is reusable for all source languages that are translated to the IR.

On the other hand, the translation to IR leads to the elimination of information which was available at source level. In compilation this leads to over-

head, e.g., control-flow analysis is required to recover control-flow structures such as conditional branches and loops, which were available in the original source program. In source-to-source transformation the loss of information makes it impossible to recreate source programs close to the original.

Source-to-source transformation systems manipulate programs at source level and produce transformed programs in the same (high-level) language. Applications include instrumentation, aspect weaving, software renovation, and domain-specific optimization. For example, the CodeBoost transformation system [4] transforms C++ programs with the purpose of eliminating the overhead caused by an abstract programming style, and applying optimizations based on knowledge of the application domain, which the C++ compiler does not have.

To (re)construct a source level program after transformation, source-to-source transformations are best applied to a representation that is as close to the original source as possible — abstract syntax trees. Since transformations on control-flow graphs are not directly applicable to abstract syntax trees, source-to-source optimizers cannot reuse optimizations on control-flow graphs.

Term rewriting suggests itself as a natural paradigm for transformations on (abstract syntax) trees. While optimizations such as algebraic simplification and constant folding can indeed be expressed elegantly using rewrite rules, data-flow optimizations such as constant propagation and copy propagation cannot be expressed directly using rewriting. First of all rewrite rules are context-free, i.e., can only access local information, while in data-flow optimizations the propagation of context information is required. Furthermore, standard rewriting strategies apply rules exhaustively throughout the tree. In data-flow optimizations, the control-flow paths of the program should be followed.

In this paper we show how data-flow optimizations on abstract syntax trees can be implemented in Stratego [19], a rewriting system extended with programmable rewriting strategies [20] for the control over the application of rules and dynamic rewrite rules [18] for the propagation of information.

To illustrate the ideas we develop the specification of constant propagation for Tiger [2]—an imperative language with nested functions. In Section 2 we present the abstract syntax of Tiger expressions and define constant folding rules and a generic strategy for simplification of Tiger expressions. In Section 3 we add assignment and sequential composition to the language. Constant propagation for such straight-line programs can be defined using dynamic rules, which take care of substituting constants for the variables to which they have been assigned. The strategy ensures that the rules are applied in the right order. In Section 4 we add structured control-flow constructs such as conditionals and loops. Propagation of constant assignments forks at such control flow statements and needs to be combined at the meeting points. Appropriate operations support saving, restoring, and computing 'meet' operations on sets of dynamic rules.

The building blocks developed for the specification of constant propagation can be considered as a framework for the implementation of data-flow optimizations. We have used these building blocks to implement copy propagation, common sub-expression elimination and dead code elimination. In Section 5 we discuss this generalization, compare the approach to related work, and mention future work.

We assume that the reader is familiar with the basic concepts of term rewriting. We will explain the Stratego constructs that are used, but not in depth. For an introduction to the basic concepts of Stratego we refer the reader to [20,18].

2 Constant Folding

Constant folding is the simplification of operator expressions with known constant values as operands, i.e., reducing expressions of the form `c1 op c2`, for some operator `op` with constant arguments `c1` and `c2`. For example, the expression `3 + (6 * (5 - 2)) / 2` can be simplified at compile time to `12`, by applying the laws of arithmetic. Constant folding is the goal of constant propagation; if a variable can be replaced by its constant value, run-time computations may be replaced with compile-time computations.

Before we study constant propagation, we consider the specification of constant folding on expressions. Figure 1 presents the signature of abstract syntax trees representing Tiger expressions. Expressions consist of arithmetic and relational operations on integer values, boolean values and strings. Boolean values in Tiger are represented with integer values, `false` is represented with the integer zero and `true` with any non zero integer value.

```
module Tiger-Expressions
signature
 constructors
  Int    : IntConst -> Exp
  String : String -> Exp
  Var    : Id -> Var
  BinOp  : BinOp * Exp * Exp -> Exp
  RelOp  : RelOp * Exp * Exp -> Exp
  Call   : Var * List (Exp) -> Exp
  NilExp : Exp
module Operators
signature
 constructors
  PLUS : BinOp  MINUS : BinOp  MUL : BinOp  DIV  : BinOp
  AND  : BinOp  OR    : BinOp  EQ  : RelOp  NE   : RelOp
  LT   : RelOp  LE    : RelOp  GE  : RelOp  GT   : RelOp
```

