Composing Source-to-Source Data-Flow Transformations with Rewriting Strategies and Dependent Dynamic Rewrite Rules

Karina Olmos
Eelco Visser

Technical Report UU-CS-2005-006
Institute of Information and Computing Sciences
Utrecht University

June 1, 2005

Preprint of:

K. Olmos and E. Visser. Composing Source-to-Source Data-Flow Transformations with Rewriting Strategies and Dependent Dynamic Rewrite Rules. In R. Bodik, editor, 14th International Conference on Compiler Construction (CC'05), volume 3443 of Lecture Notes in Computer Science, pages 204–220. Springer-Verlag, April 2005.

Copyright © 2005 Karina Olmos, Eelco Visser

ISSN 0924-3275

Address:
Institute of Information and Computing Sciences
Utrecht University
P.O.Box 80089
3508 TB Utrecht

Eelco Visser
visser@acm.org
http://www.cs.uu.nl/~visser

Composing Source-to-Source Data-Flow Transformations with Rewriting Strategies and Dependent Dynamic Rewrite Rules

Karina Olmos
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 transformations used in optimizing compilers are also useful in other programming tools such as code generators, aspect weavers, domain-specific optimizers, and refactoring tools. These applications require source-to-source transformations rather than transformations on a low-level intermediate representation. In this paper we describe the composition of source-to-source data-flow transformations in the program transformation language Stratego. The language supports the high-level specification of transformations by means of rewriting strategy combinators that allow a natural modeling of data- and control-flow without committing to a specific source language. Data-flow facts are propagated using dynamic rewriting rules. In particular, we introduce the concept of dependent dynamic rewrite rules for modeling the dependencies of data-flow facts on program entities such as variables. The approach supports the combination of analysis and transformation, the combination of multiple transformations, the combination with other types of transformations, and the correct treatment of variable binding constructs and lexical scope to avoid free variable capture.

1 Introduction

Optimizing compilers rely on data-flow facts to perform optimizations [1, 12]. 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. The implementation of these optimizations is hidden from programmers using the compiler. Data-flow transformations are useful outside the core of compilers as well. In generative programming, high-level and model-driven code generation, refactoring, aspect weaving, open compilers, and domain- and application-specific optimization, transformations are an essential part of program development. While data-flow optimizations in compilers are usually implemented to work on fixed low-level intermediate representations, these applications require transformations on source code in high-level programming languages. Furthermore, compiler optimizations are traditionally implemented in general purpose languages, optimizing for speed of the transformations rather than productivity of the

transformation writer. Higher productivity can be achieved using a language and environment that provides more support for the domain of program transformation. For such an environment for source-to-source transformations to be widely applicable it should cover a wide spectrum of transformational tasks. That is, it should not be specific to one source language and should not restrict support to one type of transformation. Rather, it should provide high-level abstractions for modeling control- and data-flow of the language under consideration, and it should support combination of data-flow transformations with other types of program manipulation such as template based code generation. Also, the environment should not require abstraction from details of program representation and should for instance support handling issues of scope of variables and help to avoid problems such as free variable capture.

In this paper we describe the composition of source-to-source data-flow transformations in the program transformation language Stratego [19]. The language is not restricted to data-flow transformations nor is it restricted to transformations on a specific source language. Instead of building-in knowledge about data-flow, Stratego provides high-level ingredients for composing data-flow transformations on abstract syntax trees. These ingredients are *rewrite rules* for definition of basic transformations, *programmable rewriting and traversal strategies* for the composition of tree traversals and controlling the application of rewrite rules, *dynamic rewrite rules* for propagation of context-sensitive information such as data-flow facts, and *dynamic rule combinators* for modeling control-flow (forks in data-flow). In particular, we introduce the concept of *dependent dynamic rewrite rules* for modeling dependencies of data-flow facts on program entities such as variables. Together these techniques support:

- An abstract interpretation style of data-flow transformation that allows the combination of data-flow analysis and transformation in the same traversal.
- The correct treatment of variable binding constructs and lexical scope to avoid free variable capture and to restrict the application of transformation rules to the scope where they are valid.
- The definition of generic data-flow strategies, which allow concise specifications of data-flow transformations, and the concise combination of multiple transformations into 'super-optimizers'.
- The combination of data-flow transformations with other types of transformations, reuse of elements of a transformation in other transformations, and easy experimentation with alternative transformation strategies.

We proceed as follows. In the next section we describe rewrite rules, strategies, and dynamic rules and illustrate their use in a specification of constant propagation. In Section 3 we motivate the need for *dependent* dynamic rules and illustrate their use in a specification of copy propagation. In Section 4 we generalize the strategies for constant propagation and copy propagation into a generic strategy for forward data-flow propagation and instantiate the strategy to common-subexpression elimination. We also show how, using the same generic strategy, the components of these transformations can be combined in a single super-optimizer. In Section 5 we discuss previous, related, and future work.

2 Rewriting Strategies and Dynamic Rules

In this section we show how rewriting strategies in combination with dynamic rewrite rules can be used to compose data-flow transformations on abstract syntax trees, using constant propagation as running example. Throughout the paper we use a subset of Appel's Tiger language [2] as the source language for transformations. The abstract syntax of this subset is defined in **Fig. 1**. However, none of the techniques we present are specific to this language. We assume the reader to be familiar with the basic notions and infrastructure for source-to-source transformations on abstract syntax trees, including parsing, tree representation, and pretty-printing. For an overview of the specific infrastructure used in the Stratego/XT framework we refer to [19].

2.1 Local Transformations with Rewrite Rules

Basic transformations on abstract syntax trees can be defined using *tree* or *term rewriting*. A *rewrite rule* (p_1 `->` p_2) defines the transformation of a tree that matches the left-hand side p_1 of the rule to the instantiation of the right-hand side p_2 of the rule. Term rewriting is the *normalization* of a tree by exhaustively applying a set of rewrite rules. **Fig. 2** shows some typical rewrite rules for constant folding and unreachable code elimination. Note that we use *concrete syntax* [18] to describe the abstract syntax tree patterns in the left-hand side and right-hand side of the rules. That is, a phrase such as `|[ if i then e1 else e2 ]|` denotes a tree pattern `If(Int(i),e1,e2)` where `i`, `e1`, and `e2` are meta-variables. Using the rules from **Fig. 2** the arithmetic expression `2 + 3 + 7` rewrites to `12` and the conditional expression `if 0 then x := 1 else x := 2` reduces to `x := 2`.

Term rewriting is declarative since rewrite rules can be defined independently and are automatically applied by a rewriting engine. The correctness of the combined transformation can be established by the correctness of the individual rules. However, static rewrite rules are not sufficient for defining data-flow transformations. A rewrite rule can only use information from the term to which it is applied, not from its parents or siblings in the abstract syntax tree. Data-flow transformations typically need information from assignments and variable declarations that are higher-up in the tree. For example,

d ::= `var` x `:=` e
e ::= x | str | i | $e_1 \oplus e_2$ | f`(`$e^*_;$`)` | x `:=` e | `(`$e^*_;$`)` | `if` e_1 `then` e_2 `else` e_3
   | `while` e_1 `do` e_2 | `let` d^* `in` $e^*_;$ `end`

Fig. 1. Abstract syntax for a subset of Tiger with $\oplus$ the usual arithmetic, relational, and Boolean operators. The non-terminals x, f, str, and i denote variables, functions, string, and integer constants, respectively. $e^*_;$ denotes a list of zero or more expressions separated by semicolons.

```
EvalBinOp : |[ i + j ]| -> |[ k ]| where <add>(i, j) => k
EvalBinOp : |[ i * j ]| -> |[ k ]| where <mul>(i, j) => k
EvalWhile : |[ while 0 do e ]| -> |[ () ]|
EvalIf    : |[ if 0 then e1 else e2 ]| -> |[ e2 ]|
EvalIf    : |[ if i then e1 else e2 ]| -> |[ e1 ]| where <not(eq)>(i,|[0]|)
```

Fig. 2. Some rewrite rules for constant folding and unreachable code elimination.

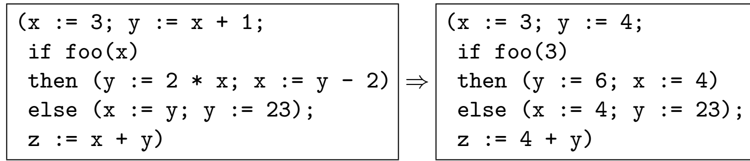


Fig. 3. Example of constant propagation.

consider the constant folding and propagation transformation in **Fig. 3**. The fact that the variable `x` is constant allows constant folding in many of the subsequent expressions. However, this requires the propagation of the initial constant value of `x` to its uses; e.g., folding the expression `x + 1` is only possible after replacing `x` with its value.

