A language and an Integrated Environment for Program Transformations

Isabelle Attali, Valérie Pascual, Christophe Roudet

INRIA
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A language and an integrated environment for program transformations

Isabelle Attali — Valérie Pascual — Christophe Roudet

N° 3313

December 1997

THÈME 2

ISSN 0249-6399

INRIA
SOPHIA ANTIPOLIS

A language and an integrated environment for program transformations

Isabelle Attali , Valérie Pascual , Christophe Roudet

Thème 2 — Génie logiciel
et calcul symbolique
Projet Croap

Rapport de recherche n° 3313 — December 1997 — 18 pages

Abstract: We present the TrfL language, a rule-based language designed for program transformations. For the end-user, TrfL is intended to support both direct manipulation in an interactive setting and automated execution in a stand-alone context. For the designer, the TrfL language features a high-level expressive power based on powerful patterns, pre-conditions and post-actions, access to contextual information such as symbol tables or dataflow graphs, and functional composition of transformations.

For the designer, we also provide an interactive environment for program transformations based on Centaur. This integrated environment makes it possible to build transformation rules by selection of syntax patterns of the object language and to automatically generate the TrfL source code. Static typechecking of TrfL rules is provided, to ensure correct construction of patterns, appropriate use of variables, and visibility rules in a program.

Among numerous application domains (legacy code problems, static optimizations, parallelizations), we propose in this article a complete example based on partial evaluation techniques on a toy imperative language.

Our final goal, with a formal description of the transformations, besides its interest per se, is to be able to provide tools for proving the correctness of the transformations, as well as other important properties (such as confluence, termination, etc).

Key-words: Program Transformations, Semantics, Pattern-Matching, Interactive Environment, Structured editing, Centaur.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 93 65 77 77 - International : +33 4 93 65 77 77 —Fax : 04 93 65 77 65 - International : +33 4 93 65 77 65
à partir du 01/01/1998
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 —Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Un langage et un environment pour les transformations de programmes

Résumé : Nous présentons le langage TrfL, un langage à base de règles conçu pour les transformations de programmes. Pour l'utilisateur final (le programmeur qui transforme ses programmes) TrfL supporte à la fois une manipulation directe dans un contexte interactif et une exécution batch (par exemple pour transformer un ensemble de librairies). Pour le concepteur de transformations, les caractéristiques du langage TrfL sont un grand pouvoir d'expression fondé sur un puissant mécanisme de pattern-matching, la présence de pré-conditions et de post-actions, une composition modulaire, l'accès à des informations contextuelles comme des tables de symboles ou des graphes dataflow. Pour le concepteur, nous fournissons un environnement interactif construit au dessus de Centaur qui facilite grandement l'écriture des transformations grâce à un mécanisme de sélection des arbres de syntaxe abstraite couplé avec une génération automatique du source TrfL.

Un vérificateur de types permet d'assurer que les spécifications TrfL sont correctes (construction et utilisation des arbres).

Parmi de nombreux domaines d'applications possibles (migrations, optimisations, parallélisations), nous proposons dans cet article un exemple complet d'évaluation partielle (la propagation de constantes).

Notre objectif ultime, avec une description formelle des transformations, est de pouvoir fournir des outils de preuve de la correction des transformations décrites, ainsi que d'autres proriétés recherchées (comme la confluence, la terminaison, etc).

Mots-clés : Transformations de programme, Sémantique, Pattern-Matching, Environnement interactif, Édition structurée, Centaur.

1 Introduction

Program transformations are a major concern in the computer scientist community: any user, from the beginner to the most expert wishes to have a set of tools to manipulate, transform, maintain, adapt, and optimize programs. Program transformations can be classified within many categories, from purely syntactic (change a `repeat` loop into a `while` loop) to context-sensitive or semantic (constant propagation, common-subexpression elimination). The reader can refer to [4, 27, 32, 31] for an overview on the subject. Another trend in program transformations is the area of partial evaluation [18, 5]. Partial evaluation is a technique for specializing programs with respect to (parts of) their inputs, building a residual program, and thus eliminating interpretive overhead. Common techniques for building the residual program are:

1. evaluation (and elimination) of expressions depending on static data (i.e. known at specialization-time);
2. duplication of expressions depending on dynamic data (i.e. only known at run-time).