Fig. 1. Abstract syntax tree of Tiger expressions

```
EvalBinOp: BinOp(PLUS, Int(i), Int(j)) -> Int(<add>(i,j))
EvalBinOp: BinOp(PLUS, Int(0), e) -> e
EvalBinOp: BinOp(PLUS, e, Int(0)) -> e
EvalBinOp: BinOp(MINUS,Int(i),Int(j)) -> Int(<subt>(i,j))
EvalBinOp: BinOp(MINUS, Var(x), Var(x)) -> Int(0)
EvalBinOp: BinOp(MUL, Int(i), Int(j)) -> Int(<mul>(i,j))
EvalBinOp: BinOp(MUL, Var(x), Int(1)) -> Var(x)
EvalBinOp: BinOp(MUL, Int(1), Var(x)) -> Var(x)
EvalBinOp: BinOp(MUL, Int(0), e) -> Int(0)
EvalBinOp: BinOp(MUL, e, Int(0)) -> Int(0)
EvalBinOp: BinOp(DIV, Int(i), Int(j)) -> Int(<div>(i,j))
   where <not(0)> j
EvalRelOp: RelOp(EQ, Int(a), Int(b)) -> Int(<eval-rel(eq)>(a,b))
EvalRelOp: RelOp(NE, Int(a), Int(b)) ->
   Int(<eval-rel(not(eq))>(a,b))
EvalRelOp: RelOp(GT, Int(a), Int(b)) -> Int(<eval-rel(gt)>(a,b))
EvalRelOp: RelOp(LT, Int(a), Int(b)) -> Int(<eval-rel(lt)>(a,b))
EvalRelOp: RelOp(GE, Int(a), Int(b)) -> Int(<eval-rel(geq)>(a,b))
EvalRelOp: RelOp(LE, Int(a), Int(b)) -> Int(<eval-rel(leq)>(a,b))
```

Fig. 2. Constant folding rules for Tiger expressions

2.1 Constant Folding Rules

The simplification of operators with constant operands can be expressed directly using rewrite rules. Figure 2 presents several constant folding rules for arithmetic and Boolean operators. The rules are labeled such that they can be used in a strategy.

As part of the construction of a new term on the right-hand side of a rule, a strategy `s` can be applied to a subterm `t` using the notation `<s>t`. In the constant folding rules this is used to compute integer operations. For example, `<add>(i,j)` denotes a call to the primitive `add` for addition of the integer values `i` and `j`. The strategy `eval-rel(s)` applies the parameter strategy `s` and turns success into the integer value `1` and failure into `0`.

2.2 Constant Folding Strategy

In Stratego rewrite rules are not applied automatically. Instead, the specification should determine which rules are applied where and in what order by defining a rewriting strategy. For the application of the constant folding rules a single, bottom-up pass over the expression tree suffices to reduce all possible constant operator applications. This can be specified with the generic term traversal `bottomup`, as follows:

```
constant-folding = bottomup(try(fold))
```

The `fold` strategy used in the definition of `constant-folding` is defined as the choice between the `EvalBinOp` and `EvalRelOp` rules from Figure 2:

```
module Tiger-Statements
signature
  constructors
    Assign : LValue * Exp -> Exp
    Let    : List (Dec) * List (Exp) -> Exp
    VarDec : Id * Option (TypeId) * Exp -> Dec
    Seq    : List (Exp) -> Exp
```

Fig. 3. Reduced abstract syntax of Tiger.

```
fold = EvalBinOp + EvalRelOp
```

The operator `s1 + s2` denotes the choice between the strategies `s1` and `s2`. Finally, the generic traversal strategy `bottomup` traverses a tree by first applying itself recursively to all direct subterms of a node using `all` and then applying the parameter strategy `s`:

```
bottomup(s) = all(bottomup(s)); s
```

Note that the `bottomup(s)` strategy is a *one pass* traversal.

3 Constant propagation

Constant propagation is defined as the process to discover values that are constant on all possible executions of a program and to propagate these values through the program. This is a common data-flow problem and the results of this specification can be generalized towards different analyses.

For simplicity, we first consider a reduced version of Tiger which consists of expressions, assignment, let and sequential composition. Figure 3 defines this language. When reduced to this syntax, constant propagation comes down to propagating constant assignments through a sequence of assignments. It is important that constants are only substituted for variables in program points where the constant assignment is valid. For example, in the transformation

```
a := 3
b := a + 2
a := y
b := a + b
```

$\Rightarrow$

```
a := 3
b := 5
a := y
b := a + 5
```

the first occurrence of `a` in a right-hand side is replaced by `3`, but the second is not because of the intervening definition of `a`.

3.1 Dynamic Propagation Rules

As the example above illustrates, constant propagation works by replacing a variable with the constant value which is assigned to it. Since this information is not available at the place where a variable is used, the association should be established at the place where the variable is defined, i.e., the assignment, and be available at usage sites. Dynamic rewrite rules [18] were designed

for exactly this purpose. The following `assign-cp` strategy recognizes an assignment statement and generates a dynamic rule `PropConst` which rewrites the variable `Var(x)` to the expression `e` assigned to it:

```
assign-cp =
  ?Assign(Var(x), e);
  where(<is-value> e
        < rules(PropConst: Var(x) -> e)
        + rules(PropConst: Var(x) -> Undefined))
```

The match construct `?t`, matches the term pattern `t`, just like the left-hand side of a rewrite rule. The `s1 < s2 + s3` construct is a guarded deterministic choice operator. If the guard strategy `s1` succeeds, `s2` is applied to the result of `s1`, else `s3` is applied to the original term. In this case the `PropConst` rule is only generated when `e` is a value, i.e., a constant. In the other case the dynamic rule is *undefined*.