2.2 Context-sensitive Transformations with Dynamic Rewrite Rules

To extend rewriting to propagation of context-sensitive information requires (1) the dynamic (run-time) definition of rewrite rules and (2) the careful control of their application. We first consider the use of dynamic rewrite rules to propagate data-flow information in a control-flow graph and then argue that this approach can also be applied to abstract syntax trees. In the next subsection we then show how this transformation on abstract syntax trees is realized in Stratego.

The left diagram in **Fig. 4** depicts the control-flow graph of the example program in **Fig. 3**. The nodes correspond to the assignments and conditionals in the program before and after transformation. The traversal of the graph follows the control-flow of the program, which corresponds to following the direction of the arrows from entry to exit. At nodes with more than one outgoing edge, the traversal subsequently visits each branch and synchronizes at the merge point. Data-flow facts are represented by a set of dynamic rewrite rules `(x -> i)` that rewrite an occurrence of a variable to its constant value. Since the set of propagation rules can be different at each point in the program, the edges of the graph are annotated with the rules that are valid at that point of the traversal.

At each node of the graph, first the right-hand side of the assignment is transformed by rewriting variables in the expression to constant values, if applicable, and attempting to apply constant folding rules such as in **Fig. 2**. For example, `y := x + 1` is transformed to `y := 3 + 1` by application of the rule `x -> 3` and then reduced to `y := 4` by constant folding. Next, an assignment x `:=` e causes the undefinition of any rules with x as left-hand side, since these are no longer valid. Finally, if the assignment has a constant value as right-hand side (x `:=` i), a new rewrite rule x `->` i is defined.

Multiple propagation rules for *different variables* can be defined at the same time. For example, after the `y := x + 1` assignment both rules `x -> 3` and `y -> 4` are valid. However, only one rule can be defined with the same left-hand side. For example, The assignment `x := y` replaces the rule `x -> 3` with the rule `x -> 4`. At a fork in the control-flow, that is at a node with more than one outgoing edge, each branch starts with the rule-set valid at the branching node. That is, each edge is annotated with a clone of that rule-set. At the merge point only those rules that are consistent in all branches are maintained. In the example, the rules for `y` are inconsistent at the merge point and are

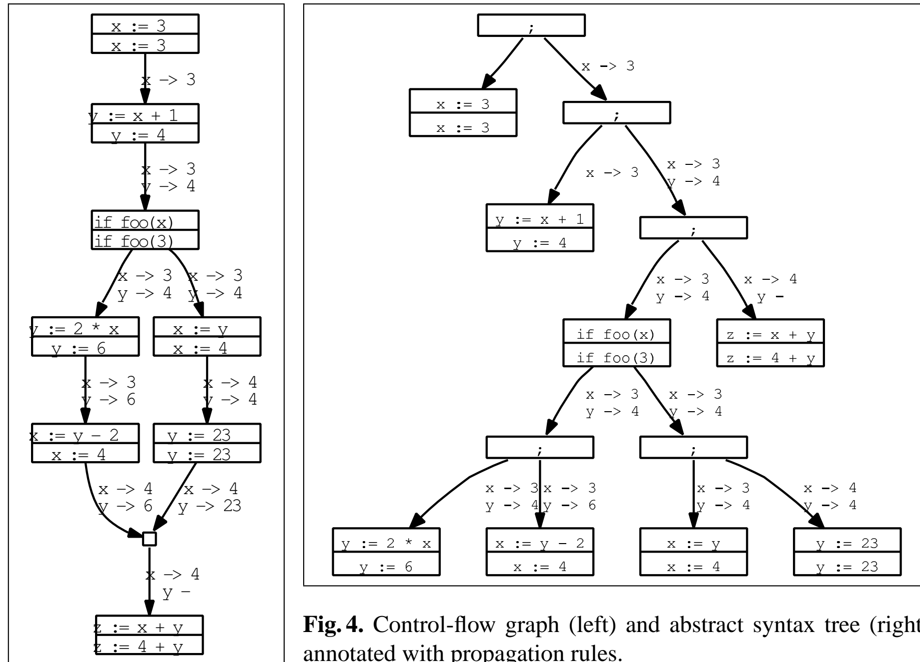


Fig. 4. Control-flow graph (left) and abstract syntax tree (right) annotated with propagation rules.

undefined. In the case of loops this process should be repeated until a stable set of rules is obtained.

A control-flow graph traversal of a program can also be realized by traversing its abstract syntax tree. This requires visiting the nodes of the tree in the order that they would be visited in a traversal of the graph. The right diagram in **Fig. 4** depicts the abstract syntax tree of the example program. Simulation of the traversal corresponds basically to a depth-first left-to-right traversal of the syntax tree. Realization of the constant propagation transformation on abstract syntax trees thus requires

- traversal of the abstract syntax tree to visit expressions in the right order
- dynamic definition of rules to reflect the constant assignments
- application of dynamic propagation rules and static constant folding rules
- forking and combining rule-sets to model forks in data-flow

2.3 Realization in Stratego

The Stratego program in **Fig. 5** defines a constant propagation transformation strategy implementing the propagation of dynamic rules as described above. In the rest of this section we examine the components of this definition and informally introduce the Stratego constructs that they use. The definition of a subset of the abstract syntax of Stratego in **Fig. 6** should be helpful in understanding the structure of Stratego programs. A full description of the language is beyond the scope of this paper; see [4, 19].

Rules and Strategies Rewrite rules as described in Section 2.1 are the basic entities of Stratego programs. A named rule f : p_1 `->` p_2 transforms a term matching pattern p_1

```
prop-const =
  PropConst <+ prop-const-assign <+ prop-const-declare
  <+ prop-const-let <+ prop-const-if <+ prop-const-while
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  |[ x := <prop-const => e> ]|
  ; if <is-value> e then rules( PropConst.x :  |[ x ]| -> |[ e ]| )
                    else rules( PropConst.x :- |[ x ]| ) end

prop-const-declare =
  |[ var x := <prop-const => e> ]|
  ; if <is-value> e then rules( PropConst+x :  |[ x ]| -> |[ e ]| )
                    else rules( PropConst+x :- |[ x ]| ) end

prop-const-let =
  ?|[ let d* in e* end ]|; {| PropConst : all(prop-const) |}

prop-const-if =
  |[ if <prop-const> then <id> else <id> ]|
  ; (EvalIf; prop-const
     <+ (|[ if <id> then <prop-const> else <id> ]|
           /PropConst\ |[ if <id> then <id> else <prop-const> ]|))

prop-const-while =
  ?|[ while e1 do e2 ]|
  ; (|[ while <prop-const> do <id> ]|; EvalWhile
     <+ (/PropConst\* |[ while <prop-const> do <prop-const> ]|))
```

Fig. 5. Constant propagation transformation strategy.

```
     program P  ::= (rules | strategies) d*
  definition d  ::= h = s | h : r
      header h  ::= f(f*,|x*,) | f(f*,) | f
        rule r  ::= p1 -> p2 (where s)?
     pattern p  ::= str | i | r | x | c(p*,) | (p*,) | [p*,|p] | [p*,] | <s> p | <s>
    strategy s  ::= ?p | !p | {x*,: s} | <s> p | s => p
                  | s1 ; s2 | f(s*,|p*,) | f(s*,) | f
                  | s1 <+ s2 | s1 < s2 + s3 | if s1 then s2 (else s3)?
                  | fail | id | not(s) | where(s) | let d* in s end
                  | rules(drd*) | {|f*,: s |} | s1 /f*,\ s2 | /f*,\* s
dyn. rule def. drd ::= h((.|+)p)? :(+)? dr  | h((.|+)p)? :- p
   dyn.rule dr  ::= r | r depends on p
```

Fig. 6. Abstract syntax of a subset of Stratego. The following additional non-terminals are used: str, i and r denote string, integer, and real constants; x a pattern variable, c a constructor, f a strategy operator. Operators are listed in the order of precedence; in particular ; has precedence over <+. Note that the use of concrete syntax for patterns and congruence strategies is not covered by this abstract syntax definition.

to the instantiation of pattern p_2. Some example rewrite rules are shown in **Fig. 2** using *concrete syntax* for the term patterns. Stratego extends the basic notion of term rewriting with programmable strategies for the controlled application of rewrite rules. A rule with name f defines a transformation from terms to terms. A rule may fail to apply to a term, e.g., when its left-hand side does not match the term it is applied to. Strategies combine rules into more complex transformations using a number of *strategy combinators*. Since rules can fail, strategies can fail to apply to a term as well. Strategy *definitions* of the form f `=` s name a strategy expression. Thus, **Fig. 5** introduces six, mutually recursive, definitions that compose the constant propagation strategy `prop-const`.