Generally speaking, transformations are expressed with *ad hoc* means (in a conventional programming language), and are neither associated with a formal description nor to certified tools which ensure the correctness of the transformation.

Nevertheless, some systems do support the specific needs of program transformations. We give an overview and compare these systems using a set of criteria in Section 2. From this comparison, we define specific objectives and we present our own proposal for a language expressing program transformations in Section 3. Section 4 describes the static semantics of the TrfL language in Natural Semantics [22]. From this semantics, using the Centaur system [3] we provide a typechecker for the language. Section 5 presents the interactive environment based on Centaur to design and perform program transformations. As an illustration (Section 6), we express in TrfL a global transformation coming from partial evaluation techniques on a toy imperative language. Finally, we conclude in Section 7 with future work, especially concerning tools we want to associate with TrfL.

2 Related Work

We are concerned here with systems that take a program in a given language (in text or tree representation) as input, and produce as output another program, in the same language or in another language (either in text or tree form).

This field of investigation – source-to-source transformation techniques – is particularly active in a variety of topics such as abstract interpretation [8], partial evaluation (PE)[6, 16], semantic-based tools [26], parallelization techniques [15] in different frameworks (functional [5], logic [11], or imperative [9, 24] programming).

Languages and systems for program transformations can be classified according to several features [32], which make easier to compare and evaluate many different approaches.

These criteria are relative to the expressive power of the meta-language (for the designer) as well as performances (time, memory) of associated tools (for the end-user). Criteria are not independent; among others, the application domain is of major influence on other features (for instance, compiling tools are usually working in batch mode).

We detail now the main features of existing transformation systems (see Figure 1 for an general overview). From this comparison, we will draw our own objectives for a proposal in the next section.

1. **Application domain**
   - compiling programs: static analysis, optimization, intermediate code generation. Generally, these systems are used in a batch context, with no interaction with the end-user; see for instance Optran [23], Puma [12], Gentle [29], FNC-2 [21, 20], TXL [7], Typol [22, 10];
   - specializing programs: originally, online partial evaluators corresponded to non-standard interpreters (as opposed to compilers); offline partial evaluators (to get self-application to work [19, 5]), are decomposed in a binding-time analyzer (BTA) and a run-time system. As mentioned in [17], there is a need for support tools for users (see e.g. Schism [5]);
   - language oriented editing: the system knows both syntax and semantics of the language and permits an interactive and guided editing. Among such systems, there are the Cornell Synthesizer Generator [26] and Centaur [3] (TransForm [25]);

- program development using successive refinements: the system gradually refines a specification of a problem into an executable and efficient program; such tools usually work in an interactive mode and provide correctness proofs; see for instance PROSPECTRA [14], KIDS [28], KORSO [30].

2. **patterns**. Pattern-matching is a key aspect of a language dedicated to transformations: the expressive power of patterns can facilitate the task of the designer (e.g. Trafola-H [13]), but can also cause performance problems in the associated tool;

3. **strategies**. Strategies include the choice of the applied transformation rule (rule order, (non-)determinism) as well as usage of unification (e.g. Typol) and access to context-sensitive information (e.g. Optran);

4. **incrementality**. This feature is particularly needed in an interactive mode, at the designer level (adding a new transformation rule) as well as at the end-user level (transforming a program after editing); Attribute Grammars systems are historically prominent in incrementality techniques [26, 21]. Also, features such as *history* and *undoing* are usually coupled with interactive environments;

5. **transformation libraries**. Some systems (PROSPECTRA and KIDS) provide the user with a set of standard transformations. Such systems may also integrate proof techniques for the correctness of the transformations.

