An Algorithm for Processing Program Transformations*

Sofoklis Efremidis
David Gries

TR 93-1389
October 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* This work was supported by DARPA-ONR Grant N00014-91-J-4123.

An Algorithm for Processing Program Transformations

Sofoklis Efremidis
David Gries
Cornell University*
Ithaca, NY 14853
{gries,sofoklis}@cs.cornell.edu

Abstract

An algorithm for processing program transformations as described by the transform construct is presented. The algorithm constructs a coordinate transformation of an abstract program based on a set of transforms and transform directives applied to it.

1 Introduction

A new language construct, the transform (as a feature of the programming language *Polya* [GV92]), has been proposed for program transformations at the source program level [GV91]. The advantage of the transform is that it lets programmers write programs in an abstract form, which can then be transformed under their directives to a more concrete form that can be accepted and executed by a machine. Hence, programs may be written at a much higher (more abstract) level than most conventional programming languages allow. A transform describes the replacement of a program variable of some type by another variable, perhaps of a different type. A transform directive specifies which transform to use to replace a particular program variable.

The abstract program is transformed to a more concrete form with a set of transforms. Each transform contains rules that prescribe various ways of transforming components of a program. The selection of the transformations to be applied to program variables is localized. By changing a single directive, a new transformation can be applied to the abstract program, resulting in a completely different concrete program.

In this paper, an algorithm that transforms programs according to a set of transforms and transform directives is presented. The paper is organized as follows: In Section 2, the necessary background for the transform construct is given and, in Section 3, an overview of the algorithm is presented. In Section 4, the pattern matching function that is used in the algorithm is defined. Section 5 contains the definition of replacement instantiation. The algorithm for program transformation is given in Section 6 and is discussed in subsequent sections. Section 7 discusses the transformation of program variables and Section 8 discusses the transformation of constants. In

*This work was supported by DARPA-ONR Grant N00014-91-J-4123.

Section 9, the construction of expression representations and transformations is presented. Section 10 discusses the transformation of statements. In Section 11, transformations of programs is discussed. In Section 12 a proof of correctness of the transformation algorithm is given and Section 13 contains a complexity analysis of the algorithm. Finally, Section 14 contains conclusions and comments on the present work.

2 The Transform

A transform describes a coordinate transformation, the replacement of some program variables by new ones [GV91]. A coordinate transformation can be a partial implementation of an abstract data type [GP85], a data refinement of a piece of code [MG90, Mor89], or a general transformation like the transformation of a dummy variable in a loop for efficiency purposes [Knu63]. The declaration of a transform has the following form:

transform $T(\overline{p}{:}\overline{t})$;
  var $\overline{av}{:}\overline{at}$ **into** $\overline{cv}{:}\overline{ct}$
  {*coupling invariant*}
  transform rules
end

In the sequel we call the program that is being transformed the *abstract program* and the program that is the result of the transformation the *concrete program.* The same naming convention applies to parts of the abstract and concrete programs, for example we talk about *abstract statements*, *abstract expressions*, *concrete statements*, and *concrete expressions.* If the abstract program is represented by a tree, we refer to it as the *abstract tree* which has *abstract nodes.* Similarly, we use the terms *concrete tree* and *concrete node.*

In the sections that follow we discuss the various components of a transform. An example of a transform is given in Figure 1.

Transform name and transform parameters

Each transform is uniquely identified by its name. The name of the transform above is T. A transform may have one or more parameters, which are enclosed in parentheses following the name of the transform. If a transform has no parameters, the parentheses are omitted. The parameters are separated by commas; both the name and the type of each parameter have to be given. Transform T above has parameters $\overline{p}$, which have types $\overline{t}$. The parameters of a transform are bound to values with the *transform directive*, which will be discussed later.

Abstract and concrete variables

In the example of transform T above, $\overline{av}$ is a list of *abstract variables* (and $\overline{at}$ is a corresponding list of their types) and $\overline{cv}$ is a list of *concrete variables* (and $\overline{ct}$ is a corresponding list of their types). The abstract variables of a transform describe the program variables (and their types) on which the transform can be applied. The concrete variables of a transform describe the program variables that are generated as the result of applying the coordinate transformation that is described by the transform.

The two lists $\overline{av}$ and $\overline{cv}$ need not have the same length. Both the abstract and the concrete variables are dummies: their consistent renaming throughout the transform will not change the meaning of the transform.

The abstract variables of a transform are matched to the program variables on which the transform is applied (with a transform directive). The concrete variables of a transform are used for generating the appropriate program variables that result from the coordinate transformation. In a later section we discuss how the abstract and concrete variables of a transform are used.

Coupling invariant

The *coupling invariant* is a predicate that relates the abstract and concrete variables. The coupling invariant has no bearing on the transformation process, which is purely syntactic. It is used by the author of a transform for proving the correctness of each transform rule. Later in this section we discuss the proof obligations for the author of a transform.

The coupling invariant of a transform T gives rise to the definition of a *representation* of an expression according to T. Let T be a transform that transforms $v_1{:}t_1$ to $v_2{:}t_2$ and let $CI_T(v_1, v_2)$ be its coupling invariant. An expression $r{:}t_2$ is a T representation of an abstract expression $e{:}t_1$ iff $CI_T(e, r)$. For example, for a variable $v{:}complex$ that is represented by

u:**record** $re, im{:}real$ **end**

the coupling invariant is $v = u.re + i \cdot u.im$. Note that a representation of an expression is defined only when transform T transforms a single variable to another single variable.

Transform rules

Each rule in the list of *transform rules* describes a way of replacing an expression or statement that involves one or more abstract variables or has a subexpression of the same type as an abstract variable. Each rule may have one of the following forms:

$\{P\}$ *expression-pattern* **into** *expression-replacement*
$\{P\}$ *statement-pattern* **into** *statement-replacement*

A transform rule may be preceded by a precondition (P in the example above) enclosed in braces. The user of a transform has the obligation to verify that the precondition of a rule is satisfied at the place in the program where the rule is applicable. The precondition has no effect on the transformation process, which is a syntactic process.

The first kind of rule prescribes the transformation of an expression that may contain abstract variables or expressions. Whenever *expression-pattern* matches an expression e of an abstract program, e can be replaced by the corresponding instance of *expression-replacement*.

The second kind of rule prescribes the transformation of a statement that may contain abstract variables or expressions. Whenever *statement-pattern* matches a statement s of an abstract program (pattern matching is defined in Section 4), s can be replaced by the corresponding instance of *statement-replacement*.

The rules above describe the *transformation* of an expression or a statement. A transform may contain a third kind of rule, a *representation rule* that has the following form

$$\{P\}\ [\![\textit{expression-pattern}]\!]\ =\ \textit{expression-replacement}\ \text{¡}\textit{representation}$$

A transform may have rules of the third kind only if it transforms exactly one abstract variable to one concrete variable. A representation rule prescribes a way of constructing a *representation* of an expression. Whenever *expression-pattern* matches an expression of an abstract program, the corresponding instance of *expression-replacement* is used to construct the specified representation of the expression. This is explained in more detail in a subsequent section. The type of the corresponding instance of *expression-replacement* is the same as the type of the concrete variable of the transform whose representation is constructed. A representation of an expression depends on the representations and/or transformations of its subexpressions. The representation of a variable according to a transform T is prescribed by the first **into** rule of T, and the representations of constants are prescribed by transform rules. A pattern can refer to a specific representation of an expression, which can be used in the corresponding replacement of the transform rule.

The name of a transform can be used as a unique identification for a concrete representation of an abstract expression. Thus, an abstract expression may have a representation according to transform T_1 (a T_1 representation) and a representation according to transform T_2 (a T_2 representation). The *representation* after symbol "¡" in the third kind of transform rule above specifies which representation of the abstract expression is constructed when the transform rule is applied. It also specifies the values of the parameters of the transform whose representation is constructed (if the transform has parameters). If "*representation*" is omitted, then the transform rule prescribes the representation of an expression according to the transform in which it appears.

As mentioned above, an expression may have more than one representation with respect to a set of transforms. A special kind of representation rule is the one that specifies a *conversion of representation*: a function from one representation to another. If *expression-pattern* in the third kind of rule above has no component subpatterns, then the corresponding rule is a conversion of representation. The syntax of patterns and replacements of transform rules is discussed later in this section.

Proof obligations

The author of a transform has several proof obligations for proving the correctness of a transform rule, depending on the kind of a transform rule. Here, we outline the proof obligations for showing the correctness of a transform rule.

1. *For rules that prescribe the transformation of an expression*: The correctness of the rule can be shown by proving that the pattern and replacement of the rule are equal, under the assumption that the abstract and concrete variables that appear in the pattern and replacement satisfy their coupling invariants.

2. *For rules that prescribe the transformation of a statement*: The correctness of the rule can be shown by proving that the simultaneous execution of the pattern and replacement of the rule maintains the coupling invariant, under the assumption that the abstract and concrete variables that appear in the pattern and replacement of the rule satisfy their coupling invariants.

3. *For rules that prescribe the representation of an expression*: The correctness of the rule can be shown by proving that the pattern and replacement satisfy the coupling invariant, under the assumption that the abstract and concrete variables that appear in the pattern and replacement of the rule satisfy their coupling invariants.