The basic strategy combinators are sequential composition s_1`;` s_2 (first apply s_1 and then s_2) and deterministic choice s_1 `<+`s_2 (first apply s_1, if that fails apply s_2). Note that sequential composition has higher precedence than deterministic choice. Thus, the `prop-const` strategy defines a choice between seven cases, which are tried in turn until one succeeds.

Term Traversal In order to transform sub-terms of a term, a strategy needs to *traverse* the term. While in conventional languages traversal requires a tedious enumeration of all elements of the data structure and their traversal, Stratego supports *generic* traversal through *one-level traversal combinators* [20]. One of these combinators is `all(`s`)`, which applies s to each direct sub-term of the subject term. Thus, the basic schema of the `prop-const` strategy is

```
prop-const = PropConst <+ (all(prop-const); try(EvalBinOp))
```

which either applies the `PropConst` dynamic rule to replace a variable by a constant value or recursively visits the direct sub-terms with a recursive call to the `prop-const` strategy (`all(prop-const)`) and then tries to apply a constant folding rule. The other cases in the definition of `prop-const` introduce exceptions to the generic traversal. For example, only the right-hand side of an assignment should be visited, and the branching of the conditional statement requires special care.

In addition to generic traversal, Stratego supports data-type specific traversal by means of *congruence operators*. For each constructor c with arity n in the abstract syntax tree format, a corresponding strategy $c(s_1, ..., s_n)$ is defined that applies only to c terms, applying the s_i strategies to the corresponding sub-terms. For example, the strategy expression `If(prop-const,id,id)` applies the `prop-const` strategy only to the first argument of `If` terms. Note that `id` is the *identity* strategy that always succeeds. We can write such congruences again using the concrete syntax of the source language, where we enclose the argument strategies in `<.>`. For instance, the strategy `|[if <prop-const> then <id> else <id>]|` denotes `If(prop-const,id,id)`.

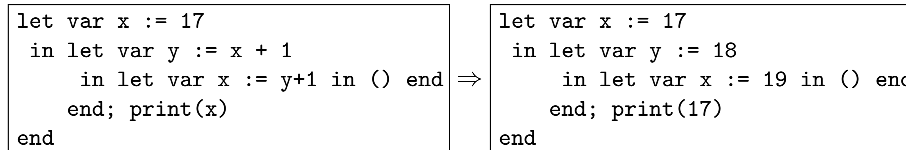
Pattern Matching While the `prop-const-if` definition **Fig. 5** uses a congruence to recognize a conditional statement, the `prop-const-let` and `prop-const-while` definitions use a *pattern match* strategy for this purpose. A pattern match `?`p matches the subject term against the pattern p, binding the meta-variables in the pattern. The construct s `=>` p is syntactic sugar for s`;` `?`p, i.e., first applying strategy s and then match the result against p. The concrete syntax congruence operators in `prop-const-assign` and `prop-const-declare` are combinations between traversal and matching; the use

of a meta-variable in a congruence denotes matching that variable. Thus, the strategy ⟦`x := <prop-const => e>`⟧ denotes `Assign(Var(?x), prop-const => e)`; it entails application of the `prop-const` strategy only to the right-hand side of an assignment and binding the result to the meta-variable `e`.

Dynamic Rules The elements we have examined so far concern the traversal order of the abstract syntax tree. The next aspect is the definition of dynamic rules for propagation of constant assignments. The `prop-const-assign` and `prop-const-declare` strategies examine the right-hand side expression `e` of an assignment and variable declaration, respectively, after these have been transformed. If the expression is a constant value, a new dynamic rule is defined with as left-hand side the variable `x` from the left-hand side of the assignment and as right-hand side the constant value `e`. Thus for an assignment ⟦ `a := 3` ⟧ the rule ⟦ `a` ⟧ `->` ⟦ `3` ⟧ is defined. In general, a dynamic rule definition `rules(` f `:` p_1 `->` p_2 `)` defines a new rule f `:` p'_1 `->` p'_2 with p'_1 and p'_2 the original patterns in which variable bindings from the context of the definition are substituted.

If the right-hand side expression is *not* a constant value, the `prop-const-assign` and `prop-const-declare` strategies *undefine* the `PropConst` rule with `x` as left-hand side. This is necessary in constant propagation since an assignment invalidates earlier assignments to the same variable. For example an assignment ⟦ `a := b + 4` ⟧ after ⟦ `a := 3` ⟧ invalidates the ⟦ `a` ⟧ `->` ⟦ `3` ⟧ rule.

Dynamic Rule Scope Dynamic rules are usually related to elements of the source program such as variables. Therefore, rules should only be applied to those parts of the tree, where they are 'in scope'. This is managed using the dynamic rule scope construct `{|` R `:` s `|}`, which limits the scope of R rules to the strategy s. That is, all `R` rules defined during the execution of s are removed when leaving the scope. This is necessary in a case such as the following:



```
let var x := 17
 in let var y := x + 1
     in let var x := y+1 in () end
    end; print(x)
end
```

⇒

```
let var x := 17
 in let var y := 18
     in let var x := 19 in () end
    end; print(17)
end
```

Without scoping the dynamic rule produced from the assignment `x := 19` in the inner scope would be used for the `print(x)` call, and produce `print(19)` instead.

In fact, not all rules defined within `s` are removed on leaving the scope. Rules can be defined relative to a named dynamic rule scope. For this purpose `prop-const-declare` *labels* the current scope with the name of the declared variable (notation: `PropConst+x`). The dynamic rule definitions by `prop-const-assign` are relative to the scope of the variable (notation: `PropConst.x`) to ensure that the rule is still visible when later scopes are exited. Therefore, the rule for `x` defined in the scope for `y` is not removed when leaving that scope.

Dynamic Rule Intersection As discussed above, when encountering a fork in the control-flow the current rule-set should be distributed over the branches and merged afterwards.

For this purpose, Stratego provides dynamic rule intersection and union operators. The intersection operator s_1 `/PropConst\` s_2 applies both strategies s_1 and s_2 to the current term in sequence, but distributes the same rule-set to both strategies. Afterwards the rule-sets are merged into one by keeping only those rules that are consistent in both sets. The union operator s_1 `\PropConst/` s_2 is similar, but keeps all rules instead. Thus, the traversal of the branches of the conditional statement is defined as

```
|[ if <id> then <prop-const> else <id> ]|
   /PropConst\ |[ if <id> then <id> else <prop-const> ]|
```

first visiting the left branch and then the right branch, keeping only the propagation rules that are valid after both branches.

The fixed-point version `/PropConst*` s of the intersection operator repeats the application of s until a stable rule-set is obtained. The transformation is applied each time using the *original term*; only the result of the last application is used to replace the term. Thus, the traversal of while statements is defined as

```
/PropConst\* |[ while <prop-const> do <prop-const> ]|
```

In fact, in the implementation of dynamic rules the rule-sets are not actually cloned. Instead, changes to the rule-set are stored in a fresh 'change-set' for each branch. These changesets are merged at the meet-point. Thus, the effort of merging two rule-sets is proportional to the number of rules in the change-sets rather than the number of rules in the rule-set.

Combining Analysis and Transformation The constant propagation strategy defined in **Fig. 5** combines analysis and transformation; the analysis of which variables are constant and the actual substitution of these constant values interact. This combination is strictly more expressive than the conventional approach of performing separate analysis and transformation phases. The application of constant folding may enable new constant propagations and the application of unreachable code elimination through the `EvalIf` and `EvalWhile` rules may discard entire sub-terms that an analysis would have to consider. This phenomenon is illustrated by the following example from [8]:

```
(x := 10;
 while A do
   if x = 10 then dosomething()
   else (dosomethingelse(); x := x + 1);
 y := x)
```

⇒

```
(x := 10;
 while A do
   dosomething();
 y := 10)
```

Since the assignment to `x` in the loop is never reached, the conditional statement can be reduced to its first branch.

3 Dependent Dynamic Rewrite Rules

While dynamic rules as presented in the previous section can be used to implement constant propagation, they cannot be used for all data-flow transformations without changes. In constant propagation a propagation rule maps a variable to a constant expression. Propagation rules are undefined when an assignment to the variable is encountered. However, in optimizations such as copy propagation and common-subexpression

elimination there are multiple variables that affect a propagation rule. We illustrate the problems using copy propagation. An assignment of a variable to a variable introduces a copy. In copy propagation these copies are replaced by their original. For example, the occurrence of `a` in the second assignment of (`a := b; c := d + a`) is replaced by the variable `b` to produce (`a := b; c := d + b`). The following dynamic rule definition for copy propagation follows naturally from the constant propagation approach:

```
copy-prop-assign = ?|[ x := y ]|;
  if <not(eq)>(x,y) then rules( CopyProp.x : |[ x ]| -> |[ y ]| )
                    else rules( CopyProp.x :- |[ x ]| ) end
```

Here we assume that the definition is embedded in a similar traversal strategy as that for constant propagation. However, it is incorrect in a number of ways.