lang/syst	mode	patterns	unification	context	notes
Ppml	incremental	basic	no	no	pretty printing
Transform	batch+interactive	basic	no	yes	guided editing
Centaur-Typol	batch	basic	yes	yes	based on Prolog
FNC2-Olga	batch+incremental	basic	no	yes	based on AG
Sgen	incremental	basic	no	yes	based on AG
TXL	batch	basic	yes	yes	functional
Optran	batch	basic	no	yes	based on AG
Trafola	batch+interactive	powerful	yes	yes	functional
Gentle	batch	basic	yes	yes	based on Prolog
Puma	batch	basic	yes	yes	based on AG
KIDS	semi-automatic	basic	no	yes	refinement
Schism	batch & interactive	basic	no	yes	Partial Evaluation
TrfL	batch & interactive	powerful	no	yes	

Figure 1: Transformation Systems: an overview

The term *basic* for patterns means that you can't write a pattern that matches anywhere in a list or that refers to a subtree far from the root of the subject tree.

3 The TrfL Language

As described in the previous section, transformations systems include many different features and application domains. Our aim, with the TrfL language is to promote the expressiveness of the language, with a powerful pattern-matching mechanism and access to contextual information. Our other concern is the run-time level, with two different modes of execution: a batch mode for compiling techniques applications and an interactive mode with support tools to guide the designer and the end-user. We describe the syntax and the semantics of the language as well as the integrated environment in the next three sections.

A TrfL specification is a set of *rules*; each rule is made of two parts: the left-hand side consists of a pattern and an application condition (related to the pattern variables and the context) and the right-hand side is composed of a replacement pattern and actions on the context. The notion of context makes it possible to integrate structures such as symbol tables or dependency graphs. Rules are gathered in *sets*, which appear as functions.

Patterns are expressed using the abstract syntax terms of the source language and, if different, of the target language. TrfL provides non-linear patterns, assignment patterns, and extraction patterns. In the remaining of this section, we give an overview of the language.

3.1 Pattern-Matching

Patterns are the means to locate relevant source code to transform. Patterns in TrfL are an extension of the abstract syntax of the source and target languages. The extension includes variables and special constructs to express more complicated patterns.

Suppose we want to find two nested while loops. In most pattern languages, you must specify how and where to find such items, you are not allowed to access objects directly from the middle of a list or to refer to a subtree whose root is far from the subject tree. In TrfL one may write:

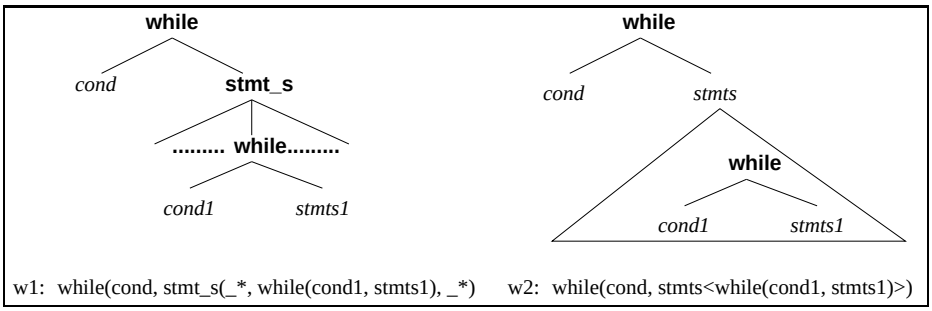

Figure 2: Examples of patterns

The two patterns in Figure 2 both search for pair of nested while loops:

- The pattern w1 is a simple pattern. Simple patterns can be variables or abstract trees possibly with variables. This pattern searches for a *while* loop that contains another *while* loop in its list of statements. Four kinds of variables coexist:
  - **general**: a general variable **name** matches any node and is instanciated with this node (*cond, cond1,* and *stmts1*);
  - **anonymous**: an anonymous variable **_** matches any node but without instanciation;
  - **list**: a list variable **name*** matches some of the descendents of a list operator and instanciates to this sublist;
  - **anonymous list**: an anonymous list variable **_*** combines the properties of anonymous and list variables.