The `rules(L : t1 -> t2)` construct generates a new (dynamic) rewrite rule with label `L`, which rewrites a term matching `t1` to `t2`, just like an ordinary rule `L : t1 -> t2`. The difference is that variables in `t1` and `t2` that are bound in the context are inherited from that context. Thus, in the case above, the (meta)variables `x` and `e` are instantiated when the rule is generated, and not when it is applied. The `assign-cp` strategy can be applied several times for the same variable while a program is being traversed. Every time that this strategy is applied a new rule is generated overriding a previous rule for the same variable. However, rules for different variables are not overriden.

In case the expression `e` assigned to the variable `x` is not a constant, it is necessary to undefine the `PropConst` rule in order to kill a previously generated rule *for the same variable*. For instance, in the previous example the rule `PropConst: Var("a") -> Int(3)` is undefined by the assignment `Assign(Var("a"),Var("y"))`. At this point, the rule is overriden with `PropConst: Var("a") -> Undefined`. The application of an undefined rule always fails. Thus, attempting to apply this rule to the term `Var("a")` fails. Rules for different varialbes are not undefined.

3.2 Propagation Strategy

The propagation strategy `const-prop1` for straight-line code is defined in Figure 4. The strategy is almost the same as the `constant-folding` strategy of the previous section. That is, during a bottom-up traversal over the program, constant folding rules are applied. The combined constant folding rules `fold` of the previous section are extended with the dynamically generated `PropConst` rules and with the `assign-cp` strategy for generating these rules. Thus, if a variable is encountered during constant folding, the dynamic rule `PropConst` is applied to discover its constant value and replace it with that value. If the variable is not associated with a constant, the rule will not succeed, and the variable remains in the program.

```
module Tiger-ConstProp
strategies
 const-prop1 = bottomupS(try(fold1 + assign-cp), control-flow1)
 fold1 =  PropConst + fold
 control-flow1(cp) = Assign(id, cp)
```

Fig. 4. Constant propagation for straight line code.

However, the strategy cannot be a complete `bottomup` traversal. Consider, for example, the sequence `a := 3; a := a + 2`. Figure 5 illustrates the visit order through the abstract syntax tree for this sequence using the generic `bottomup(s)` traversal. The numbers on the edges indicate at which point the parameter strategy `s` is applied to the node the edge points to. Clearly the first assignment is visited before the variables in the second one, thus correctly propagating the constant `3` to the variable `a` in the second assignment. However, the variable `a` in the *left-hand side* of the second assignment would also be replaced. To avoid this the constant propagation traversal should not visit the left-hand sides of assignments. This is achieved with the `bottomupS` strategy, a variant of `bottomup`, which is defined as:

```
bottomupS(s, skip: a * (a -> a) -> a) =
  (skip(bottomupS(s, skip))  <+ all(bottomupS(s, skip))) ; s
```

The `skip` parameter provides an alternative for the generic traversal with `all`. In Figure 4 the `skip` strategy is instantiated with the `control-flow1` strategy, which only applies the constant propagation strategy to the right-hand side of assignments. This is achieved using the congruence strategy `Assign(id, cp)`. A congruence strategy `c(s1,...,sn)` matches with a term `c(t1,...,tn)` and constructs the term `c(<s1>t1,...,<sn>tn)`, applying the argument strategies to the corresponding argument terms. Thus, `Assign(id,cp)` applies the identity strategy `id` to the left-hand side of an assignment and constant propagation `cp` to the right-hand side. Thus, achieving a traversal that is faithful to the control flow of the programming language. In Figure 5, the corresponding visit order is the one where visits 1 and 4 are skipped.

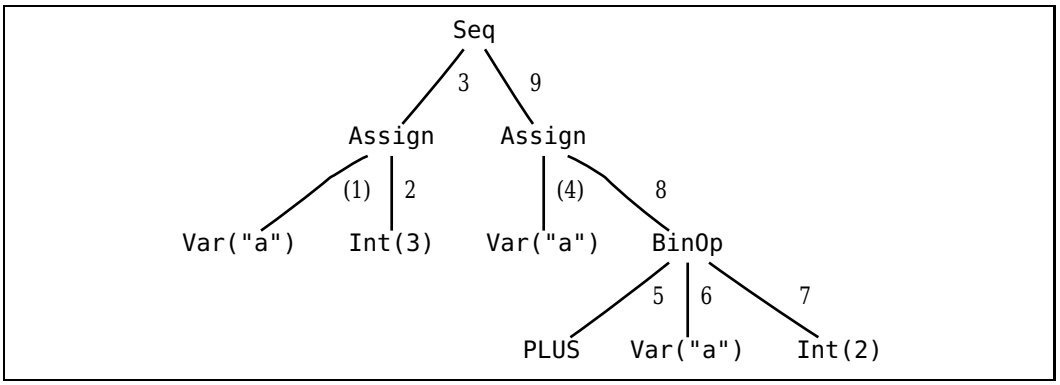

Fig. 5. Visit order for `bottomup(s)`

4 Propagation and Control-Flow

So far we have considered constant propagation in straight line code. Real programs use conditionals and loops which fork or iterate the data flow. Control-flow statements in Tiger consist of the `if-then-else` and `if-then` conditional statements, and the `for` and `while` iterative statements. Figure 6 gives the abstract syntax of these constructs.