Rule patterns and replacements

In this section, we discuss the syntax of patterns and replacements of transform rules. Examples are given using transform BN of Figure 1.

A pattern of a transform rule may have one of the following forms:

1. An abstract statement operator applied to subpatterns. For example, if *stmt_op* is an abstract statement operator of arity n and $\overline{p}$ is a list of patterns such that $\#\overline{p} = n$, then *stmt_op* $\overline{p}$ is a pattern. Pattern BN¡b¡1 := BN¡b¡2 of rule (9) of BN is an example of this case.

2. An abstract expression operator applied to subpatterns. For example, if *exp_op* is an abstract expression operator of arity n and $\overline{p}$ is a list of patterns such that $\#\overline{p} = n$, then *exp_op* $\overline{p}$ is a pattern. Pattern BN¡b¡1 $\vee$ BN¡b¡2 of rule (4) of BN is an example of this case.

3. A reference to a representation of an expression. For example, if T is the name of a transform and a is the name of its abstract variable, then T¡a is a pattern. Different instances of the same representation in the same pattern are distinguished by a number after a second "¡" symbol. For example, T¡a¡1 and T¡a¡2 refer to two (possibly different) T representations of different subtrees of the abstract tree. Pattern BN¡b¡1 of rule (4) of BN is an example of this case.

4. **stmt**-s is a pattern, where s is an identifier. The scope of s is the transform-rule pattern in which it appears. If **stmt**-s appears more than once in a transform-rule pattern, then all occurrences of **stmt**-s refer to the same s. Pattern **stmt**-s_1 of rule (10) of BN is an example of this case.

5. **exp**-e:t is a pattern, where e is an identifier and t is a type. The scope of e is the transform-rule pattern in which it appears. If **exp**-e:t appears more than once in a transform-rule pattern, then all occurrences of **exp**-e:t refer to the same e. Pattern **exp**-e:*bool* of rule (7) of BN is an example of this case.

6. **var**-v:t is a pattern, where v is an identifier and t is a type. The scope of v is the transform-rule pattern in which it appears. If **var**-v:t appears more than once in a transform-rule pattern, then all occurrences of **var**-v:t refer to the same v. Pattern **var**-v:*bool* of rule (11) of BN is an example of this case.

7. **const**-$c\{re\}$:t is a pattern, where c is an identifier and re is a regular expression that describes constants of type t. This pattern can be simplified to c:t if re is just string c. The scope of c is the transform-rule pattern in which it appears. Pattern *false*:*bool* of rule (2) of BN is an example of this case. The same pattern can be written as **const**-*false*{*false*}:*bool*.

8. (p) is a pattern, where p is a pattern. Pattern (BN¡b) of rule (7) of BN is an example of this case.

A replacement of a transform rule has one of the following forms:

1. A concrete statement operator applied to subreplacements. For example, if *stmt_op* is an abstract statement operator of arity n and $\overline{r}$ is a list of replacements such that $\#\overline{r} = n$, then *stmt_op* $\overline{r}$ is a replacement. Replacement BN¡j¡1 := BN¡j¡2 of rule (9) of BN is an example of this case.

2. A concrete expression operator applied to subreplacements. For example, if *exp_op* is an abstract statement operator of arity n and $\overline{r}$ is a list of replacements such that $\#\overline{r} = n$, then *exp_op* $\overline{r}$ is a replacement. Replacement BN¡j¡1 $*$ BN¡j¡2 of rule (5) of BN is an example of this case.

3. A reference to a representation of an expression. For example, if T is a transform, a is the name of its abstract variable and c is the name of its concrete variable, then T¡c is a replacement. This replacement corresponds to the representation of an expression that is referred to by pattern T¡a of the same transform rule. Different instances of the T representation in the same replacement are distinguished by a number that follows a second "¡" symbol. For example, T¡c¡1 and T¡c¡2 refer to two (possibly different) T representations of different subtrees of the abstract tree. The BN¡j part of the replacement of transform rule (4) of BN is an example of this case.

4. s is a replacement, where s is a name defined with **stmt**-s in the pattern of the corresponding transform rule. The s_1 part of the replacement of transform rule (10) of BN is an example of this case.

5. e is a replacement, where e is a name defined with **exp**-e:t in the pattern of the corresponding transform rule. The x part of the replacement of transform rule (7) of BN is an example of this case.

6. v is a replacement, where v is a name defined with **var**-v:t in the pattern of the corresponding transform rule. The v part of the replacement of transform rule (11) of BN is an example of this case.

7. c is a replacement, where c is a name defined with **const**-$c\{re\}$:t in the pattern of the corresponding transform rule.

8. A reference to a parameter of a transform is a replacement. For example, if p is a parameter of a transform T whose abstract variable is a, then T¡p is a replacement. This replacement corresponds to the parameter that is associated with the T representation that is referenced by the T¡a part of the corresponding pattern of the transform rule. Different instances of the

same parameter in the same replacement are distinguished by a number following a second "¡" symbol. For example, T¡p¡1 and T¡p¡2 refer to the two (possibly different) instances of parameter p of the T representations that are referenced by T¡a¡1 and T¡a¡2, respectively, of the corresponding pattern.

9. (r) is a replacement, where r is a replacement. Replacement (BN¡j) of transform rule (3) of BN is an example of this case.

In the case of a rule that specifies a representation of an expression, the replacement of the rule has to specify which concrete representation of the abstract expression is constructed and the values of the parameters of the corresponding transform. For example, if a rule specifies the construction of the T representation of an expression, then this is denoted by the symbols "¡T" that follow the replacement (assuming that transform T has no parameters). If no such symbols follow the replacement, then the rule specifies a representation according to the transform in which it appears.

Assume that T has an abstract variable a and parameters $\overline{p}$. Each reference T¡a to a T representation of an expression has an instance of parameters $\overline{p}$ associated with it. We use the notation T¡p_i ($0 \leq i < \#\overline{p}$) to refer to parameter p that is associated with representation T¡a. If a transform rule specifies the construction of a T representation for an expression, then it has to specify the values of parameters of T. For example, assume that T has one parameter p and one abstract variable a. If a rule specifies the T representation of an expression and the associated value of parameter p is twice the value of p that is associated with T¡a, then the notation ¡$T(2 *$ T¡$p)$ is used for the *representation* part of the rule. The value of the parameter of the resulting representation can be a function of the parameters of the transforms that are associated with transform references only.

An example of a transform

Figure 1 contains an example of a transform for the transformation of variables of type *bool*. Its abstract variable is of type *bool*, and its concrete variable is of type *nat*. The transform provides a way of replacing a variable of type *bool* by a variable of type *nat* and ways of replacing boolean expressions that contain the operators $\vee$, $\wedge$ and $\neg$ by new ones that do not contain these operators. The parenthesized numbers on the left side of the transform rules are used only for reference purposes and do not appear in an actual program.

We discuss some points about transform BN.

- Transform BN has no parameters. The precondition of each rule is *true* and is omitted. Each one of the rules (1) to (7) prescribes the construction of a BN representation, so the symbol "¡BN" is omitted from the end of the rule.

- Transform BN contains rules that define the BN representation of boolean constants *true* and *false* (rules (1) and (2)). According to BN, one representation of *true* is natural number 1 and one representation of *false* is natural number 0.

- Transform BN contains rules that define the BN representation of boolean expressions that are formed using operators $\vee$, $\wedge$ and $\neg$ from the BN representations of their subexpressions (rules (4), (5) and (6)). For example, let $e_1 \vee e_2$ be a boolean expression that appears in

transform BN;			
(0)	**var** $b{:}bool$	**into**	**var** $j{:}nat$
	$\{$*Coupling invariant*: $CI(b,j) = b \equiv j > 0\}$		
(1)	$[\!]$ $[\![true{:}bool]\!]$	$=$	1
(2)	$[\!]$ $[\![false{:}bool]\!]$	$=$	0
(3)	$[\!]$ $[\![(BN¡b)]\!]$	$=$	$(BN¡j)$
(4)	$[\!]$ $[\![BN¡b¡1 \vee BN¡b¡2]\!]$	$=$	$BN¡j¡1 + BN¡j¡2$
(5)	$[\!]$ $[\![BN¡b¡1 \wedge BN¡b¡2]\!]$	$=$	$BN¡j¡1 * BN¡j¡2$
(6)	$[\!]$ $[\![\neg BN¡b]\!]$	$=$	**if** $BN¡j > 0$ **then** 0 **else** 1
(7)	$[\!]$ $[\![$**exp**-$x{:}bool]\!]$	$=$	**if** x **then** 1 **else** 0
(8)	$[\!]$ $BN¡b$	**into**	$BN¡j > 0$
(9)	$[\!]$ $BN¡b¡1 := BN¡b¡2$	**into**	$BN¡j¡1 := BN¡j¡2$
(10)	$[\!]$ **if** $BN¡b$ **then** **stmt**-s_1 **else** **stmt**-s_2	**into**	**if** $BN¡j > 0$ **then** s_1 **else** s_2
(11)	$[\!]$ **var**-$v{:}bool := BN¡b$	**into**	$v := BN¡j > 0$
end			

Figure 1: An example of a transform for the transformation of variables of type *bool*.

a program (where e_1 and e_2 are boolean expressions). If e_1 has a BN representation r_1 (say) and e_2 has a BN representation r_2 (say), then according to rule (4), the BN representation of $e_1 \vee e_2$ is $r_1 + r_2$.