  The last two kinds of variables make it possible to match everywhere in a list and not only the first or last position like in most pattern systems.
- The w2 pattern is an *extract pattern*. An extract pattern matches a tree when:
  - the first pattern (left-hand side of '<') matches the root of the tree;
  - a subtree that matches the second pattern (right-hand side of '<') occurs in the tree.

  This pattern searches for a *while* loop that contains another *while* loop at arbitrary distance in its statements block.

Pattern matching in lists and extract patterns introduce non-determinism, i.e., to one pattern (and one input tree) can correspond several instanciations. In the case of TrfL, the first instanciation is selected.

Some other pattern constructs exist in TrfL:

- Assign patterns. They combine matching and definition of a new variable, which instanciates to the actual matched tree.

```
a:=stmt_s(_*, assign(ident ''foo'', _), _*)
```

- Non-linear patterns. In non-linear patterns a variable may occur more than once. For instance, to match a statement where a variable is incremented, one may write:

```
assign(var, plus(var, _)
```

- Annotation patterns. They combine two patterns, one to match against the subject tree and one to match against an annotation. They can be used to save comments or to match informations stored in structures such as a symbol table,

  ```
  while(cond, block)^ comments(com)
  ```

  comments is the name of the annotation and *com* a variable.An annotation pattern works when:

  - the pattern on the subject tree matches,
  - the annotation is present,
  - and the pattern on the annotation matches.

3.2 Transformation rules organized in sets

Each rule, contains a name, a left-hand side (a pattern for a matching in a given context, with possible pre-conditions), and a right-hand side (an expression and an action to trigger after applying the rule).

An optional input context is designated by an identifier, or a tuple of identifiers. Application conditions can be used to limit the application domain of a rule. They can constrain the left-hand side pattern itself, and possibly induce a composition of patterns, or the context. The expression on the right-hand side can be a simple replacement pattern, or a composed expression (the usual *if, let, case* constructs), or a call to another set of rules. Post-actions are optional and can be used to update contextual information (symbol table).

Transformation rules can be gathered in *sets* for readability and modularity purpose. Sets are typed (types of parameters and results), which avoids variable declarations. A set can be referenced in a rule from another set with its name, as for usual function calls. A set can be viewed as a function that takes as input a tree plus contextual information and returns a new tree or fails if no rule applies. Here's how it works (see figure 3):

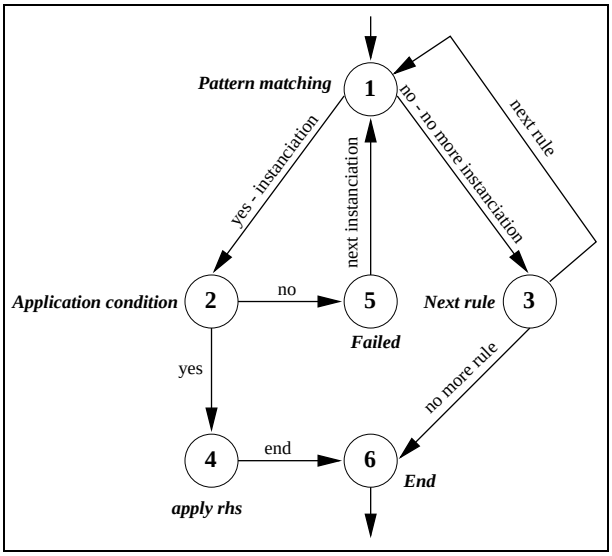

Figure 3: Semantics of a set

1 we check if the left hand side pattern matches the input tree,

  2 if yes, the first instanciation is returned and we check if the pre-condition applies

    4 if yes, we apply the right-hand side of the rule and then we end (6),

    5 if no, we try to find another instanciation for non-deterministic patterns (extract patterns and list patterns) (1)

  3 if no, we try (1) on the next rule , or (6) if there is no more rule left in the set,

6 end.