The propagation of constant values on control-flow statements forks and merges according to different execution paths. The propagation of constant values has to follow the different paths imposed by the data flow of the program. For instance, consider the following transformation:

```
b := 4;                          b := 4;
d := 2;                          d := 2;
if b > x then                    if 4 > x then
   (a := 6 * d;                     (a := 12;
    b := 45)          ⇒              b := 45)
  else                             else
   (b := 6;                         (b := 6;
    a := b + b);                     a := 12);
c := a + d + b                   c := 14 + b
```

where the variables `a, b, d` are of interest: `b` and `d` contain constant values before the occurrence of the `if` statement and these values are propagated to the branches of the `if`. After the `if`, `a` and `d` contain constant values, `a` contains a constant value, because it is defined with the same constant value in both branches. The variable `d` is not defined in the if statement. On the other hand, `b` is assigned a different value in each branch, hence it does not contain a constant value after the `if`, even though it does contain a constant value *locally* in the branches.

4.1 Unreachable Code Elimination

Control-flow statements fork the execution paths of a program. At execution time some paths are not reachable. To avoid considering unreachable code, the constant folding rules are extended to eliminate branches from control-flow statements when conditions can be evaluated statically. Consider the

```
module Tiger-Statements
signature
 constructors
  IfThen : Exp * Exp -> Exp
  If     : Exp * Exp * Exp -> Exp
  While  : Exp * Exp -> Exp
  For    : Var * Exp * Exp * Exp -> Exp
```

Fig. 6. Control-flow statements of Tiger.

```
ElimIf    : If(Int(0), e1, e2) -> e2
ElimIf    : If(Int(i), e1, e2) -> e1 where <not(eq)> (i, 0)
ElimIfThen: IfThen(Int(0),e1) -> Seq([])
ElimIfThen: IfThen(Int(i),e1) -> e1
ElimWhile : While(Int(0),e) -> Seq([])
ElimFor   : For(v,Int(i),Int(j),e) -> Seq([])
            where <eval-rel(lt)> (j,i) => 1
```

Fig. 7. Rules to eliminate unreachable code

`If` statement with three argument expressions, a `condition`, a `then`, and an `else` expressions. Constant propagation first evaluates the `condition` (as the execution of the program will do). If this evaluation results in a constant value, it determines which branch will be executed (Tiger uses 0 for false and another integer value for true). Unreachable code elimination can be expressed by means of rewrite rules as shown in Figure 7. The application of the `ElimIf` and `ElimIfThen` rules selects the executable branch and the `If` expression is replaced by the surviving branch. The search for constants will continue in the selected branch. If a loop will not be executed it can be removed from the program.

4.2 Distributing and Merging Propagation Rules

The specification of constant propagation is based on dynamic rules to propagate constant values. Dynamic propagation rules must follow the data flow of a program. To achieve this behavior, propagation rules are passed to the execution paths and merged at the point where execution paths are joined. Figure 8 depicts the data flow of an if statement. A structured if-then-else statement forks and merges two execution paths. At the point where the if-then-else statement forks, dynamic propagation rules are passed to the `then` and `else` branches. At the merge point, two different versions of dynamic propagation rules are combined.

Propagation rules carry the information required to propagate constants. These rules have to traverse the program in a way that emulates the data flow of a program. In order to manipulate active propagation rules, operations such as save, restore and intersect(merge) are used.

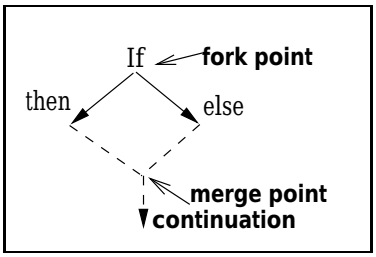

Fig. 8. Data flow of a structured If statement

Input Term	*Transformation Rules*		*Transf. Term*
	Active rules	*Sets of rules*	
`b := 4;`	`PC: Var(b) -> 4` *(1)*		`b := 4;`
`d := 2;`	`PC: Var(d) -> 2` *(2)*		`d := 2;`
`if b>x then`	*saving active rules*	`dr-in ={PC: Var(b) -> 4,` `PC: Var(d) -> 2}`	`if 4>x then`
`(a := 6 * d;` `b := 45 )`	`PC: Var(a) -> 12` *(3)* `PC: Var(b) -> 45` *(4)*		`(a := 12;` `b := 45)`

`else`	*saving active rules*
	
	`PC: Var(b) -> 4` *(5)*
	`PC: Var(d) -> 2` *(6)*
		`dr-then ={PC: Var(d) -> 2,`
		`PC: Var(a) -> 12,`
		`PC: Var(b) -> 45}`
		restoring dr-in rules
			`else`

`(b := 6;` `a := b + b);`	`PC: Var(b) -> 6` *(7)* `PC: Var(a) -> 12` *(8)*	*overrides rule (5)*	`(b := 6;` `a := 12);`

	⋂ `dr-then =`
	`{PC: Var(d) -> 2,`
	`PC: Var(a) -> 12}`
		intersection with active rules (6,7,8)	

`c:= a + d + b`	`PC: Var(c) ->` *Undefined*		`c := 14 + b`

Fig. 9. Example of how dynamic rules are used to propagate constants

Figure 9 illustrates how distributing and merging of propagation rules is accomplished. The example is decorated with two extra columns to illustrate the result of operations on dynamic rules. For space reasons, the name of the rule `PropConst` is depicted as `PC`. The first and last column represent the input and the transformed term. The intermediate columns are introduced to show how dynamic rules are used to propagate constant values. The second column shows the state of the active rules. The third column shows the result of an operation on dynamic rules which uses extra storage.