(1) Insufficient Dependencies. The rule is not undefined when the variable in its right-hand side changes. For example, in the program (`a := b; b := foo(); c := d + a`) the variable `a` in the last statement will be replaced by `b` even though its value changed in the second statement. Thus, a `CopyProp` rule should be undefined when any of its variables is assigned.

(2) Free Variable Capture. The rule is not undefined when the local variable shadows the variable in the right-hand side. For example, in the program

```
let var a := bar() var b := baz()
 in a := b; let var b := foo() in print(a) end end
```

the occurrence of `a` in the call to `print` will be replaced with `b`, which now refers to the variable in the inner scope. Thus, a `CopyProp` rule should be undefined in a local scope when the local variable is used in the rule.

(3) Escaping Variables. The rule is not undefined when its target is going out of scope. For example, in the following program

```
let var a := bar() in let var b := foo() in a := b end; print(a) end
```

the assignment `a := b` causes the definition of a dynamic rule `a -> b`, which replaces the variable `a` in `print(a)` by `b`, which is then used outside its scope. This suggests that a `CopyProp` rule should be defined in the local scope, i.e., the scope in which the assignment lives. However, in the following variant of the program

```
let var a := bar() var c := baz()
 in let var b := foo() in a := b; a := c end; print(a) end
```

the assignment `a := c` leads to a copy propagation rules which *can* be applied in the outer scope, since neither `a` nor `c` are declared in the inner scope. Thus, a `CopyProp` rule should be defined in the *innermost* scope of the variables involved, but not necessarily the innermost scope.

This sums up the problems with the extrapolation of the use of dynamic rules for constant propagation to transformations involving variables in the right-hand sides of rules. The first two problems are solved by means of *dependent* dynamic rules, the last problem is solved by defining rules in the innermost scope of all variables involved. A correct definition of copy propagation using these techniques is presented in **Fig. 7**. Note that the traversal part of the specification is similar to the one of constant propagation and is omitted.

```
copy-prop-declare = ?|[ var x := e ]|
  ; where( new-CopyProp(|x,x) )
  ; where( try(<copy-prop-assign-aux> |[ x := e ]|) )

copy-prop-assign = ?|[ x := e ]|
  ; where( undefine-CopyProp(|x) )
  ; where( try(copy-prop-assign-aux) )

copy-prop-assign-aux = ?|[ x := y ]|
  ; where( <not(eq)>(x,y) )
  ; where( innermost-scope-CopyProp => z )
  ; rules( CopyProp.z : |[ x ]| -> |[ y ]| depends on [(x,x), (y,y)] )

innermost-scope-CopyProp =
  get-var-names => vars; innermost-scope-CopyProp(elem-of(|vars))
```

Fig. 7. Specification of copy propagation with dependent dynamic rules.

A *dependent dynamic rule* is a dynamic rule that declares its dependencies on program entities such as variables. The `depends on` clause of a dependent rule declares a list of pairs of the scope and value of the dependencies. For example, a copy propagation rule `|[ a ]| -> |[ b ]|` depends on the object variables `a` and `b`, entailing the dependency list `[(a,a),(b,b)]`. In the case of the Tiger transformations in this paper, variable names are used as scope labels and as dependencies. However, this is not necessarily the case in general, which motivates the distinction. Rule dependencies are used to undefine or shadow a dynamic rule when one of its dependencies is changed. For example, if the object variable `b` is assigned to, all copy propagation rules in which that variable is involved become invalid. For this purpose, a mapping from dependencies to the rules they affect is maintained. For a dependent dynamic rule *R*, the strategies `undefine-`*R*, `new-`*R*, and `innermost-scope-`*R* solve the problems discussed above.

(1) The `undefine-`*R*`(|`*dep*`)` strategy undefines all rules depending on *dep*. It should be used when the meaning of *dep* has changed, e.g. in `copy-prop-assign`.

(2) The `new-`*R*`(|`*l*`,`*dep*`)` strategy labels the current scope with *l* and *locally* undefines any rules that depend on *dep*. This strategy is typically used when encountering a local declaration for dependency *dep* with scope label *l*, e.g., in `copy-prop-declare`, and avoids rules depending on `dep` living in external scopes from being applied, which would result in free variable capture.

(3) The `innermost-scope-`*R*`(`*s*`)` strategy examines the labels in the scopes for *R* starting with the most recent one, producing the first for which *s* succeeds. This is used in the definition of `innermost-scope-CopyProp` to obtain the innermost scope label for the set of variables in an expression. Thus, in `copy-prop-assign-aux`, new `CopyProp` rules are defined in the innermost scope *z*, which is the innermost scope of *x* and *y*. This ensures that the rule is removed as soon as one of its dependencies goes out of scope. As a consequence rules are only applied to those parts of the tree where both variables are in scope, avoiding variables to escape from their scope.

Dependent dynamic rules are a generative extension of basic dynamic rules. Thus, the effect of dependent dynamic rules can be achieved using only basic dynamic rules,

but the implementation of the administration of dependencies and their mapping is rather tedious. The language feature supports the reuse of this code pattern by means of a code generator in the compiler, which can also exploit the internal representation of dynamic rules.

4 Generic Data-flow Transformation Strategies

The definition of copy propagation in **Fig. 7** is very similar to the definition of constant propagation in **Fig. 5**. The difference between the two transformations is restricted to the optimization specific strategies for handling declarations and assignments. Control flow constructs for forking and iteration share a common strategy with the dynamic rule name as only difference. The generic forward propagation strategy for Tiger (`forward-prop`) in **Fig. 9** allows individual optimizations to focus on their essential elements by reusing the code for the common parts of the transformation. A dual strategy for backwards propagation is defined in similar fashion [14].

The `forward-prop` strategy is parameterized with strategies that are applied at certain stages of the transformation of a language construct. The strategies `transform`, `before` and `after` are local rewrites of a construct and can be used to tune the transformation. Further parameters are the names of rules to be intersected (*Rs1*) and unified (*Rs2*) at fork and join points, and rule names (*Rs3*) that are part of the transformation, but do not require a dynamic rule operation at confluence points.

Common-Subexpresson Elimination **Fig. 10** presents an instantiation of `forward-prop` for common-subexpression elimination (CSE). CSE is a transformation that replaces common expressions with a variable that already contains the value of the expression. For example, CSE transforms `(a := b + c; d := b + c)` to `(a := b + c; d := a)`. By instantiating `forward-prop`, we can focus on the definition of the conditions that enable the propagation of non-trivial expressions by defining `CSE` rules. Scoping and undefining of dynamic rules are handled in the `forward-prop` strategy. This is a major simplification of the implementation of CSE, since we do not have to handle all the control-flow constructs separately in this specific optimization.

Combining Transformations The `forward-prop` strategy uses generalized versions of the dynamic rule combinators to deal with multiple rules. The `new-dynamic-rules` and `undefine-dynamic-rules` strategies apply the `new-`*R* and `undefine-`*R* rules for all parameter rules. Similarly, the `/`*Rs1*`\`*Rs2*`/` and `/`*Rs1*`\`*Rs2*`/*` operators generalize the intersection and union operators to a single combined operator, which performs intersection over the first set of rules and union over the second. Thus, the generic

```
super-opt =
  forward-prop(prop-const-transform, bvr-before,
    bvr-after; copy-prop-after; prop-const-after; cse-after
    | ["PropConst", "CopyProp", "CSE"], [], ["RenameVar"])
```

Fig. 8. ‘Super’ transformation combining constant propagation, copy propagation, common-subexpression elimination, and bound variable renaming.

```
forward-prop(transform, before, after | Rs1, Rs2, Rs3) =
<conc>(Rs1, Rs2, Rs3) => RsSc; <conc>(Rs1, Rs2) => RsDf;
let
  fp = prop-assign <+ prop-declare <+ prop-let <+ prop-if <+ prop-while
       <+ transform(fp) <+ (before; all(fp); after)

  prop-assign =
    |[ <id> := <fp> ]|
    ; (transform(fp)
       <+ before; ?|[ x := e ]|; undefine-dynamic-rules(|RsDf,x); after)

  prop-declare =
    |[ var <id> := <fp> ]|
    ; (transform(fp)
       <+ before; ?|[ var x := e ]|; new-dynamic-rules(|RsSc,x,x);after)

  prop-let =
    ?|[ let d* in e* end ]|
    ; (transform(fp) <+ {|~RsSc : before; all(fp); after |})

  prop-if =
    |[ if <fp> then <id> else <id> ]|
    ; (transform(fp)
       <+ before ; (|[ if <id> then <fp> else <id> ]| /~Rs1\~Rs2/
                    |[ if <id> then <id> else <fp> ]|); after)
  prop-while =
    ?|[ while e1 do e2 ]|
    ; (transform(fp)
       <+ before; /~Rs1\~Rs2/* |[ while <fp> do <fp> ]|; after)
 in fp
end
```