- Rule (7) prescribes the construction of a BN representation of any expression of type *bool*. It is a conversion of representation from the *default* representation of an expression (which is the expression itself) to its BN representation.

- Rule (8) prescribes the construction of a *transformation* of an expression that has a BN representation. If expression e has a BN representation r, then according to rule (8) the transformation of e is $r > 0$.

- Rule (9) prescribes the construction of a *transformation* of a statement that involves abstract variables that have BN representations.

- Rule (10) prescribes the construction of a transformation of an **if** statement whose boolean expression has a BN representation. (This rule is not needed in BN and is presented only as an example).

- Rule (11) prescribes the construction of a transformation of an assignment statement whose right-hand-side expression has a BN representation. (This rule is not needed in BN and is presented only as an example).

The correctness of BN can be shown by proving each rule correct, using coupling invariant CI, as discussed earlier in this section. For example the correctness of rule (5) of BN is shown as follows.

$$\begin{aligned} & b_1 \wedge b_2 \\ = \; & \ll CI(b_1, j_1), CI(b_2, c_2) \gg \end{aligned}$$

$$
\begin{aligned}
& j_1 > 0 \wedge j_2 > 0 \\
= \;& \ll \textit{since } j_1,\, j_2 \textit{ are of type nat} \gg \\
& j_1 * j_2 > 0
\end{aligned}
$$

i.e. $CI(b_1 \wedge b_2, j_1 * j_2)$.

Transform directives

A *transform directive* specifies a transform to be applied to a variable and gives values to the parameters of the transform. There are two kinds of transform directives.

The directive

change $\overline{v}$ **using** $T(\overline{w})$

specifies transform T to be applied to variables $\overline{v}$. T is *applicable* to $\overline{v}$ only if the types of $\overline{v}$ are the same as the types of the abstract variables of T. When a transform directive is processed, $\overline{v}$ is replaced by new variables that have the same types as the types of the concrete variables of T. Different variables of the same type may be transformed with different transforms.

For example, if abstract variable a:*bool* is to be transformed with *BN*, the transform directive would be:

change a **using** *BN*.

Transform *BN* is applicable to a since the type of variable a is the same as the type of the abstract variable of *BN*. When this directive is processed, a is replaced by a new variable a_c (say) of type *nat*. Transform *BN* has no parameters. If *BN* had one parameter of type *int*, then a transformation directive would had been

change a **using** $BN(10)$.

A second kind of directive may be given for the transformation of variables and expressions. When the transform directive

default t **using** $T(\overline{w})$

is given, then, by default, every variable and every constant of type t that is not expressly transformed by a directive is transformed with T. In addition, every expression of type t is removed from the program.

An example of use of a transform

Consider transform *BN* of Figure 1 and suppose that a program contains the following definitions and directives:

var a:*bool*;
var b:*bool*
...
change a **using** *BN*;
change b **using** *BN*

According to these directives variables a and b are transformed with transform *BN*. *BN* is applicable to a and b since the type of the abstract variable of *BN* is the same as the type of a and b, namely *bool*. The definition of a is replaced by

var a_c:*nat*

and the definition of b is replaced by

var b_c:*nat*

where a_c and b_c are the concrete variables that correspond to a and b, respectively. Variables a_c and b_c are the *BN* representations of a and b, respectively.

Suppose that the abstract program contains subexpression

$a \lor b.$

According to rule (4) of transform *BN*, the *BN* representation of the above expression is

$a_c + b_c$

since a_c is the *BN* representation of a and b_c is the *BN* representation of b. Rule (8) of transform *BN* can be used to construct the following transformation of the above expression:

$a_c + b_c > 0.$

Employing rule (9) of transform *BN*, a transformation of statement

$a := a \lor b$

is

$a_c := a_c + b_c.$

Suppose that variable c is defined in the abstract program as

var c:*bool*

and no directive is given for its transformation. Then statement

$$c := a \vee b$$

gets transformed to

$$c := a_c + b_c > 0$$

as prescribed by rule (11) of *BN*. On the other hand, statement

$$c := c \vee b$$

gets transformed to

$$c := c \vee b_c > 0.$$

It should be emphasized that a transform can be a *partial* implementation of a data type [Pri87]. Consequently, it may not provide implementations for all operations of the data type. In addition, different transforms may provide different partial implementations of the same data type. A given transform is useful in the implementation of an abstract data type that is used in a program if it provides implementations for all operations of the data type that are used in the program. For example, it would be acceptable if transform *BN* did not contain any representation rule for $\vee$. In that case, *BN* could not be used in a program to transform a boolean expression that contains operator $\vee$.

Program transformation

A program is transformed successfully with a set of transforms $\mathcal{T}$ and a set of transform directives $\mathcal{D}$ if

1. Every list of program variables $\overline{v}$ for which a directive

   change $\overline{v}$ **using** $T(\overline{w})$

   is in $\mathcal{D}$ and T is applicable to $\overline{v}$ is replaced by fresh variables $\overline{v_c}$ (say) as prescribed by T,

2. For a directive

   default t **using** $T(\overline{w})$

   in $\mathcal{D}$, every variable $v{:}t$ for which no **change** directive is given, is replaced using $T(\overline{w})$ and every expression of type t is removed from the program.

If the source program is type correct and the transforms have been proved correct, then the transformed program is guaranteed to be type correct.

3 Overview of the transformation algorithm

The algorithm for processing transforms and transform directives works in two phases. During the first phase, the source program, the transforms and the transform directives are processed and converted to an internal representation and the necessary lists and tables are constructed. The second phase carries out the actual program transformation. It processes the internal representation of the source program and applies the transformations as directed by the transforms and the transform directives.

In this section, we give an overview of the transformation algorithm, we describe the representation of abstract tree nodes, transforms and transform directives, and we define functions that will be used later.

Symbol table entries for variables

The symbol table has an entry for each abstract program variable. It also has an entry for each concrete variable that is generated as the result of applying a transform directive to one or more abstract program variables. Each symbol table entry for a variable contains information about the variable, like its name, its type etc.

Suppose transform T transforms $v_1{:}t_1$ to $v_2{:}t_2$. For a directive

change v **using** $T(\overline{w})$

or a directive

default t_1 **using** $T(\overline{w})$

a new variable v_c (say) of type t_2 is generated and a symbol table entry is created for it. The symbol table entry of v contains a reference to transform T, its arguments $\overline{w}$, and the corresponding concrete variable v_c.

Suppose T transforms $\overline{v_1}{:}\overline{t_1}$ to $\overline{v_2}{:}\overline{t_2}$. For a directive

change $\overline{v}$ **using** $T(\overline{w})$

a list of new variables $\overline{v_c}$ (say) of corresponding types $\overline{t_2}$ is generated and a symbol table entry is created for each one of them. The symbol table entry of each v_i, $0 \leq i < \#\overline{v}$, contains a reference to transform T, its arguments $\overline{w}$, every other variable in $\overline{v}$ and the list of the corresponding concrete variables $\overline{v_c}$.

Let v be an abstract program variable. The following functions are used later and are assumed to be primitive:

1. *get_conc_vars*(v) is the list of concrete variables that are created when transform T is applied to v,
2. *get_abs_vars*(v) is the list of abstract variables that are transformed along with v when transform T is applied to v,

3. $get\_trans(v)$ is the transform that is applied to program variable v.

The generation of concrete variables that result from application of a transform to a list of program variables is discussed in Section 7.

Abstract tree nodes

As mentioned before, the first phase converts the source program into an internal representation, which is a tree (henceforth referred to as the *abstract tree*). A node of the abstract tree (an *abstract node*) has one of the following forms:

1. An abstract statement operator applied to substatements and subexpressions. For example, if *stmt_op* is an abstract statement operator of arity n and $\overline{s}$ is a list of statements or expressions such that $\#\overline{s} = n$, then *stmt_op* $\overline{s}$ is an abstract statement node,
2. An abstract expression operator applied to subexpressions. For example, if *exp_op* is an abstract expression operator of arity n and $\overline{e}$ is a list of expressions such that $\#\overline{e} = n$, then *exp_op* $\overline{e}$ is an abstract expression node,
3. A node that is labeled with an abstract variable *av*. For example *var* v is a node labeled with variable v,
4. A node that is labeled with a constant *ac*. For example *const* c is a node labeled with constant c,
5. A node that is labeled with a list of declarations of variables and their types. For example *decl* $\overline{v}{:}\overline{t}$ is an example of such node.

We assume that the source program has been type-checked and that each node of the abstract tree is annotated with its type. Statement nodes have type *void*.

With each node n of the abstract tree we associate the following values:

1. $V(n)$: indicates if there are any variables that need to be replaced in the subtree rooted at n,
2. $C(n)$: indicates if there are any constants that need to be replaced in the subtree rooted at n,
3. $reprs(n)$: set of representations of node n,
4. $trans(n)$: transformation of node n or $\perp$ if n has no transformation.

In a later section we discuss how these values are constructed.