Each row in the table shows an inspected node, the result of an operation on dynamic rules, and the transformed term. The first and the second row of the table generate a propagation rule respectively. The if statement evaluates the condition and saves active propagation rules in `dr-in` to be passed to its branches. The `then` branch is traversed and the resulting state of propagation rules is stored in `dr-then` set. To traverse the `else` branch with the state of incoming propagation rules to the if statement, rules in `dr-in` set are restored. At the merge point of the if statement an intersect operation is specified to obtain the active propagation rules valid after the if statement.

The specification of constant propagation for an if statement is defined by the following strategy:

```
if-cp(cp) =
  If(id, id, id)
  ; where(save-PropConst => dr-in)
  ; If(id, cp, id)
  ; where(save-PropConst => dr-then; <restore-PropConst> dr-in)
  ; If(id, id, cp)
  ; where(<isect-PropConst> dr-then)
```

The strategy uses the operations `save-PropConst`, `restore-PropConst`, and `isect-PropConst` to manipulate the dynamic rule. The `where(s)` strategy applies a strategy `s` to a term without modifying the term.

In order to distribute incoming active rules to the branches of an if, the active set of propagation rules is saved. Constant propagation continues traversing the `then` branch, the result of this inspection is saved in the `dr-then` set of dynamic rules. To proceed in the `else` branch, a restore operation is defined to activate the `dr-in` set. With this state, constant propagation continues in the `else` branch. To merge two different versions of propagation rules from both branches, an intersect operation is specified. The intersection is specified with the current state of dynamic rules (active rules considering the else branch) and the `dr-then` set. A propagation rule is maintained in the active set if it is defined in both sets and rewrites to the same constant value.

4.2.1 Generalization

The data flow schema for `if` statements above was specific for constant propagation. It turns out that the schema can be reused for other optimizations by using different save, restore, and merge operations. This generalization can be formalized by abstracting the schema over these operations, as follows:

```
if(s, save, restore, merge) =
  If(id, id, id)
  ; where(save => dr-in)
  ; If(id, s, id)
  ; where(save=> dr-then; <restore> dr-in)
  ; If(id, id, s)
  ; where(<merge> dr-then)
```

An example of instantiation of this schema is `if(cp, save-PropConst, restore-PropConst, isect-PropConst)` for constant propagation. Different analyses and optimizations can reuse this building block by providing different strategies to operate on dynamic rules. In the specification of the other control-flow constructs we take this general approach into account.

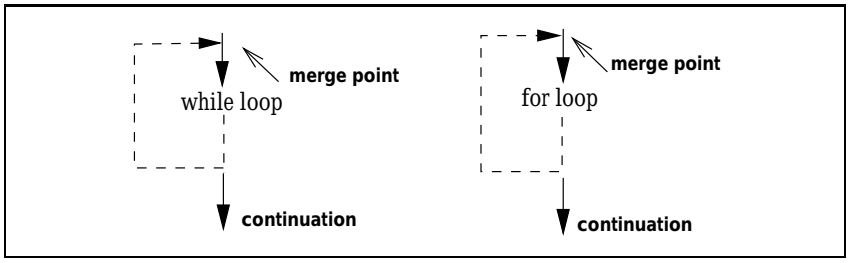

Fig. 10. Merging points of control flow statements.

4.3 Loop statements

Loop statements execute a number of times (including zero) the body of a loop statement. The execution flow of a loop has a reentering path as is depicted in Figure 10. Therefore, loop statements fork the data flow and iterate the body of the loop.

Structured loop statements execute the iteration conditions as the first and last execution. As a motivating example for constant propagation on loops; consider the following code:

```
(a := 5;
 b := 4;
 c := 8;
 d := 5;
 while((c < 100)|x)
 do(b := a;
    e := a + b;
    a := e * d + c;
    c := c + e;
    e := e + c;
    a := b);
 f := a + b + d + c + e)
```

$\Rightarrow$

```
(a := 5;
 b := 4;
 c := 8;
 d := 5;
 while((c < 100)|x)
 do(b := 5;
    e := 10;
    a := (50 + c);
    c := (c + 10);
    e := (10 + c);
    a := 5);
 f := (15 + c + e))
```

In this example the variables `a`,`b`,`c` and `d` contain constant values before the while statement is inspected. This example illustrates the following situations:

- `a` contains a constant value, regardless of the execution of the while statement.
- `b` is defined in the body of the while, if the while is executed at least one iteration, b contains the value of `a`, i.e `5`, and it can be propagated outside the while statement.
- `c` does not contain a constant value because it is iteratively defined in the while body.
- `d` contains a constant value since it is not defined in the while.
- `e` contains a constant value from its first definition until its second definition inside the while loop. After the while `e` does not contain a constant value.

```
while-p(s,save,restore,merge)=
  While(id, id)
  ; where(save => dr)
  ; where(While(id, s))
  ; where(While(id, s))
  ; where(<merge> dr)
  ; While(s, s)
  ; where(<merge> dr)
```

```
for-p(s,save, restore,merge)=
  For(id, id, id, id)
  ; where(save => dr)
  ; where(For(id, id, id, s))
  ; where(For(id, id, id, s))
  ; where(<merge> dr)
  ; For(id, id, id, s)
  ; where(<merge> dr)
```

Fig. 11. Data flow schemas for loops

When the assignment to `f` is reached, the variables `a, b` and `d` contain constant values.

The specification of constant propagation has to cover all these cases. To discover that the value of `a` contains a constant value, we inspect the while body twice, in order to take into account propagation rules discovered in the first inspection. This analysis considers the reentering path of the loop. After the unfolded inspection of the while, the merge operation selects valid propagation rules considering possible execution paths. With the outcome of valid propagation rules the transformation is performed. A last merge operation is required to prevent propagation rules coming from the while execution being active outside the scope of the while loop. The `for` statement is defined in an analogous way.

The definition of constant propagation for loop statement is shown in Figure 11.

4.3.1 Non Pessimistic Constant Propagation

The constant propagation specification so far is still pessimistic, even though we avoid considering unreachable code. Loop statements that are not executed are removed from the program when the condition can be evaluated. If the analysis cannot determine if a loop will execute or not, it considers both situations. That is the reason for the second occurrence of `merge` for loop statements. This specification cannot determine the value of `b` as constant in the last example.

A less pessimistic approach is to keep propagation rules after a loop statement when the analysis guarantees at least one execution of the body of a loop. The `is-safe` strategy checks this condition and avoids performing the second `merge` of propagation rules. Thus, constant propagation rules of a loop statement are active after the loop occurrence in the program. The strategies in Figure 12 consider these situations.

4.4 Strategies for Control-flow Statements

Now we have defined all ingredients for putting together a full blown constant propagation strategy. Figure 13 combines the ingredients into a specification

```
is-safe =
 ?While(Int(i),e);
 where(<not(eq)> (i,0);
       <not(oncetd(Break))> e)

while(s,save,restore,meet) =
 While(id,id)
 ; where(save => dr)
 ; (where(While(s,id);is-safe)
    ; where(While(id,s))
    ; where(While(id,s))
    ; where(<meet> dr)
    ; While(s,s)
 <+ where(While(id,s))
    ; where(While(id,s))
    ; where(<meet> dr)
    ; While(s,s)
    ; where(<meet> dr)
 )
```

```
is-safe =
  ?For(_,Int(i),Int(t),_);
  where( <geq> (t, i))


for(s,save,restore,merge) =
  For(id,id,id,id)
  ; where(save => dr)
  ; where(For(id,id,id,s))
  ; where(For(id,id,id,s))
  ; where(<merge> dr)
  ; For(id,id,id,s)
  ; ( is-safe
     <+ where(<merge> dr) )
```

Fig. 12. Non-pessimistic loop data flow schemas

```
const-prop2 =
 bottomupS(
  try(fold2(const-prop2) <+
  eval-conditionals2(const-prop2))
  , control-flow2
 )

control-flow2(s) =
   Assign(id, s)
 + IfThen(s, id)
 + If(s, id, id)
 + For(id, s, s, id)
 + While(id, id)
 + Let([try(VarDec(id,id,s))]
        , id)

fold2(s) =
   fold1
 + ElimIf; s
 + ElimFor
 + ElimIfThen; s
 + While(s,id); ElimWhile
```

```
eval-conditionals2(s) =
  eval-conditionals(s,
        assign-cp,
        declVar,
        save-PropConst,
        restore-PropConst,
        isect-PropConst)

eval-conditionals(s,
    assign, declare,
    save, restore, meet) =
  assign
+ ifthen(s,save,restore,meet)
+ if(s,save,restore,meet)
+ for(s,save,restore,meet)
+ while(s,save,restore,meet)
+ let-exp(s,declare)

let-exp(s, declare) =
  Let([try(declare)],list(s))
```

Fig. 13. Constant propagation for control-flow statements

of non-pessimistic intra-procedural constant propagation. The `const-prop2` strategy is defined in terms of the `bottomupS` strategy and includes strategies for control-flow statements. The non-pessimistic constant propagation uses the evaluation rules `ElimIf`, `ElimIfThen`, `ElimWhile`, and `ElimFor`, to simplify the execution flow of expressions when possible. The strategy `fold2` is parameterized with the `cp` strategy to apply constant propagation to the resulting paths after the simplification of statements.

When constant folding reduction is not applicable we consider different execution paths as explained before. The `eval-conditionals2` strategy is defined in terms of `assign-cp`, and the data flow schemas for conditionals and loops. The `control-flow1` strategy is extended with terms that need control when the reduction strategy is applied. The strategy `control-flow2` defines how to traverse control-flow statements to simulate the execution of the program.