Fig. 9. A generic strategy for forward propagation transformations.

```
cse = forward-prop(cse-transform, id, cse-after | ["CSE"], [], [])

cse-transform(recur) = fail

cse-after = try(cse-assign <+ cse-declare <+ CSE)

cse-declare = ?|[ var x := e ]|; where( <cse-assign> |[ x := e ]| )

cse-assign = ?|[ x := e ]|
  ; where( <pure-and-not-trivial(|x)> |[ e ]| )
  ; where( get-var-dependencies => xs )
  ; where( innermost-scope-CSE => z )
  ; rules( CSE.z : |[ e ]| -> |[ x ]| depends on xs )
```

Fig. 10. Common-subexpression elimination using generic forward propagation strategy.

forward propagation strategy can apply different analyses and transformations at the same time by combining elements from several one issue transformations. As an example, **Fig. 8** shows a strategy that combines constant propagation, copy propagation, common-subexpression elimination, unreachable code elimination and bound-variable renaming. We have included bound-variable renaming on the fly in this combined transformation to avoid dynamic rules from being unnecessarily undefined/shadowed.

5 Discussion

Previous Work Scoped dynamic rules were introduced in [17] to overcome the limitations of the context-free nature of static rewrite rules with applications to bound variable renaming, function inlining, and dead code elimination. A first version of constant propagation based on that design is described in [13]. Scoped dynamic rules have been extended, improved, and formalized in [4], introducing labeling of scopes to provide more fine-grained control over the definition and removal of dynamic rules, and introducing the fork, intersection, union and fixed-point operations on sets of dynamic rules. The contributions of this paper with respect to that work are the introduction of dependent dynamic rules, the definition of generic data-flow transformation strategies, and the combination of data-flow transformations. In the technical report version of this paper [14] we also present a generic backwards propagation strategy, the other instantiations of the generic forward propagation strategy used in the combined optimizer and a specification of partial redundancy elimination, illustrating how two separate analyses (backwards and forwards) can communicate via annotations.

Related Work A discussion of techniques for data-flow transformations is beyond the scope of this paper. Rather, we focus on languages and tools that automate part of the effort of producing program data-flow transformations.

Program analyzer generators such as Sharlit [16] and PAG [11] produce analyzers from a specification of the flow values and flow functions for the problem at hand. In Sharlit [16] these have to be implemented in C++ following the conventions of the tool. PAG provides a dedicated domain-specific language for all aspects of the specification. These tools do not support combined super-analyses, nor the specification of transformations; applications of analysis and transformation are alternated.

Graph transformation tools such as OPTIMIX [3] and the tools of De Moor et al. [7, 6] provide a transformation-oriented approach, aiming at declarative specification of individual transformations, in contrast to the global approach of data-flow analyses. An OPTIMIX program consists of a set of rewrite rules on a graph representation of a program. The graph can be extended with additional edges to express analysis results. Transformation is by exhaustive application of rules. Lacey and De Moor [7] use graph rewrite rules with temporal logic conditions to check properties of the control-flow graph; that is, enabling conditions are checked from the point of view of the node that is transformed, rather than as a global analysis. Path logic programming [6] is a variation on this approach using path patterns, regular expressions over paths through the control-flow graph of a program that express the properties that should hold on all or some paths to the node subject to transformation. The drawback of these approaches is that

pattern matching requires performing a global program analysis and a search for graph nodes that match a certain pattern. After applying a transformation, the analysis needs to be redone. Obtaining efficient optimizers requires *incrementally* updating the analysis information after applying transformations. There is some progress in this area [15] with a technique for compositional analysis based on path expressions. Our approach provides effective procedures for finding data-flow redices in abstract syntax trees.

Combination of analysis and transformation is not only desirable from the point of view of performance, but can also produce better results. Wegman and Zadek introduced conditional constant propagation, a combination of constant propagation and unreachable code elimination [21], which produces better results than applying the two transformations in sequence. Click and Cooper [5] formally defined in which cases integrating two data-flow analyses results in better results than a sequential application of the individual analyses, and they combined constant propagation, unreachable code elimination and value numbering. Rather than implementing such combined transformations in dedicated algorithms, we provide high-level constructs for the composition of such combined transformations. In this sense our work is most related to that of Lerner et al. who have developed a series of frameworks [8–10] for the composition of data-flow transformations in a modular way. Similarly to our approach they combine analysis and the application of transformations as long as they share the same direction. There is a difference in perspective, though; while we model program analysis by dynamic transformation rules, they let the analysis framework *simulate* transformations. Another difference is that their frameworks operate on fixed control-flow graph representations. In contrast, Stratego is not specifically designed for data-flow transformations. Rewrite rules, strategy combinators, and dynamic rules are useful in a wide variety of transformations. In addition, our approach handles variable bindings correctly.

Conclusion We have presented a language for the concise specification of source-to-source data-flow transformations. The generic high-level constructs allow adaptation of the approach to other programming languages with little effort; we have used the approach to implement optimizations in a compiler for the Octave language. Transfer functions are elegantly captured by dynamic rewrite rules and confluence operators for intersection or fixed-point applications are used to specify program analysis and transformation. The language supports combination of analysis and transformation in one traversal and the combination of multiple transformations in the same traversal.

The techniques presented in this paper are supported by Stratego/XT 0.14, which is available from `http://www.stratego-language.org/`.

Acknowledgments We thank Martin Bravenboer for his help with the preparation of this paper, Tom de Vries and the anonymous referees for their comments on a previous version of this paper, and Oege de Moor and Ganesh Sittampalam for the discussions of specification of optimizers.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. A. Appel. *Modern compiler implementation in ML.* Cambridge University Press, 1998.
3. U. Assmann. How To Uniformly Specify Program Analysis and Transformation. In T. Gyimóthy, editor, *Internationational Conference on Compiler Construction (CC'96)*, volume 1060 of *LNCS*, pages 121–135, Linköping, Sweden, 1996. Springer.
4. M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. Technical Report UU-CS-2005-005, Institute of Information and Computing Sciences, Utrecht University, 2005.
5. C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.
6. S. Drape, 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)*, pages 133–144, Pittsburgh, Pensylvania, USA, October 2002. ACM.
7. D. Lacey and O. de Moor. Imperative program transformation by rewriting. In R. Wilhelm, editor, *Proceedings of the 10th International Conference on Compiler Construction*, volume 2027 of *LNCS*, pages 52–68. Springer Verlag, 2001.
8. S. Lerner, D. Grove, and C. Chambers. Combining dataflow analyses and transformations. In *SIGPLAN Symposium on Principles of Programming Languages (POPL'02)*, pages 270–282, Portland, Oregon, January 2002.
9. S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Programming Language Design and Implementation (PLDI'03)*, pages 220 – 231. ACM SIGPLAN, June 2003.
10. S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Principles of Programming Languages (POPL'05)*, pages 364–377. ACM SIGPLAN, January 2005.
11. F. Martin. PAG an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer STTT*, 2(1):46–67, November 1998.
12. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.
13. K. Olmos and E. Visser. Strategies for source-to-source constant propagation. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies (WRS'02)*, volume 70 of *ENTCS*, page 20, Copenhagen, Denmark, July 2002. Elsevier Science Publishers.
14. K. Olmos and E. Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. Technical Report UU-CS-2005-006, Institute of Information and Computing Sciences, Utrecht University, 2005.
15. G. Sittampalam, O. de Moor, and K. F. Larsen. Incremental execution of transformation specifications. In *SIGPLAN Symposium on Principles of Programming Languages (POPL'04)*, pages 26–38. ACM, January 2004.
16. S. W. K. Tjiang and J. L. Hennessy. Sharlit—A tool for building optimizers. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, July 1992.
17. E. Visser. Scoped dynamic rewrite rules. In M. van den Brand and R. Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *ENTCS*. Elsevier Science Publishers, September 2001.
18. E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LNCS*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

19. E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 216–238. Spinger-Verlag, June 2004.
20. E. Visser, 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)*, pages 13–26. ACM Press, September 1998.
21. M. Wegman and F. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13:181–210, April 1991.