For a node n of the abstract tree, the following functions are used later and are assumed to be primitive:

1. $type\_of(n)$ is the type of node n,
2. $is\_var(n) \equiv$ "*n is a variable*",

3. $is\_const(n) \equiv$ "*n is a constant*",
4. $get\_var(n)$ is the variable at node n,
5. $get\_const(n)$ is the constant at node n,
6. $mk\_stmt\_node(stmt\_op, \overline{n})$ is a statement node that has operator $stmt\_op$ and children $\overline{n}$,
7. $mk\_exp\_node(exp\_op, \overline{n})$ is an expression node that has operator exp_op and children $\overline{n}$,
8. $mk\_var\_node(v)$ is a node labeled with variable v,
9. $mk\_const\_node(c)$ is a node that is labeled with constant c,
10. $mk\_decl\_node(\overline{v}, \overline{t})$ is a declaration node labeled with variables $\overline{v}$ that have corresponding types $\overline{t}$.

In the following, the same name is used interchangeably for a node of the abstract tree and for a subtree of the abstract tree that is rooted at this node.

Transforms

The transforms are represented by a table and three lists of rules. Each transform rule r is represented by a record that contains the following information:

1. The pattern of r,
2. The replacement of r,
3. An indication if r is a representation or a transformation rule,
4. If r is a representation rule, the representation that is constructed (i.e. the name of the corresponding transform and the expressions for constructing the associated parameters).

All representation rules that are not conversion-of-representation rules are kept in a list s_1. Similarly, all conversion-of-representation rules are kept in a list s_2 and all transformation rules form a list s_3. For a rule r, the following functions are used later and are assumed to be primitive.

1. $pattern(r)$ is the pattern of rule r,
2. $replacement(r)$ is the replacement of rule r,
3. $representation(r)$ is the name of the transform whose representation is constructed by rule r,
4. $parameters(r)$ is the list of replacements for constructing the parameters that are associated with $representation(r)$,
5. $trans\_of(r)$ is the name of the transform that contains rule r.

For each transform, its parameters, the list of its abstract variables and their types, and the list of its concrete variables and their types, are kept in a record. For a transform T, the following functions are used later and are assumed to be primitive.

1. $parameters(T)$ is the list of parameters of transform T,
2. $abs\_vars(T)$ is the list of abstract variables of transform T,
3. $conc\_vars(T)$ is the list of concrete variables of transform T.

In the sequel $\mathcal{T}$ is the set of transforms that are used to transform a program.

The form of transform-rule patterns that are used in the algorithm is the following

$$p = stmt\_op\ \overline{p} \mid exp\_op\ \overline{p} \mid T¡a¡k \mid \textbf{stmt-}s \mid \textbf{exp-}e{:}t \mid \textbf{var-}v{:}t \mid \textbf{const-}c\{re\}{:}t \mid (p).$$

The form of transform-rule replacements that are used in the algorithm is the following

$$r = stmt\_op\ \overline{r} \mid exp\_op\ \overline{r} \mid T¡c¡k \mid s \mid e \mid v \mid c \mid T¡p¡k \mid (r).$$

Patterns and replacements were discussed in Section 2.

Transform directives

All transform directives are kept in a set $\mathcal{D}$. Each directive of the form

change $\overline{v}$ **using** $T(\overline{w})$

is represented by a record that contains an indication that it is a **change** directive, the list of variables $\overline{v}$, the name of transform T and the list of parameters $\overline{w}$.

Each directive of the form

default t **using** T $(\overline{w})$

is represented as a record that contains an indication that it is a **default** directive, the name of type t, the name of transform T and the list of parameters $\overline{w}$.

For a directive D in $\mathcal{D}$, the following functions are used later and are assumed primitive

1. $is\_change(D) =$ "*D is a* **change** *directive*",
2. $is\_default(D) =$ "*D is a* **default** *directive*",
3. $vars\_of(D) = \overline{v}$, if D is a **change** directive,
4. $type\_of(D) = t$, if D is a **default** directive,
5. $trans\_of(D) = T$,
6. $params\_of(D) = \overline{w}$.

In the sequel $\mathcal{D}$ is the set of transform directives used for transforming a program.

Program transformation

The first phase of the algorithm is essentially a front end that processes the abstract source program and converts it into the intermediate form that is described above [ASU86]. The details are omitted.

The second phase of the algorithm carries out the transformations described by the transforms and the transform directives. It performs a bottom-up traversal of the abstract tree. At the leaves of the tree, the algorithm uses the transform directives to construct the representation of the node. As each node is visited, the representations and transformation for that node are constructed.

Let n be an expression, a variable or a constant node. If $\neg(V(n) \vee C(n))$, then the transformation algorithm need not construct any transformations for it (but it should construct representations for the node, if possible, since they may be needed later in the transformation). If $V(n) \vee C(n)$, then the transformation algorithm should try to construct representations for n and a transformation for it if there is no directive

default t **using** $T(\overline{w})$

in $\mathcal{D}$ where *type_of* $(n) = t$.

Let n be a statement node. If $\neg(V(n) \vee C(n))$, then the transformation algorithm need not construct any transformation for n. If $V(n) \vee C(n) > 0$, then the transformation algorithm should try to construct a transformation for n.

To construct a representation of an internal node n of the abstract tree, the algorithm tries to match the patterns of the representation rules with n. If a match is successful, the corresponding replacement of the rule is used and the concrete representation of the node is constructed as prescribed by the replacement. Constructing conversions of representation and transformations of a node is done in a similar way, by using the list of conversions-of-representation rules and the list of transformation rules, respectively.

After the construction of representations and transformations for the abstract program is complete, the algorithm checks if all directives in $\mathcal{D}$ are satisfied: if all variables v for which a **change** directive is given in $\mathcal{D}$ have been replaced by new ones and if the transformed program does not contain any expressions of type t if a directive "**default** t **using** ..." is in $\mathcal{D}$.

Given a program, a set of transforms and a set of transform directives, it may not be possible to find a transformation for the program. In such cases, the algorithm returns an indication that no transformation is possible. On the other hand, it may be possible to construct more than one transformation for the program. In such cases, heuristic methods can be employed to construct the most suitable transformation. The complexity of the structure of a pattern, the cost of the operations involved in a replacement when the corresponding pattern is selected, and the relative order of the pattern with respect to other patterns can serve as some simple heuristics for choosing the most suitable transformation rule to apply.

4 Pattern matching

Let *Id* be the domain of names and $\mathcal{N}$ be the set of natural numbers. Let *AbsNodes* be the domain of abstract tree nodes. A *binding* is an ordered pair of the form (X,Y), where X is a member of $Id + (Id \times \mathcal{N})$ and Y is a member of $Id + AbsNodes$, i.e. a binding is a member of the cartesian product

$$(Id + (Id \times \mathcal{N})) \times (Id + AbsNodes).$$

An *environment* E is a set of bindings that denotes a function, i.e. for each element X of $Id+(Id\times\mathcal{N})$ there is at most one pair (X,Y) in E. The *domain* of an environment E is the set

$$dom(E) = \{X \mid (\exists\, Y \mid : (X,Y) \in E)\}$$

and the *range* of an environment E is the set

$$rng(E) = \{Y \mid (\exists\, X \mid : (X,Y) \in E)\}.$$

The *empty* environment is denoted by $\emptyset$. We distinguish a special environment, the *fail* environment, that is denoted by $\perp$. The fail environment is different from all other environments, including $\emptyset$. Its importance will be explained later, when the definition of pattern matching is given.

In the sequel, notation $[X \mapsto_E Y]$ is used for binding (X,Y) in E. If environment E is obvious from the context, then it is omitted from the subscript of $\mapsto$. If E is an environment and X belongs to $dom(E)$, then we write $E(X)$ for the Y in $rng(E)$ for which (X,Y) is in E.

Pattern matching is defined to be a function

$$match\text{: } pattern \times node \times environment \rightarrow environment.$$

Intuitively $match(p,n,E)$ is:

- An environment that augments E by the new bindings that result from the pattern match of p and n, if p matches n with respect to E,
- $\perp$, if there is no match between p and n with respect to E.

The definition of *match* is given in Figure 2. ML-style pattern matching is used in the definition.

Function *match* makes use of three functions, which we describe informally here.

1. $has\_repr(n,T) \equiv$ "*subtree* n *has a* T *representation*",
2. $match\_re(re,s) \equiv$ "*string* s *belongs to the language of regular expression* re",
3. $rank(c,\bar{c}) = (\downarrow i \mid 0 \le i < \#\bar{c} : c = \bar{c}[i])$.

Function *has_repr* will be described in more detail later, *match_re* is assumed to be primitive. There are efficient algorithms for deciding if a string of characters belongs to the language of a regular expression [HU79].