5 Discussion

We have shown how to specify intra-procedural constant propagation using strategic dynamic term rewriting. The techniques used in the paper such as dynamic scoped rules and context-sensitive term traversal permitted us to write a specification which is elegant and concise.

5.1 Generalization and Future Work

As presented here, the constant propagation optimization is intra-procedural and does not take function calls into account. We have specified an extension that analyzes function definitions and at function calls only undefines propagation rules for global variables killed in the function body. Generalization to inter-procedural constant propagation [7,5] needs further investigation, but seems straightforward.

Other source-to-source optimizers can be built using the same building blocks by providing different propagation rules and corresponding rule operations. We have already specified copy propagation, common sub-expression elimination and dead code elimination using this approach. We are confident that the techniques can be used for the specification of many other machine independent optimizations such as code motion [11], although this requires further investigation.

The current building blocks can be easily extended to include control-flow constructs that are not present in Tiger such as case and switch. Also extension to semi-unstructured control-flow using breaks can be dealt with. The framework can be made reusable with little effort across different languages.

Different dynamic rules can be created to propagate or to provide information for different aspects of a program. At the moment we are interested in investigating the possibility of combining several optimizations while a pro-

gram is being inspected in the style of [13].

Constant propagation or in general information propagation is been used in partial evaluators [15], supercompilers [10], interpreters [8], as a few examples. In [8] dynamic rules are used to propagate information of the evaluation state of expressions. The present work can be extended and combined with other techniques such as program specialization and generalization in order to construct such systems.

5.2 Previous Work

This paper is part of an ongoing investigation into the specification of different kinds of program transformations using rewriting strategies and dynamic rules. In [20] programmable rewriting strategies, in particular for generic traversal, were introduced and applied to simplification of functional programs. In [18] dynamic rules are introduced and applied to bound variable renaming, function inlining, and dead function elimination. In [6] dynamic rules are used to implement dynamic programming for instruction selection, i.e., associating the lowest cost reduction with a term. The Tiger optimization presented in this paper is part of the Tiger-in-Stratego [1] project, which is aimed at exploring program transformation techniques for compilation.

The contribution of this paper is the formalization of merge operations for dynamic rules and the traversal strategy that emulates the control flow of a program. These mechanisms allow us to write the building blocks that are reusable for other optimizations that require a *forward* analysis. These building blocks can be parameterised for different optimizations, providing different information through dynamic rules for each aspect that is relevant to inspect. A *backward* traversal of the program was implemented using similar constructions with dead code elimination as an instance.

5.3 Related Work

Path logic programming [9] is a related specification technique for data-flow optimizations. The applicability conditions of a transformation rule are expressed by means of regular expressions over the paths through a program. For example, a constant assignment can be propagated to a usage site if there are no intervening definitions of the variable on any path between definition and use. Instead of transforming abstract syntax trees, control-flow graphs are used as basis for transformations. The specification of optimizations is more declarative than the one presented in this paper. On the other hand, our approach allows for the simultaneous propagation of all constants or other properties throughout the program, whereas in the path logic programming approach, the entire graph needs to be reanalyzed after each transformation.

[1] `http://www.stratego-language.org/Tiger`

Expressing transformations by rewriting tends to be clear and intuitive. There are three major factors to consider for a rewrite system. Expressiveness of transformations, representation of control-flow paths, and data availability to test applicability conditions to perform rewriting. Rewriting is split into two different areas: term rewriting and graph rewriting [3]. Term rewriting expresses transformations in a clear and elegant way, but it is more difficult to express applicability conditions, when data not available in the inspected term is needed, term rewriting systems suffers from this issue. In contrast, graph rewriting provides mechanisms to represent the control flow of a program, the applicability conditions of rewriting are also easier to represent, although the rewriting itself is more difficult to represent [12]. Providing information to term rewriting by means of dynamic rules and a strategy to emulate the data flow of programs leads to a high level specification for constant propagation which is directly executable.

There are many algorithms for constant propagation, most of them use a graph representation such as control-flow graphs with the *static single assignment* (SSA) property [21], or even the *gated single assignment* property [17]. Special ϕ assignments are introduced to provide the representation of a program with unique variable definitions. The most used algorithm for constant propagation is the *Sparse conditional constant propagation* (SCC) [21,17,14,2]. SCC is an optimistic algorithm which avoids to consider unreachable code and continues the propagation of constant values in the branches of conditionals statements. These algorithm properties are also present in our specification.

Another approach uses *system dependence graphs* (SDG) to represent the data-flow and control-flow of a program [5]. This representation focuses on data dependencies of a program, which is represented by means of graphs. This system is provided with a data-flow interpreter to propagate constant values to the dependent nodes. Assignment nodes provide the hints to propagate constants in the graph. At each node the SDG graph is updated following transformation rules. The SDG approach is more convenient for source-to-source optimizers than algorithms based on control-flow graphs with SSA property, since they do not lower the representation language. Program reconstruction preserves the structure of the program, although building the system dependence graph is required in order to perform the transformation.