A Forward Propagation: Complete Specifications

This appendix provides complete specifications of forward propagating data-flow transformations.

A.1 Generic Forward Propagation

Follows the specification of the generic forward strategy for a large subset of Tiger. It includes uninitialization of variable declarations, `for` statements, function definitions and function calls.

```
  forward-prop(transform, before, recur, after, effects | Rs1 , Rs2 , Rs3) =
    let fp(transform, before, recur, after, effects | Rs1 , Rs2 , RsSc , RsDF) =
           ( (  forward-prop-let    (transform, before, recur, after | RsSc)
             <+ forward-prop-for    (transform, before, recur, after | Rs1,Rs2,RsSc)
             <+ forward-prop-vardec (transform, before, recur, after | RsSc)
             <+ forward-prop-funcdec(transform, before, recur, after | RsSc)
             <+ forward-prop-assign (transform, before, recur, after | RsDF)
             <+ forward-prop-if     (transform, before, recur, after | Rs1 , Rs2)
             <+ forward-prop-while  (transform, before, recur, after | Rs1 , Rs2)
             <+ forward-prop-call   (transform, before, recur, after,effects | RsSc)
             )
            <+ transform <+ before; all(recur); after
          )
     in fp(transform, try(before), recur, try(after), effects
          | Rs1 , Rs2 , <conc>(Rs1 , Rs2 , Rs3), <conc>(Rs1 , Rs2))
   end

strategies // variables

  forward-prop-let(transform, before, recur, after | Rs) =
    ? |[ let d* in e* end ]|
    ; (transform <+ {| ~Rs  : before; all(recur); after |})

  forward-prop-vardec(transform, before, recur, after | Rs) =
    |[ var <id> <id> := <recur> ]|
    ; (transform
       <+ before
          ; try(?|[ var x  ta  := e ]|; where( <new-dynamic-rules(|Rs , x)>x ))
          ; after
       )

  forward-prop-vardec(transform, before, recur, after | Rs) =
    ? |[ var x  ta]|
    ; (transform
        <+ before
           ; try(?|[ var x  ta ]|; where( <new-dynamic-rules(|Rs , x)>x ))
           ; after
      )
```

```
  forward-prop-assign(transform, before, recur, after | Rs) =
    |[ <id> := <recur> ]|
    ; (transform
        <+ before
           ; try( ?|[ x := e ]|; where( <undefine-dynamic-rules(|Rs)>x ) )
           ; after
      )

strategies // control-flow

  forward-prop-if(transform, before, recur, after | Rs1, Rs2) =
    |[ if <recur> then <id> ]|
    ; (transform
       <+ before
          ; (|[ if <id> then <recur> ]| /~Rs1\~Rs2/ id)
          ; after)

  forward-prop-if(transform, before, recur, after | Rs1, Rs2) =
    |[ if <recur> then <id> else <id> ]|
    ; (transform
       <+ before
          ; (|[ if <id> then <recur> else <id> ]|
                /~Rs1\~Rs2/  |[ if <id> then <id> else <recur> ]|)
          ; after)

  forward-prop-while(transform, before, recur, after | Rs1, Rs2) =
    |[ while <id> do <id> ]|
    ; (transform
       <+ before
          ; (/~Rs1\~Rs2/* |[ while <recur> do <recur> ]|)
          ; after)

  forward-prop-for(transform, before, recur, after | Rs1,Rs2,Rs) =
      |[ for <id:id> := <recur> to <recur> do <id> ]|
    ; (transform
       <+ {| ~Rs1:
               before
             ; ?|[for x:= e to e1 do e3 ]|
             ; where(<new-dynamic-rules(|Rs1, x)>x)
             ; (/~Rs1\~Rs2/* |[ for <id:id> := <id> to <id> do <recur> ]|)
             ; after
          |}
      )

 forward-prop-funcdec (transform, before, recur, after | RsSc) =
  |[ function <id>(<*id>) <id> = <id> ]|
  ; (transform
     <+ dr-ignore-rules-state(fp-function(transform, before, recur, after)|RsSc)
    )
```

```
fp-function(transform, before, recur, after) =
 |[ function <?f>(<*id>) <id> = <id> ]|
 ; where(free-vars;?vs ; rules(ForwardProp: |[ f(a*) ]| -> vs))
 ; |[ function <id>(<*id>) <id> = <recur> ]|

forward-prop-call (transform, before, recur, after, effects | Rs) =
   |[ <id>(<*map(try(recur))>) ]|
   ; (transform
      <+ before
         ; ?|[ f(a*) ]|
         ; where(try(ForwardProp; effects) )
         ; after
     )
```

A.2 Bound Variable Renaming

Bound-variable renaming is a transformation that renames variables in order to avoid free variable capture during transformation. The transformation only renames variables that are already used in an outer scope.

```
bvr = forward-prop(fail, bvr-before, bvr, bvr-after, id| [],[],["RenameVar"])

bvr-before = try(bvr-declare <+ bvr-assign)

bvr-after = try(RenameVar)

bvr-declare : |[ var x ta := e ]| -> |[ var y ta := e ]|
  where <rename-variable> x => y

bvr-declare : |[ var x ta ]| -> |[ var y ta ]|
  where <rename-variable> x => y

rename-variable : x -> y
  where innermost-scope-RenameVar(?x)
      ; <newname> x => y
      ; rules( RenameVar : |[ x ]| -> |[ y ]| )

bvr-assign : |[ x := e ]| -> |[ y := e ]|
  where <RenameVar> |[ x ]| => |[ y ]|
```

A.3 Constant Propagation

```
prop-const = forward-prop( prop-const-transform(prop-const)
             , id
             , prop-const
             , prop-const-after
             , prop-const-function-effects
             |  ["PropConst"], [], [] )
```

```
prop-const-transform(recur) =
  EvalFor <+ EvalIf; recur <+ |[ while <recur> do <id> ]|; EvalWhile; recur

prop-const-after =
  PropConst + prop-const-assign <+ prop-const-declare
  <+ EvalBinOp <+ EvalRelOp <+ EvalString

prop-const-declare =
  ?|[ var x ta := e ]|; where( <prop-const-assign> |[ x := e ]| )

prop-const-assign =
  ?|[ x := e ]|
  ; where( <is-value> e )
  ; rules( PropConst.x : |[ x ]| -> |[ e ]| depends on [(x,x)] )

prop-const-function-effects =
  map({x: ?|[ x ]|; undefine-dynamic-rules(|["PropConst"], x)})
```

A.4 Copy Propagation

```
copy-prop = forward-prop(fail, id
              , copy-prop
              , copy-prop-after
              , prop-const-function-effects
              |["CopyProp"],[],[] )

copy-prop-after =
  copy-prop-assign <+ copy-prop-declare <+ repeat1(CopyProp)

copy-prop-declare =
  ? |[ var x ta := e ]|
  ; where( try(<copy-prop-assign> |[ x := e ]|) )

copy-prop-assign =
  ? |[ x := y ]|
  ; where( <not(eq)>(x,y) )
  ; where( get-var-dependencies => xs )
  ; where( innermost-scope-CopyProp => z )
  ; rules( CopyProp.z : |[ x ]| -> |[ y ]| depends on xs )

prop-const-function-effects =
  map({x: ?|[ x ]|; undefine-dynamic-rules(|["CopyProp"], x)})

innermost-scope-CopyProp =
  get-var-names => vars
  ; innermost-scope-CopyProp(elem-of(|vars))
```

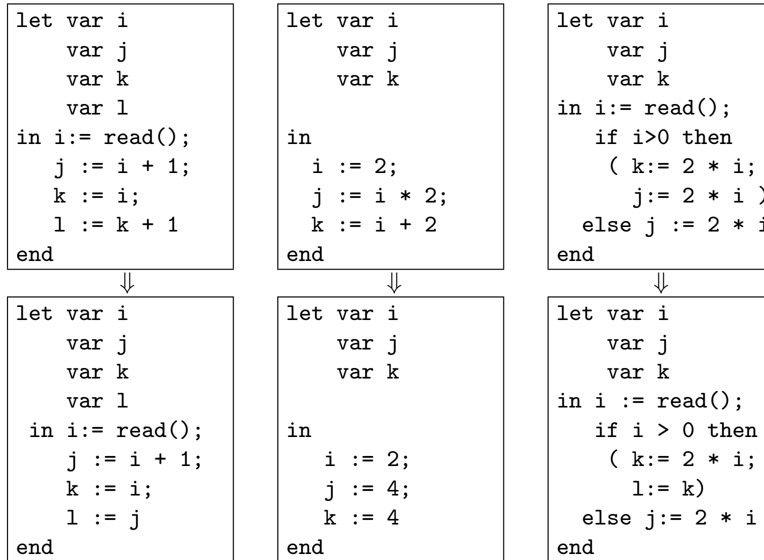


Fig. 11. Example of 'Super' transformation for constant propagation, copy propagation and common sub-expression elimination

A.5 SuperOpt

SuperOpt is a combined transformation of bound variable renaming, constant propagation, copy propagation and common subexpression elimination. Three different code fragments are shown in **Fig. 11** that had tranformed with this super-optimization. If only one of these transformations would have been applied, it would have benefit only one code fragment. For instance, if constant propagation will be applied only the middle fragment would have been optimized. An alternative would have been to apply three optimizations in sequence requiring three different traversals.