$match(p, n, E)$

$\quad match(p, n, \perp) \rightarrow \perp$

$\quad match(stmt\_op_1\ \overline{p}, stmt\_op_2\ \overline{n}, E)$

$\qquad$ **case** $stmt\_op_1 = stmt\_op_2 \wedge \#\overline{p} = \#\overline{n} \rightarrow MatchList(\overline{p}, \overline{n}, E)$

$\qquad$ **otherwise** $\perp$

$\quad match(exp\_op_1\ \overline{p}, exp\_op_2\ \overline{n}, E)$

$\qquad$ **case** $exp\_op_1 = exp\_op_2 \wedge \#\overline{p} = \#\overline{q} \wedge type\_of(p) = type\_of(n) \rightarrow MatchList(\overline{p}, \overline{n}, E)$

$\qquad$ **otherwise** $\perp$

$\quad match($$T$¡$a$¡$k$$, n, E)$

$\qquad$ **case** $(T, k) \in dom(E) \wedge E((T, k)) = n \rightarrow E$

$\qquad$ **case** $(T, k) \notin dom(E) \wedge$

$\qquad\quad (has\_repr(n, T) \vee (is\_var(n) \wedge \#abs\_vars(T) > 1 \wedge$

$\qquad\quad rank(c, abs\_vars(T)) = rank(get\_var(n), get\_abs\_vars(get\_var(n))))) \rightarrow$

$\qquad\qquad E \cup \{[(T, k) \mapsto n]\}$

$\qquad$ **otherwise** $\perp$

$\quad match($**stmt**-$s, n, E)$

$\qquad$ **case** $s \in dom(E) \wedge E(s) = n \rightarrow E$

$\qquad$ **case** $s \notin dom(E) \rightarrow E \cup \{[s \mapsto n]\}$

$\qquad$ **otherwise** $\perp$

$\quad match($**exp**-$e{:}t, n, E)$

$\qquad$ **case** $type\_of(n) = t \wedge e \notin dom(E) \rightarrow E \cup \{[e \mapsto n]\}$

$\qquad$ **case** $type\_of(n) = t \wedge e \in dom(E) \wedge E(e) = n \rightarrow E$

$\qquad$ **otherwise** $\perp$

$\quad match($**var**-$v{:}t, n, E)$

$\qquad$ **case** $type\_of(n) = t \wedge v \in dom(E) \wedge E(v) = get\_var(n) \rightarrow E$

$\qquad$ **case** $type\_of(n) = t \wedge is\_var(n) \wedge v \notin dom(E) \rightarrow E \cup \{[v \mapsto get\_var(n)]\}$

$\qquad$ **otherwise** $\perp$

$\quad match($**const**-$ac\{re\}{:}t, n, E)$

$\qquad$ **case** $type\_of(n) = t \wedge is\_const(n) \wedge match\_re(re, get\_const(n)) \rightarrow E \cup \{[ac \mapsto get\_const(n)]\}$

$\qquad$ **otherwise** $\perp$

$\quad$ **otherwise** $\perp$

end $match$

$MatchList(\overline{p}, \overline{n}, E)$

$\quad$ **if** $\#\overline{p} = 1 \rightarrow match(\overline{p}[0], \overline{n}[0], E)$

$\quad\ [] \ \#\overline{p} > 1 \rightarrow MatchList(\overline{p}[1..], \overline{n}[1..], match(\overline{p}[0], \overline{n}[0], E))$

$\quad$ **fi**

end $MatchList$

Figure 2: Definition of pattern matching.

5 Replacement Instantiation

As explained in Section 4, pattern matching is defined to be an environment that binds names and representation references of a pattern to names and nodes of the abstract tree. When a pattern p matches a tree node n, the corresponding rule that contains p can be used to construct a representation or a transformation of n. To construct such a representation or transformation of n, the corresponding replacement of the rule is used along with the environment of pattern matching.

Let r be a transform rule and n an abstract tree node. Let $E = match(pattern(r), n, \emptyset)$ and assume that $E \neq \perp$. Transform rule r is *applicable* with respect to environment E if

1. If s appears in $replacement(r)$, where s was defined by **stmt**-s in $pattern(r)$

$$V(E(s)) \vee C(E(s)) \Rightarrow has\_trans(E(s)),$$

2. If e appears in $replacement(r)$, where e was defined by **exp**-e:t in $pattern(r)$

$$V(E(s)) \vee C(E(s)) \Rightarrow has\_trans(E(s))$$

   and there is no directive "**default** t **using** ..." in $\mathcal{D}$,

3. If v appears in $replacement(r)$, where v was defined by **var**-v:t in $pattern(r)$, then there is no directive "**default** t **using** ..." in $\mathcal{D}$,

4. If c appears in $replacement(r)$, where c was defined by **const**-$c\{re\}$:t in $pattern(r)$, then there is no directive "**default** t **using** ..." in $\mathcal{D}$,

5. If the syntactic classes of the components that are used in the replacement instantiation are the same as those that are required for the operator that appears in the replacement. We distinguish four syntactic classes: *stmt*, *exp*, *var*, and *const*. Function app', shown in Figure 3, is the definition of this requirement. Its type is

$$inst\colon replacement \times environment \rightarrow bool.$$

   $kind\_of(op, i)$ is the syntactic class of component i of operator op. We assume that it is a primitive function.

We write $app(r, E)$ to denote that transform rule r is applicable with respect to environment E.

Replacement instantiation is defined to be a function

$$inst\colon replacement \times environment \rightarrow node.$$

Intuitively, $inst(r, E)$ is the instantiation of replacement r with respect to environment E.

For a transform rule r we define $inst(replacement(r), E)$ only for cases in which

$$E = match(pattern(r), n, \emptyset) \neq \perp \textbf{ cand } app(r, E)$$

where n is an abstract tree node. The definition of $inst$, given in Figure 4, makes use of three functions, which we describe here.

$$app'(stmt\_op\ \overline{r}, E) = (\wedge i \mid 0 \leq i < \#\overline{r} : kind\_of(stmt\_op, i) = kind(\overline{r}_i, E))$$
$$app'(exp\_op\ \overline{r}, E) = (\wedge i \mid 0 \leq i < \#\overline{r} : kind\_of(exp\_op, i) = kind(\overline{r}_i, E))$$
$$app'(var\ v, E) = true$$
$$app'(const\ v, E) = true$$

$$kind(stmt\_op\ \overline{r}, E) = stmt$$
$$kind(exp\_op\ \overline{r}, E) = exp$$
$$kind(T\text{¡}c\text{¡}k, E) = \mathbf{if}\ is\_var(get\_nrepr(E((T,k)), T, c))\ \mathbf{then}\ var\ \mathbf{else}\ exp$$
$$kind(\mathbf{stmt}\text{-}s, E) = stmt$$
$$kind(\mathbf{exp}\text{-}e, E) = exp$$
$$kind(\mathbf{var}\text{-}v, E) = var$$
$$kind(\mathbf{const}\text{-}c, E) = const$$
$$kind(T\text{¡}p\text{¡}k, E) = exp$$

Figure 3: Definition of app'

1. $has\_trans(n) \equiv$ "*n has a transformation*",
2. $get\_repr(n, T)$ is the T representation of node n,
3. $get\_param(n, T, p)$ is the value of parameter p that is associated with the T representation of abstract tree node n.

Functions *get_param* and *get_repr* will be described in more detail later.

$inst(r, E)$

  $inst(stmt\_op\ \overline{r}, E) = mk\_stmt\_node(stmt\_op, InstList(\overline{r}, E))$

  $inst(exp\_op\ \overline{r}, E) = mk\_exp\_node(exp\_op, InstList(\overline{r}, E))$

  $inst(T$¡c¡$k, E) = get\_nrepr(E((T, k)), T, c)$

  $inst(s, E) =$ **Comment** For s defined in pattern **stmt**-s
    if $V(E(s)) \vee C(E(s)) \rightarrow trans(E(s))$
    $[]\ \neg(V(E(s)) \vee C(E(s))) \rightarrow E(s)$
    fi

  $inst(e, E) =$ **Comment** For e defined in pattern **exp**-e
    if $V(E(e)) \vee C(E(e)) \rightarrow trans(E(e))$
    $[]\ \neg(V(E(e)) \vee C(E(e))) \rightarrow E(e)$
    fi

  $inst(v, E) =$ **Comment** For v defined in pattern **var**-v
    $mk\_var\_node(v)$

  $inst(c, E) =$ **Comment** For c a constant
    $mk\_const\_node(c)$

  $inst(T$¡p¡$k, E) = get\_param(E((T, k)), T, p)$
end *inst*

$InstList(\overline{r}, E)$
  if $\#\overline{r} = 0 \rightarrow [\,]$
  $[]\ \#\overline{r} > 0 \rightarrow inst(\overline{r}[0], E) \frown InstList(\overline{r}[1..], E)$
  fi
end *InstList*

$get\_nrepr(n, T, c)$
  if $\#abs\_vars(T) = 1 \rightarrow get\_repr(n, T)$
  $[]\ \#abs\_vars(T) > 1 \rightarrow mk\_var\_node(get\_conc\_vars(get\_var(n))[rank(c, conc\_vars(T))])$
  fi
end *get_nrepr*

Figure 4: Definition of replacement instantiation.

6 The main algorithm

In this section we present the main algorithm that processes program transformations. We assume that the abstract program, the transforms and transform directives have been preprocessed and presented to the algorithm in an internal form, as discussed in Section 3. We also assume that if there is a directive "**default** t **using** ..." in $\mathcal{D}$, then no transform in $\mathcal{T}$ has concrete variable of type t.

The abstract program to be transformed is presented to the algorithm as a tree n. The set of transforms $\mathcal{T}$ and the set of transform directives $\mathcal{D}$ are accessed through the functions that were defined in Section 3. Figure 5 contains the transformation algorithm. Later sections discuss parts of the algorithm. In particular, functions *mk_repr*, *closure* and *mk_trans* are discussed in subsequent sections. The algorithm is invoked as

xform(n)

where n is the root of the abstract tree. It returns a tree n', which is the coordinate transformation of n according to $\mathcal{T}$ and $\mathcal{D}$ if such a tree exists, $\perp$ otherwise. Tree n' is the internal representation of the concrete program.