In all publications we encountered (with [9] as exception) algorithms for constant propagation are expressed using pseudocode or using formal English to describe properties of a system. Our specification is consice and executable without the need for def-use chains [1] or similar analysis information. All information required is present in the rewriting rules and the mechanisms to operate on them suffices for our specification.

6 Conclusion

In this paper we have described the application of programmable rewriting strategies to the specification of a data-flow optimization. Programable strategies allow us to emulate the data flow of a program, even though using abstract syntax trees to represent a program. The parameterization of the traversal strategies with the rules to apply makes it possible to define different optimizations by providing a different set of rules and even enables us to combine different optimizations.

The use of dynamic rules for the propagation of data-flow information (instead of environments) makes it possible to separate traversal from propagation. The extension of dynamic rules with operations to save, restore, and intersect sets of rules makes it possible to model data-flow through multiple paths. Together, these mechanisms support the combination of analysis and transformation while the abstract syntax tree is traversed.

References

[1] Aho, A., R. Sethi and J. Ullman, "Compilers: Principles, Techniques, and Tools," Addison-Wesley, 1986.

[2] Appel, A., "Modern compiler implementation in ML," Cambridge University Press, 1998.

[3] Assmann, U., *How To Uniformly Specify Program Analysis and Transformation*, in: T. Gyimóthy, editor, *Internationational Conference on Compiler Construction (CC'96)*, Lecture Notes in Computer Science **1060** (1996), pp. 121–135.

[4] Bagge, O., M. Haveraaen and E. Visser, *CodeBoost: A framework for the transformation of C++ programs*, Technical Report UU-CS-2001-32, Institute of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands (2001).

[5] Binkley, D., *Interprocedural constant propagation using dependence graphs and a data-flow model*, in: P. A. Fritzson, editor, *International Conference on Compiler Construction (CC'94)*, Lecture Notes in Computer Science **786** (1994), pp. 374–388.

[6] Bravenboer, M. and E. Visser, *Rewriting strategies for instruction selection*, in: S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, Lecture Notes in Computer Science **2378** (2002), pp. 237–251.

[7] Carini, P. R. and M. Hind, *Flow-sensitive interprocedural constant propagation*, in: *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California*, SIGPLAN Notices **30/6**, 1995, pp. 23–31.

[8] Dolstra, E. and E. Visser, *Building interpreters with rewriting strategies*, in: M. van den Brand and R. Laemmel, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'02)*, Electronic Notes in Theoretical Computer Science **65/3** (2002).

[9] Drape, S., O. de Moor and G. Sittampalam, *Transforming the .NET intermediate language using path logic programming*, in: C. Kirchner, editor, *Proceedings of the Fourth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'02)*, ACM, Pittsburgh, Pensylvania, USA, 2002, pp. 133–144.

[10] Glück, R. and M. H. Sørensen, *A roadmap to metacomputation by supercompilation*, in: O. Danvy, R. Glück and P. Thiemann, editors, *Partial Evaluation*, Lecture Notes in Computer Science **1110** (1996), pp. 137–160.

[11] Knoop, J., O. Ruthing and B. Steffen, *Partial dead code elimination*, in: *SIGPLAN Conference on Programming Language Design and Implementation*, 1994, pp. 147–158.

[12] Lacey, D. and O. de Moor, *Imperative program transformation by rewriting*, in: R. Wilhelm, editor, *Proceedings of the 10th International Conference on Compiler Construction*, Lecture Notes in Computer Science **2027** (2001), pp. 52–68.

[13] Lerner, S., D. Grove and C. Chambers, *Combining dataflow analyses and transformations*, in: *SIGPLAN Symposium on Principles of Programming Languages (POPL 2002)*, Portland, Oregon, 2002, pp. 270–282.

[14] Muchnick, S., "Advanced compiler design and implementation," Morgan Kaufmann Publishers, 1997.

[15] Sakharov, A., *Specialization of imperative programs through analysis of relational expressions*, in: O. Danvy, R. Glück and P. Thiemann, editors, *Partial Evaluation*, Lecture Notes in Computer Science **1110** (1996), pp. 430–445.

[16] Standford Compiler Group, *The SUIF library,1.0 edition* (1994).

[17] Stoltz, E., M. Wolfe and M. Gerlek, *Constant propagation: A fresh demand-driven look*, in: *Proceedings of ACM Symposium on Applied Computing*, ACM SIGAPP (1994), pp. 400–404.

[18] Visser, E., *Scoped dynamic rewrite rules*, in: M. van den Brand and R. Verma, editors, *Rule Based Programming (RULE'01)*, Electronic Notes in Theoretical Computer Science **59/4** (2001).

[19] Visser, E., *Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5*, in: A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, Lecture Notes in Computer Science **2051** (2001), pp. 357–361.

[20] Visser, E., Z.-e.-A. Benaissa and A. Tolmach, *Building program optimizers with rewriting strategies*, in: *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)* (1998), pp. 13–26.

[21] Wegman, M. and F. Zadeck, *Constant propagation with conditional branches*, ACM Transactions on Programming Languages and Systems **13** (1991), pp. 181–210.

[22] Wilson, R., R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam and J. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," (1996).