It follows the specification of SuperOpt as an instantiation of the generic propagation framework.

```
super-opt =
  forward-prop( prop-const-transform(super-opt)
              , super-opt-before(super-opt)
              , super-opt
              , super-opt-after
              , prop-const-function-effects
              | ["PropConst","CopyProp","CSE"], [], ["RenameVar"]
  )

super-opt-before(recur) =
 try(bvr-before(recur)); try(prop-const-before)

super-opt-after =
```

```
   try(bvr-after); try(copy-prop-after); try(prop-const-after); try(cse-after)

prop-const-function-effects =
  map({x: ?|[x]|; undefine-dynamic-rules(|["PropConst","CopyProp","CSE"], x)})
```

B Backwards Propagation

Generic (and transformation independent strategy) for backward propagation transformations.

B.1 Generic Strategy

```
  backward-prop(transform, before, recur, after, effects | Rs1, Rs2, Rs3) =
    let bp(transform, before, recur, after, effects | Rs1, Rs2, RsSc, RsDF) =
        (( backward-prop-seq       (transform, before, recur, after)
           <+ backward-prop-let    (transform, before, recur, after | RsSc)
           <+ backward-prop-vardec (transform, before, recur, after | RsSc)
           <+ backward-prop-assign (transform, before, recur, after | RsDF)
           <+ backward-prop-if     (transform, before, recur, after | Rs1, Rs2)
           <+ backward-prop-while  (transform, before, recur, after | Rs1, Rs2)
           <+ backward-prop-for    (transform, before, recur, after | Rs1, Rs2, RsSc)
           <+ backward-prop-call   (transform, before, recur, after, effects | RsSc)
         )
          <+ transform <+ before; all(recur); after
         )
     in bp(transform, try(before), recur, try(after), effects
          | Rs1, Rs2, <conc>(Rs1, Rs2, Rs3), <conc>(Rs1, Rs2))
    end

strategies // variables

  backward-prop-let(transform, before, recur, after | Rs) =
    ? |[ let d* in e* end ]|
    ; (transform
       <+ {| ~Rs
             : |[ let <*decls-matter(recur|Rs)> in <*id> end ]|
             ; before
             ; |[ let <*id> in <*reverse-map(recur)> end ]|
             ; after
          |}
      )

  decls-matter(recur|Rs) =
    where(map(try(init-scopes(|Rs))))

  init-scopes(|Rs) =
    (?|[ var x ta ]| <+  ?|[ var x ta := e ]| <+ ?FArg|[ x ta ]|)
    ; new-dynamic-rules(|Rs, x, x)

  init-scopes(|Rs) =
    ?|[<fundecs: map(init-func-scopes(|Rs))>]|

  init-func-scopes(|Rs) =
    |[ function <?f>(<*id>) <id> = <id> ]|
```

```
    ; new-dynamic-rules(|Rs, f, f)
    ; where(free-vars; map(!Var(<id>)); ?vs; rules(BackwardProp: |[ f(a*) ]| -> vs))

  backward-prop-seq(transform, before, recur, after) =
    ? |[ ( e* ) ]|
    ; before
    ; |[ ( <*reverse-map(recur)> ) ]|
    ; after

  backward-prop-vardec(transform, before, recur, after | Rs) =
    |[ var <id> <id> := <id> ]|
    ; (transform <+ before; |[ var <id> <id> := <recur> ]|; after)

  backward-prop-vardec(transform, before, recur, after | Rs) =
    ? |[ var x ta]|
    ; (transform <+ before; after)

  backward-prop-assign(transform, before, recur, after | Rs) =
    |[ <?|[ x ]|> := <id> ]|
    ; (transform
       <+ before
         ; undefine-dynamic-rules(|Rs, x)
         ; |[ <id> := <recur> ]|
         ; after
       )

strategies // control-flow

  backward-prop-if(transform, before, recur, after | Rs1, Rs2) =
    |[ if <id> then <id> ]|
    ; (transform
       <+ before
         ; (|[ if <id> then <recur> ]| /~Rs1\~Rs2/ id)
         ; |[ if <recur> then <id> ]|
         ; after)

  backward-prop-if(transform, before, recur, after | Rs1, Rs2) =
    |[ if <id> then <id> else <id> ]|
    ; (transform
       <+ before
         ; (|[ if <id> then <recur> else <id> ]|
             /~Rs1\~Rs2/  |[ if <id> then <id> else <recur> ]|)
         ; |[ if <recur> then <id> else <id> ]|
         ; after)


  backward-prop-while(transform, before, recur, after | Rs1, Rs2) =
    |[ while <id> do <id> ]|
    ; (transform
       <+ before
```

```
        ; (/~Rs1\~Rs2/* |[ while <recur> do <recur> ]|)
        ; after)

  backward-prop-for(transform, before, recur, after | Rs1, Rs2, Rs) =
     |[ for <?x> := <recur> to <recur> do <id> ]|
    ; (transform
       <+ {| ~Rs
            : before
            ; new-dynamic-rules(|Rs, x, x)
            ; (id /~Rs1\~Rs2/ |[ for <id> := <id> to <id> do <recur> ]|)
            ;  |[ for <id> := <recur> to <recur> do <id> ]|
            ; after
           |}
      )

 backward-prop-call (transform, before, recur, after,effects | Rs) =
  |[ <id>(<*id>) ]|
  ; (transform
       <+ before
          ; |[<id>(<*map(recur)>) ]|
          ; where(try(BackwardProp; effects))
          ; after
     )
```

B.2 Dead Code Elimination

```
  dead-code-elim =
    backward-prop(dce-transform
                  , dead-code-before
                  , dead-code-elim
                  , dead-code-after(dead-code-elim)
                  , dce-function-effects
                  | [], ["Needed", "Used"], []
    )

  dce-transform =
    VarNeeded <+ transform-assign

  dead-code-before =
    try(DeclareNotNeeded)

  dead-code-after(dce) =
        ElimIf <+ ElimIfThen
        <+ ElimFor <+ ElimWhile
        <+ |[ (<*reverse-filter(not(?|[ () ]|))>) ]|
        <+ dce-let-after(dce)
        <+ dce-call-after(dce)
        <+ dce-assign-after(dce)
```

```
 dce-function-effects =
   map(VarNeeded)


 dce-call-after(dce) =
  ?|[ f(a*) ]|
  ; try(rules(Used.f : |[ f ]| -> |[ f ]| depends on [(f,f)] ))

 dce-let-after(dce) =
  |[ let <*reverse-filter(dce-decls-after(dce))>
      in <*reverse-filter(where(not(|[ () ]|)))> end ]|

 dce-decls-after(dce) =
   |[ <typedecs: id> ]|

 dce-decls-after(dce) =
   ?|[ var x ta := e ]|
   ; where(<Used> |[ x ]|)
   ; |[var <id> <id> := <dce> ]|

 dce-decls-after(dce) =
   ?|[ var x ta ]|
   ; where(<Used> |[ x ]|)

 // Multiple recursive functions may call each other
 dce-decls-after(dce) =
  |[ <fd*: reverse-filter(dce-elim-function(dce))>]|

 dce-elim-function(dce) =
   ?|[ function f(x*) ta = e ]|
   ; where(<Used>|[ f ]|)
   ; where(!|[f()]|; try(BackwardProp; map({?x; <VarNeeded>|[ x ]|})  ))
   ;     |[ function <id:id>(<*map(init-scopes(|["Needed","Used"])
              ; DeclareNotNeeded)>) <id> = <dce> ]|

transform-assign =
   ?|[ x := e ]|
   ; not(<Needed> |[ x ]| <+ <oncetd(?|[f(a*)]|)> |[ e ]|)
   ; !|[ (<*<collect-statements> |[ e ]|>) ]|
   ; dead-code-elim

 dce-assign-after(dce) =
   ?|[ x := e ]|
   ; rules(Used.x : |[ x ]|  -> |[ x ]| depends on [(x,x)])

 DeclareNotNeeded =
   |[ <fundecs: map({?|[ function f(x*) ta = e ]|; NotNeeded(|f)})> ]|

 DeclareNotNeeded =
  ( ?|[ var x ta := e ]| <+ ?|[ var x ta ]| <+ ?FArg|[ x ta ]|)
```

```
 ; NotNeeded(|x)


NotNeeded(|x) =
  rules(
    Needed.x :- |[ x ]|
    Used.x   :- |[ x ]|
  )

VarNeeded =
  ?|[ x ]|
  ; rules(
      Needed.x : |[ x ]| -> |[ x ]| depends on [(x,x)]
      Used.x   : |[ x ]| -> |[ x ]| depends on [(x,x)]
    )
```