In several places, the algorithm in Figure 5 has the following form:

Definition of variable v;
if "**change** v **using** $T(\overline{w})$" $\in \mathcal{D}$ **then** ...
...
... T ...

The semantics of constructs like the above is as follows: We assume that T is bound to the name of a transform at the conditional expression of statement **if** and that this very name is used later on in expressions involving T.

Input: Program P, set of transforms $\mathcal{T}$, set of transform directives $\mathcal{D}$.
Output: Program P': the coordinate transformation of P according to $\mathcal{T}$ and $\mathcal{D}$ if such a program exists, $\perp$ otherwise.

xform(*n*:*node*)
    trav(*n*);
    if *check*(*n*) **then return** *mk*(*n*) **else return** $\perp$
end *xform*

{ *check*(*n*) checks if the tree rooted at n has been successfully transformed, i.e.

1. For every leaf that is not in a subtree of a node that has a transformation and is labeled with a variable v, no directive in $\mathcal{D}$ is applicable to v,
2. For every node m that is not in a subtree of a node that has a transformation there is no directive "**default** t **using** ..." in $\mathcal{D}$ with $\mathit{type\_of}(m) = t$.

}

check(*n*:*node*)
    check(*decl v*)
        return *true*

    check(*var v*)
        return "**change** v **using** ..." $\in \mathcal{D}$ $\vee$ "**default** $\mathit{type\_of}(n)$ **using** ..." $\in \mathcal{D}$

    check(*const c*)
        return $\mathit{has\_trans}(n)$ $\vee$ "**default** $\mathit{type\_of}(n)$ **using** ..." $\in \mathcal{D}$

    check($\mathit{exp\_op}\ \overline{n}$)
        return $\mathit{has\_trans}(n)$ $\vee$ $\neg$("**default** $\mathit{type\_of}(n)$ **using** ..." $\in \mathcal{D}$) $\vee$
            $(\wedge n \mid n \in \overline{n} : \mathit{check}(n))$

    check($\mathit{stmt\_op}\ \overline{n}$)
        return $\mathit{has\_trans}(n)$ $\vee$ $(\wedge n \mid n \in \overline{n} : \mathit{check}(n))$
end *check*

{ *trav*(n) traverses bottom-up the abstract tree rooted at n and constructs
the representations and transformations at each node of the tree.
}

```
trav(n:node)
    trav(decl v:t)
        if #v = 1 then
            if "change v using T(w̄)" ∈ 𝒟 ∨ "default t using T(w̄)" ∈ 𝒟 then begin
                Create new instance v_c of type prescribed by T
                    and make a symbol table entry for it;
                get_abs_vars(v) := v;
                get_conc_vars(v) := v_c
            end else skip
        else
            if "change v using T(w̄)" ∈ 𝒟 then begin
                Create new instances v_c of type prescribed by T
                    and make a symbol table entry for each one;
                foreach u in v do
                    begin get_abs_vars(v) := v;  get_conc_vars(v) := v_c end
            end

    trav(var v)
        if #abs_vars(get_trans(v)) = 1 then begin
            if "change v using T(w̄)" ∈ 𝒟 ∨ "default type_of(n) using T(w̄)" ∈ 𝒟 then
                V(n) := true
            else V(n) := false;
            C(n) := false;
            reprs(n) := {mk_repr(T, mk_var_node(get_conc_vars(v)[0]),
                                 InstList(parameters(T), [parameters(T) ↦ w̄]))};
            closure(n); mk_trans(n)
        end else begin V(n) := false;  C(n) := false;  reprs(n) := ∅;  trans(n) := ⊥ end

    trav(const c)
        V(n) := false;
        if "default type_of(n) using ..." ∈ 𝒟 then C(n) := true else C(n) := false;
        reprs(n) := ∅;  closure(n);  mk_trans(n)

    trav(exp_op n̄)
        foreach n in n̄ do trav(n);
        V(n) = (∨ n | n ∈ n̄ : V(n));  C(n) = (∨ n | n ∈ n̄ : C(n));
        reprs(n) := ∅;  mk_reprs(n);  closure(n);  mk_trans(n)

    trav(stmt_op n̄)
        foreach n in n̄ do trav(n);
        V(n) = (∨ n | n ∈ n̄ : V(n));  C(n) = (∨ n | n ∈ n̄ : C(n));
        reprs(n) := ∅;  mk_trans(n)
end trav
```

{ $mk(n)$ traverses the abstract tree rooted at n and replaces each
node with its transformation if one has been constructed.
}

```
mk(n:node)
    mk(decl v:t)
        if #v = 1 then
            if "change v using T(w̄)" ∈ D ∨ "default t using T(w̄)" ∈ D then
                return mk_decl_node(get_conc_vars(v), conc_type(T))
            else return n
        else
            if "change v using T(w̄)" ∈ D then
                return mk_decl_node(get_conc_vars(v), conc_type(T))
            else return n

    mk(var v)
        if has_trans(n) then return get_trans(n) else return n

    mk(const c)
        if has_trans(n) then return get_trans(n) else return n

    mk(exp_op n̄)
        if has_trans(n) then return get_trans(n)
        else return mk_exp_node(exp_op, mkList(n̄))

    mk(stmt_op n̄)
        if has_trans(n) then return get_trans(n)
        else return mk_stmt_node(stmt_op, mkList(n̄))
end mk

mkList(p̄)
    if #p̄ = 1 → [mk(p̄[0])]
     [] #p̄ > 1 → mk(p̄[0]) ⌢ mkList(p̄[1..])
    fi
end mkList
```

Figure 5: Main transformation algorithm.

7 Representations and transformation of variables

The transformation of variables is directed by transform directives. A transform directive specifies the transform to be used for the replacement of an abstract program variable. We assume that each program variable has a symbol table entry where information about the variable is stored. In addition, each concrete variable that is generated as the result of applying a transform directive to a program variable has a symbol table entry.

Suppose T transforms $v_1{:}t_1$ to $v_2{:}t_2$ and let $v{:}t_1$ be a program variable. If either

change v **using** $T(\overline{w})$

or

default t_1 **using** $T(\overline{w})$

is given, then a new variable v_c (say) of type t_2 is generated and a symbol table entry for it is created. The declaration of v is replaced by

var $v_c{:}t_2$

In addition, for every leaf n of the abstract program tree that is labeled with v

$$V(n) = \begin{cases} \mathit{true} & \text{if "}\textbf{change } v \textbf{ using } T(\overline{w})\text{" or} \\ & \quad\text{"}\textbf{default } t \textbf{ using } T(\overline{w})\text{" is in } \mathcal{D} \\ \mathit{false} & \text{otherwise} \end{cases}$$

and $C(n) = \mathit{false}$. In addition, the following statements are executed.

$\mathit{reprs}(n) := \{\mathit{mk\_repr}(T, \mathit{mk\_var\_node}(v_c), \mathit{InstList}(\mathit{parameters}(T), [\mathit{parameters}(T) \mapsto \overline{w}]))\};$
$\mathit{closure}(n);$
$\mathit{mk\_trans}(n)$

The T representation of v is variable v_c with associated parameters $\overline{w}$. Set *reprs*(n) contains initially the T representation of n. *closure*(n) constructs all representations that can be derived by using conversion-of-representation rules. Functions *closure*, *mk_repr* and *mk_trans* will be defined later.

Suppose T transforms $\overline{v_1}{:}\overline{t_1}$ to $\overline{v_2}{:}\overline{t_2}$. As mentioned in a previous section, for a directive

change $\overline{v}$ **using** $T(\overline{w})$

a list of new variables $\overline{v_c}$ (say) of corresponding types $\overline{t_2}$ is generated and a symbol table entry is created for each one of them. The declaration of $\overline{v}$ is replaced by

var $\overline{v_c}{:}\overline{t_2}$.

For every leaf n of the abstract program tree that is labeled v_i, for $0 \le i < \#\overline{v}$, $V(n) = \mathit{true}$ and $C(n) = \mathit{false}$. In addition, the following statements are executed.

$repr s(n) := \emptyset$; $trans(n) := \perp$

Recall that a transform like T that transforms lists of variables does not contain representation rules for expressions.

8 Representations and transformations of constants

Constants of a type are introduced in the type definition. The concrete form of a constant is described by a regular expression. Constants may appear in a program and may need to be transformed before further transformation of the program can proceed. The transformation of a constant $c{:}t$ can only be directed by a directive

default t **using** $T(\overline{w})$.

A transform T that transforms $v_1{:}t_1$ to $v_2{:}t_2$ may contain rules for the transformation of constants of type t_1. The pattern of each such rule, **const**-*id*$\{re\}{:}t_1$, is a regular expression (*re*) that describes one or more constants of type t_1. The replacement is an expression that may contain *id.*

For an abstract tree node n that is labeled with a constant $c{:}t$, $V(n) = \mathit{false}$ and

$$C(n) = \begin{cases} \mathit{true} & \text{if “}\textbf{default } t \textbf{ using } T(\overline{w})\text{” is in } \mathcal{D} \\ \mathit{false} & \text{otherwise.} \end{cases}$$

In addition, the following statements are executed.