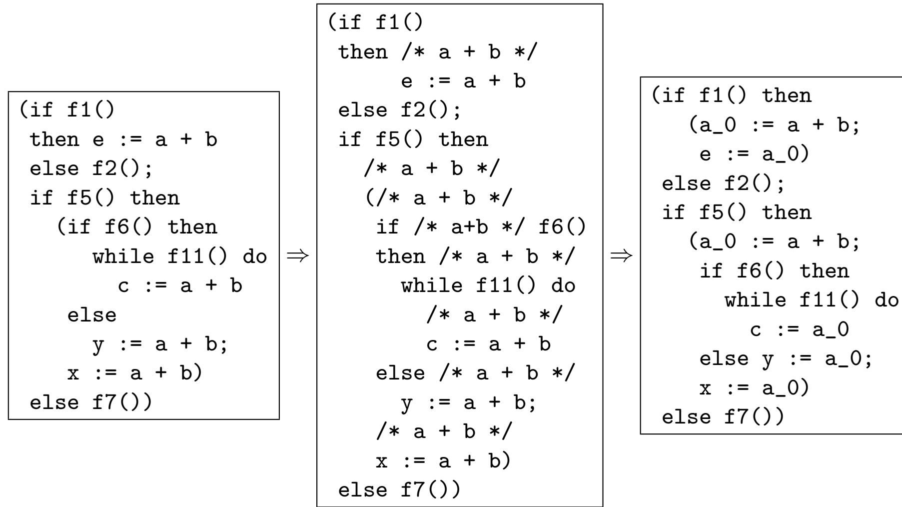


Fig. 12. Application of Safe-Earliest partial redundancy elimination. The annotations in the intermediate step label the points at which an expression is down-safe.

C Safe-Earliest Partial Redundancy Elimination

Partial redundancy elimination is an optimization that attempts to eliminate repeated computations along program paths. Such repeated computation are marked as redundant and can be replaced by use of previously computed expressions. If a computation is redundant along a certain path, it may still be needed in another execution path. Finding redundant computations and placing them at program points where the occurrence is minimized, is the goal of this optimization.

This transformation requires a complex analysis to determine program points where new computations will be lifted. This transformation cannot be achieved by means of a single traversal in which data-flow transformation is propagated. Lazy Code Motion [**?**,**?**] is an algorithm to compute this transformation. It requires the computation of the following predicates: down-safe, earliest, delay, latest, and isolated. **Fig. 12** shows an example of the first two pahses of this transformation namely down-safe and earliest.

To show that the Stratego approach can be used to implement transformations that require coordination between multiple analyses, we present the implementation of the Safe-Earliest Partial Redundancy Elimination transformation, which corresponds to the first two steps of the Lazy Code Motion algorithm. These two phases of the algorithm result in programs which are computationally optimal (redundant computations are performed the least possible number of times), but not lifetime optimal, since computations are lifted to the earliest point in the program.

The Safe-Earliest transformation is implemented using two separate traversals `down-safe`, a backwards traversal, and `earliest`, a forward traversal. The complete transformation is simply the sequential composition `down-safe; earliest`.

C.1 Down-Safe

Down-safe or anticipatebility of expressions is computed following a backwards flow. Down-safe is a property of an expression along program path sections. At program points, several expressions can be down-safe, and we have to keep track of them. Term annotations will contain down-safe expressions at statement level. Stratego term annotations are terms attached to tree constructs; they are denoted by `t{t'}`.

The specification of `down-safe` is defined by instantiation of the generic backwards propagation strategy. The backward propagation defines for each non-trivial and pure expression `e` a dynamic rule `DownSafe`, which serves as a predicate for the down-safety of `e` at the current program point. The rule `down-safe-annotate` includes down-safe expressions as annotations of the current term. The `DownSafe` rule depends on the variables in `e`, which means that it is undefined as soon as one of those variables is assigned. The rule uses the intersection operation in control-flow constructs, i.e an expression is down-safe only if it is down-safe along all paths.

The predicate is used to *annotate* statements with the list of expressions that is down-safe *before* that statement. For this purpose, the transformation keeps track of *all* expressions that are being considered for lifting using the `AllDownSafe` rule. The rule `down-safe-annotate` takes the list of all these expressions and filters the ones that are down-safe at the current program point.

```
down-safe = backward-prop(fail
                         , down-safe-before
                         , down-safe
                         , down-safe-after
                         , down-safe-function-effects
                         | ["DownSafe"], [], [] )

down-safe-before = try(down-safe-exp)

down-safe-after = try(is-statement; down-safe-annotate)

down-safe-exp =
  ?|[ e ]|
  ; where( not(trivial); pure )
  ; where( get-var-names => xs )
  ; where( get-var-dependencies => deps )
  ; where( innermost-scope-DownSafe(elem-of(|xs)) => z )
  ; rules( DownSafe.z : |[ e ]| -> |[ e ]| depends on deps )
  ; include-AllDownSafe(|e)

down-safe-annotate :
  e -> e{e*}
  where <bagof-AllDownSafe>(); filter(where(DownSafe)) => e*

include-AllDownSafe(|e) =
  if <not(AllDownSafeSet)> e
  then rules( AllDownSafeSet : e  AllDownSafe :+ () -> e )
  end
```

```
down-safe-function-effects =
  map({x: ?|[ x ]|; undefine-dynamic-rules(|["DownSafe"], x)}
```

C.2 Earliest

The next step is to place assignments of the lifted expressions to temporary variables at the earliest possible point in the program. This transformation uses the annotations created by the `down-safe` transformation.

Basically, this can be done by a forward propagation in which the first point that has an expression in its annotation can be extended with an assignment for the expression to a temporary variable. All subsequent annotations should be ignored and all uses of the expression should be replaced with the temporary variable that contains the computation.

The specification of `earliest` defines this transformation using a forward propagation with the strategy `earliest-transform` to replace annotated statements in two steps using three dynamic rules.

First, `earliest-filter-annotations` reduces the annotations of a statement to the annotations that are actually earliest at that point. Those expressions are then declared to be `NotEarliest`, such that occurrences of the expression in subsequent statements will be removed.

Next, `assign-earliest-to-var` turns expressions in annotations into assignments of the expressions to temporary variables. The dynamic rule `ReplaceExp` replaces a lifted expression with the corresponding temporary variable. The rule is only propagated if consistent on all paths, i.e. definitions of `ReplaceExp` for the same expression in different branches should use the same temporary variable. Otherwise, it cannot be replaced after the branch. In order to use the same temporary variable for all placements and replacements of the same expression, the dynamic rule `TempVar` maps expressions to temporary variables. The rule `new-temp` creates a new temporary variable if one does not yet exist for an expression.

```
earliest =
  forward-prop(earliest-transform
              , id
              , earliest
              , earliest-after
              , earliest-function-effects
              | ["NotEarliest","ReplaceExp"], [], ["TempVar"]
  )

earliest-transform =
  earliest-filter-annotations
  ; try(assign-earliest-to-var; |[ (<id>; <earliest>) ]| <+ earliest)

earliest-after = try(ReplaceExp)
```

```
earliest-filter-annotations :
  e{e1, e1*} -> e{e2*}
  where <filter(not(NotEarliest))> [e1 | e1*] => e2*
      ; map({e', deps: ?e'; get-var-dependencies => deps;
             rules( NotEarliest : e' -> e' depends on deps )})

assign-earliest-to-var :
  e{e1, e1*} -> |[ ((e2*); ~e{}) ]|
  where <map(assign-to-var)> [e1 | e1*] => e2*

assign-to-var :
  |[ e ]| -> |[ x := e ]|
  where get-var-dependencies => deps
      ; <TempVar <+ new-temp> |[ e ]| => x
      ; rules( ReplaceExp : |[ e ]| -> |[ x ]| depends on deps )

new-temp :
  |[ e ]| -> x  where new => x; rules( TempVar : |[ e ]| -> x )

earliest-function-effects =
  map({x: ?|[ x ]|; undefine-dynamic-rules(|["Earliest","ReplaceExp"], x)}
```