$reprs(n) := \emptyset$; $closure(n)$; $mk\_trans(n)$.

9 Representations and transformations of expressions

If n is an expression node $exp\_op\ \overline{n}$, then

$$\begin{aligned} V(n) &= (\vee n \mid n \in \overline{n} : V(n)) \\ C(n) &= (\vee n \mid n \in \overline{n} : C(n)) \end{aligned}$$

As mentioned in Section 2, each expression may have more than one representation. A representation rule provides a way of constructing a representation of an expression. These rules can be classified in two categories:

- Rules that provide a way of constructing the representation of an expression from the representations and/or transformations of its subexpressions. These rules are kept in a set s_1.
- Conversion-of-representation rules, which convert one representation of an expression into another. All conversion-of-representation rules are kept in a set s_2.

Application of a transform rule to a node is a function

$$apply\colon rule \times node \longrightarrow repr + \perp$$

Intuitively, $apply(r, n)$ is

- A representation that is constructed for abstract tree node n using rule r,
- $\perp$, if r can not be applied to node n.

Type *repr* will be discussed shortly. The definition of *apply* is given in Figure 6. Function *apply* uses *app* whose definition is given in Section 5.

```
apply(r, n)
    var E:environment;
    E := match(pattern(r), n, ∅);
    if E ≠⊥ cand app(r, E) then
        mk_repr(trans_of(r), inst(replacement(r), E), InstList(parameters(r), E))
    else ⊥
end apply
```

Figure 6: Application of a transform rule to an abstract tree node.

Function *apply* makes use of function *mk_repr*, which we describe here. $mk\_repr(T, r, rl)$ is a T representation of an expression whose replacement instantiation is r and list of replacement instantiations rl for the transform parameters that are associated with the T representation. Each such representation is implemented with a record that contains the following information:

1. The name of the transform whose representation is constructed,
2. The representation of the expression according to this transform,
3. A list of the values of the parameters of the transform whose representation is constructed. The values of the parameters appear in the same order as the parameters of the transform in the transform declaration.

We denote $[T, r, rl]$ such a record. Type *repr* is the type of these records. Functions *has_repr*, *get_repr* and *get_param* that were used previously are defined as follows.

1. $has\_repr(n, T) \equiv (\exists r, rl \mid : \ [T, r, rl] \in reprs(n))$,
2. $get\_repr(n, T) = (\exists rl \mid [T, r, rl] \in reprs(n) : r)$,
3. $get\_param(n, T, p) = (\exists r \mid [T, r, rl] \in reprs(n) : rl[rank(p, parameters(T))])$.

```
mk_reprs(n)
    var t:repr + ⊥;
    for r in s1 do
        t := apply(r, n);
        if t ≠⊥ then
            if has_repr(n, representation(r)) then
                if (∃T, rl | t = [r, T, rl] : r) ≻ get_repr(n, representation(r)) then
                    reprs(n) := reprs(n) − {get_repr(n, representation(r))} ∪ {t}
                else skip
            else reprs(n) := reprs(n) ∪ {t}
    end
end mk_reprs
```

Figure 7: Construction of representations of an expression.

With each expression node n of the abstract tree is associated a set *reprs*(n) of all representations of n. At each expression node n of the abstract tree, the transformation algorithm constructs the set *reprs*(n) of representations of n from the representations and/or transformations of the subtrees of n. Function *mk_reprs* of Figure 7 takes a node of the abstract tree as an argument and constructs the representations of the node.

If two different patterns match the same node n, then one of the two has to be chosen for the representation of n to be constructed. Logically, it does not matter which one is chosen. Predicate $\succ$ decides which representation is preferable. Since the test for choosing the appropriate pattern to be used is localized, more elaborate heuristics may be employed for choosing the most suitable pattern.

The conversion-of-representation rules are like the other representation rules, with the exception that the corresponding pattern has no subpatterns. These rules describe the conversion of one representation of an expression to another. Before a conversion of representation rule can be applied to convert the T_1 representation of an expression to a T_2 representation, the T_1 representation has to be constructed. This can be accomplished by maintaining a set S_2 that contains the conversion of representation rules that have not been used yet. As mentioned in a previous section, all conversion-of-representation rules are kept in set s_2. The algorithm for applying the conversion of representation rules in the appropriate order, when abstract tree node n is visited is shown in Figure 8. If n is an abstract tree node, then *closure*(n) augments *reprs*(n) with representations that are obtained with conversions-of-representation rules.

The transformation of an expression is constructed in the same way as the representations of the expression. The only difference is that set s_3, which contains the transformation rules, is used instead. Figure 9 shows the algorithm for constructing a transformation of an expression. $\max_{\succ}(a, b)$ is the maximum of a and b with respect to relation $\succ$. We assume that $a \succ \perp$ for all a. Again, heuristics can be employed for choosing the most appropriate transformation rule when more than one rule can be applied to an abstract tree node.

The transformation algorithm performs the following steps when it visits an expression node n:

```
closure(n)
    var S₂ := s₂;
    var t:repr + ⊥;
    do
        for r in S₂ do
            t := apply(r, n);
            if t ≠⊥ then begin
                S₂ := S₂ − {r};
                if has_repr(n, representation(r)) then
                    if (∃T, rl | t = [r, T, rl] : r) ≻ get_repr(n, representation(r)) then
                        reprs(n) := reprs(n) − {get_repr(n, representation(r))} ∪ {t}
                    else skip
                else reprs(n) := reprs(n) ∪ {t}
            end
        end
    until S₂ is unchanged
end closure
```

Figure 8: Construction of conversion of representations.

```
mk_trans(n)
    var E:environment;
    trans(n) := ⊥;
    for r in s₃ do
        E := match(pattern(r), n, ∅);
        if E ≠⊥ cand app(r, E) then
            trans(n) := max≻(inst(replacement(r), E), trans(n))
    end
end mk_trans
```

Figure 9: Construction of transformation of an expression.

$reprs(n) := \emptyset;\ mk\_reprs(n);\ closure(n);\ mk\_trans(n)$

10 Transformations of statements

If n is a statement node *stmt_op* $\overline{n}$, then

$$\begin{aligned} V(n) &= (\vee n \mid n \in \overline{n} : V(n)) \\ C(n) &= (\vee n \mid n \in \overline{n} : C(n)) \end{aligned}$$

The transformation of a statement is constructed in the same way as the transformation of an expression. When the transformation algorithm visits a statement node n, it executes the following statements.

$reprs(n) := \emptyset;\ mk\_trans(n)$

Again, special heuristics can be employed for choosing the most appropriate transformation rule when more than one rule can be applied to an abstract statement node.

11 Transformations of programs

Let $\mathcal{W}$ be the set of abstract tree nodes that are the highest nodes in the abstract tree for which a transformation has been constructed. The transformation of the original program is successful if

1. For every leaf that is not in a subtree of a node in $\mathcal{W}$ and is labeled with a variable v, no directive in $\mathcal{D}$ is applicable to v,
2. For every node n that is not in a subtree of a node in $\mathcal{W}$, there is no directive

   default t **using** $T \ldots$

   in $\mathcal{D}$ with $type\_of(n) = t$.

These conditions are checked by function *check* that was shown in Section 6.

The transformed program consists of the original tree where each node in $\mathcal{W}$ is replaced by its transformation. Function *mk* presented in Section 6 constructs the concrete tree.

12 Correctness

In this section we discuss the correctness of the transformation algorithm. First, we define *validity* of a transformation with respect to a set of transforms and transform directives. Then we show that the algorithm constructs valid transformations of a program with respect to the transforms and transform directives that appear in the program.

A program P' is a *valid* transformation of program P with respect to a set of transforms $\mathcal{T}$ and a set of transform directives $\mathcal{D}$ if

1. For every directive "**change** $\overline{v}$ **using** $T(\overline{w})$" in $\mathcal{D}$ where $T \in \mathcal{T}$, all free instances of $\overline{v}$ have been eliminated from P, as prescribed by T,

2. For every directive "**default** t **using** $T(\overline{w})$" in $\mathcal{D}$ where $T \in \mathcal{T}$, every variable $v{:}t$ for which no **change** directive is given is replaced using $T(\overline{w})$ and every expression of type t is removed from P.

We write $P \models^{\mathcal{T}}_{\mathcal{D}} P'$ to denote that P' is a valid transformation of P with respect to $\mathcal{T}$ and $\mathcal{D}$.

In this section we show that the transformation algorithm produces a program P' such that $P \models^{\mathcal{T}}_{\mathcal{D}} P'$ iff such a program exists.

Let n' be a representation of an abstract expression node n or a transformation of an abstract statement or expression node n. Define properties $P_V(n, n')$ and $P_C(n, n')$ as follows:

1. $P_V(n, n')$: If $V(n)$ then n' contains no instances of variables in the subtree rooted at n that need to be transformed,

2. $P_C(n, n')$: If $C(n)$ then n' contains no instances of constants in the subtree rooted at n that need to be transformed.

Let r be a representation of an abstract expression node n and t be a transformation of an abstract expression or statement node n. During program transformation, the transformation algorithm maintains the invariant

$$I_n = P_V(n, r) \wedge P_V(n, t) \wedge P_C(n, r) \wedge P_C(n, t)$$

at node n.

Lemma 1 *Let n be an abstract tree node. If for every descendent m of n properties $P_V(m, m')$ and $P_C(m, m')$ hold (where m' is a transformation or a representation of node m), then for every rule r for which $E = match(pattern(r), n, \emptyset)$ and $app(r, E)$, $inst(replacement(r), E)$ has properties P_V and P_C.*

Proof: Let n be an abstract tree node for which $V(n) \vee C(n)$. Assume that for every descendent m of n properties $P_V(m, m')$ and $P_C(m, m')$ hold (i.e. I_m is true), where m' is a transformation or a representation of node m. Let r be a transform rule and $E = match(pattern(r), n, \emptyset)$.

Since $V(n) = (\vee n \mid n \in \overline{n} : V(n))$ and $C(n) = (\vee n \mid n \in \overline{n} : C(n))$, it follows from the definition of *app* in Section 5 that whenever rule r such that $app(r, E)$ is used to construct a representation or a transformation of n, the constructed representation or transformation does not contain any instances of variables or constants in the subtree rooted at n. Functions *apply* and *mk_trans* construct a representation and a transformation of a node, respectively. When transform rule r is applied using environment $E = match(pattern(r), n, \emptyset)$ (i.e. when $inst(replacement(r), E)$ is used), condition $app(r, E)$ holds.

Hence, for a representation r or a transformation t of n

$$P_V(n,r) \wedge P_V(n,t) \wedge P_C(n,r) \wedge P_C(n,t)$$

i.e. I_n is maintained during program transformation. □

Lemma 2 *Let n be a leaf of the abstract tree. I_n is true.*

Proof: We distinguish the following cases:

1. Let n be a node labeled with variable v:t. If "**change** v **using** $T(\overline{w})$" is in $\mathcal{D}$, then $V(n) =$ *true* and $C(n) =$ *false*. In this case v is replaced by a new variable v' as prescribed by T, and v' is v's T representation. Hence $P_V(n, v') \wedge P_C(n, v')$ trivially. In addition, every other conversion of representation r that is derived from v' can not contain any instance of v, hence $P_V(n,r) \wedge P_V(n,r)$. Similarly for a transformation t. Hence I_n is true in this case.

2. Let n be a node labeled with variable v:t. If no "**change** v **using** ..." is in $\mathcal{D}$, but "**default** t **using** ..." is in $\mathcal{D}$, then $V(n) =$ *true* and $C(n) =$ *false*. The proof of this case is the same as the previous one.

3. Let n be a node labeled with variable v:t. If no transform directive is applicable to v, then v is not replaced by another variable and $V(n) = C(n) =$ *false*. Hence I_n is trivially true.

4. Let n be a node labeled with constant c:t. If "**default** t **using** ..." is in $\mathcal{D}$, then $V(n) =$ *false* and $C(n) =$ *true*. In this case the representations and transformations of n are constructed. Hence $P_V(n, n') \wedge P_C(n, n')$ trivially for every representation and transformation n' of n.

5. Let n be a node labeled with constant c:t. If no transform directive is applicable to c, then $V(n) = C(n) =$ *false*. Hence I_n is trivially true.

Hence for a leaf n of the abstract tree I_n is true. □

It follows from the previous two lemmas that I_n is true at every node n of the abstract tree. This implies that every representation and transformation that is constructed for a node n of the abstract tree observes all transform directives in $\mathcal{D}$. In addition, the definition of a variable v on which a transform directive D in $\mathcal{D}$ is applicable is replaced by a new definition according D. At the end, the algorithm checks if the original tree has any variables that ought to be transformed and they are not. In this case no rules could be used to transform these variables and the expressions that contain them so the algorithm returns $\perp$. Hence part (1) of the definition of validity is satisfied.

From the definition of *app* it follows that every expression, variable or constant that is used in constructing a representation or a transformation of a program fragment is not of type t where "**default** t **using** ..." is in $\mathcal{D}$. At the end, function *check* checks if the resulting program contains any remaining expressions of type t where "**default** t **using** ..." is in $\mathcal{D}$. If not, then the transformed program is constructed by function *mk*. Hence part (2) of the definition of validity is satisfied and we have the following soundness theorem.

Theorem 1 *Given a program P, a set of transforms $\mathcal{T}$ and a set of transform directives $\mathcal{D}$, the transformation algorithm produces a program P' such that $P \models_{\mathcal{D}}^{\mathcal{T}} P'$ if such a program exists.*

Conversely since the algorithm applies all transform rules and constructs all possible representations for expressions, if there is a program P' such that $P \models_{\mathcal{D}}^{\mathcal{T}} P'$ the algorithm will construct it. Hence we have the following completeness theorem.

Theorem 2 *Given a program P, a set of transforms $\mathcal{T}$ and a set of transform directives $\mathcal{D}$, if there is a program P' such that $P \models_{\mathcal{D}}^{\mathcal{T}} P'$ then the transformation algorithm will construct it.*

13 Complexity analysis of the transformation algorithm

The basic transformation algorithm, which transforms a program fragment that contains no function or procedure calls, runs in polynomial time with respect to the size of the program and the transform rules.

Let n be the size of the abstract program P. Program P is represented internally by an abstract tree that has size $O(n)$. Let also t be the number of transform rules in all transforms in $\mathcal{T}$, p be the maximum pattern size, r be the maximum replacement size and k be the maximum number of transform parameters a transform may have.

At each internal node n of the abstract tree, the transformation algorithm computes $V(n)$ and $C(n)$. These values are the disjunction of the corresponding values associated with the children of node n. Computing $V(n)$ and $C(n)$ takes time $O(w)$ where w is a constant and is the maximum number of children a node can have.

Constructing a representation or a transformation takes time $O(pn + kr)$ since both the pattern and the replacement of a rule are traversed and the matching may require testing equality of subtrees of P.

Since there are at most t rules, the algorithm spends time $O(t(pn + kr))$. We assume that predicate $\succ$ takes time proportional to the size of the abstract tree. The time spent by $\succ$ at each node of the tree is thus $O(tn)$.

Hence at each node of the tree, the algorithm spends time $O(tn + t(pn + kr) + w)$, which is $O(t(pn + kr))$. Therefore, the total time spent by the transformation algorithm is $O(nt(pn + kr))$, which is $O(n^2)$.

14 Conclusions

A prototype implementation of the algorithm described in this report has been developed in *Scheme*. The preprocessor that processes the abstract program, the transforms, and transform directives and produces the necessary data structures for the second phase is about 1500 lines of code. The second phase that carries out the coordinate transformation of the program consists of about 1000 lines of code. The implementation has been tested on small examples containing transforms like BN of Section 2.

The pattern-matching algorithm used for program transformations could perhaps be modified to take advantage of the techniques for tree pattern matching presented in [HO82]. It is not clear how these techniques could be adopted to the program transformation context and the data structures used by the program transformation algorithm.

Other issues pertaining to processing program transformations are reported in other articles. Suppose for example that the abstract program contains a procedure p with one parameter a. If a call $p(x)$ occurs where x is transformed with transform S, then the transformation algorithm must construct a fresh copy of p in which parameter a is transformed with transform S. The problem becomes more complicated if p has more parameters.

Another issue that has been investigated is the case in which one of the parameters or the concrete variable of a transform S is transformed with another transform T. In this case substantial preprocessing of the transforms is required in the first phase of the algorithm, before the program transformation is constructed.

Finally the development of good heuristics for choosing the appropriate pattern when two or more patterns match an abstract tree node, needs further investigation. It seems that a metric should be defined that depends on the complexity of the structure of the pattern, the complexity of the operations involved in the replacement, and the structure of the subtree of the abstract tree that matches the pattern. This metric can then be used for comparing the appropriateness of the patterns that match a subtree of the abstract tree.

References

[ASU86] Aho, A., R. Sethi, and J. Ullman. *Compilers: Principles, Tools and Techniques.* Addison-Wesley, 1986.

[GP85] Gries, D. and J. F. Prins. A new notion of encapsulation. In *Proceedings SIGPLAN 1985 Symposium on Language issues in Programming Environments*, vol. 20 of *SIGPLAN Notices*, July 1985, pp. 131–139.

[GV91] Gries, D. and D. Volpano. The transform—a new language construct. *Structured Programming*, **11**, 1991, pp. 1–10.

[GV92] Gries, D. and D. Volpano. The Definition of Polya. *Technical report*, Department of Computer Science, Cornell University, 1992.

[HO82] Hoffmann, C. H. and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, **29**(1), January 1982, pp. 68–95.

[HU79] Hopcroft, J. E. and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Knu63] Knuth, D. E. *The Art of Programming, Vol. 1: Fundamental Algorithms.* Addison Wesley, 1963.

[MG90] Morgan, C. and P. H. Gardiner. Data refinement by calculation. *Acta Informatica*, **27**, 1990, pp. 481–503.

[Mor89] Morris, J. Laws of data refinement. *Acta Informatica*, **26**, 1989, pp. 287–308.

[Pri87] Prins, J. F. *Partial Implementations in Program Derivations.* Ph.D. dissertation, Department of Computer Science, Cornell University, 1987.