4 Typechecking transformations

In this section, we describe the static semantics of the TrfL language. We used Natural Semantics [22] and Typol [10] to express this semantics, in approximatively 450 inference rules and axioms.

One interest of this semantics is to provide a formal description of the language ; the other advantage within the Centaur system [3] is that this description is executable and provides a typechecker with helpful error messages (as illustrated in Figure 4).

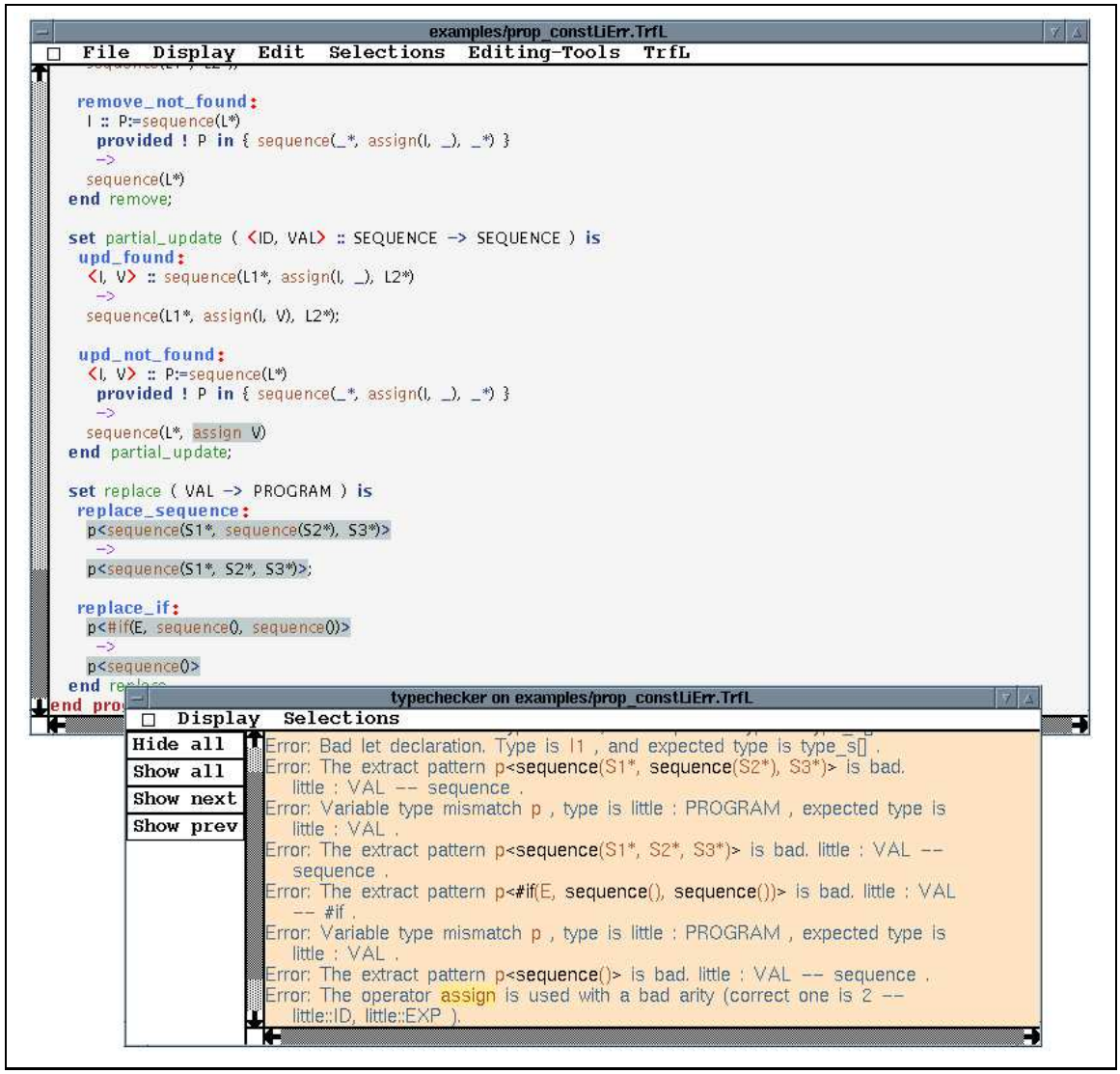

Figure 4: Typechecking a TrfL specification

Some of the points verified by the typechecker are:

- the correctness of rule types given the set type,
- the scoping and double declaration of sets,
- the construction of patterns (arity of operators, correctness of pattern compositions, ...),
- the typing of variables

This semantic specifies the required conditions for a TrfL program to be well typed. Some informations are stored in an environment: each variable found in a pattern (and its type) is stored in a table. Let's focus on the typechecking of patterns. In *tc_pattern* there is one rule for each kind of pattern.

```
set tc_pattern is
judgement TYPE, LANG, SYMBOL_TABLE |- PATT : SYMBOL_TABLE;
var node_patt: NODE_PATT;
var atom_patt: ATOM_PATT;
var variable_patt: VARIABLE;

extract_pattern:
   type, lang, symbol_table |- patt1 : symbol_table'  &
   type, lang, symbol_table' |- patt2 : symbol_table''  &
   tc_sub_type(lang |- patt1', patt2')
   ----------------
   type, lang, symbol_table
      |- extract_patt(patt1, patt2) : symbol_table'' ;

assign_pattern:
   tc_variable(type, lang, symbol_table |- variable : symbol_table')  &
   type, lang, symbol_table' |- patt : symbol_table''
   ----------------
   type, lang, symbol_table
      |- assign_patt(variable, patt) : symbol_table'' ;

annotation_pattern:
   type, lang, symbol_table |- patt : symbol_table'  &
   annot_exists(lang |- annot_name)  &
   anonymous_var, lang, symbol_table' |- patt1 : symbol_table''
   ----------------
   type, lang, symbol_table
      |- annot_patt(patt, annot_name, patt1) : symbol_table'' ;

node_pattern:
   tc_node_patt(type, lang, symbol_table
                  |- node_patt : symbol_table')
   ----------------
   type, lang, symbol_table |- node_patt : symbol_table' ;

atom_pattern:
   tc_atom_patt(type, lang, symbol_table
                  |- atom_patt : symbol_table')
   ----------------
   type, lang, symbol_table |- atom_patt : symbol_table' ;


variable_pattern:
   tc_variable(type, lang, symbol_table |- variable_patt : symbol_table')
   ----------------
   type, lang, symbol_table |- variable_patt : symbol_table' ;
end tc_pattern;
```

For instance, for a *node pattern* we have to verify the following features:

- the operator belongs to the source (or target) language,
- the operator is used with a correct arity,
- the operator belongs to the expected type and
- each son is well typed.

TYPOL has some predefined predicates that give information on operators:

- **operator_in_table(language, operator, integer)**, instantiates *integer* to 1 if *operator* belongs to *language*, 0 otherwise,

- ***look_table(lang, operator, Sons, _)***, returns a structure *Sons* that gives the arity of *operator* and the expected types of its sons. There is four kinds of operators: atomic, singleton (a node with no sons, usually used to represent value like *false* or *true* for example), fixed arity and list arity operator. The following table gives for each type of operator the result of the predicate *look_table*.

atomic	id "integer" or id "string"
singleton	type_s[]
fixed arity	type_s[son1_type, son2_type, ...]
list arity	star_list_of(son_type) or plus_list_of(son_type)

Let's have a look to the set *tc_node_patt*:
This first rule only triggers an error if the operator doesn't belong to the language.

```
operator_not_in_table:
   _, lang, symbol_table
      |- node_patt(operator, patt_s) : symbol_table ;
            provided operator_in_table(lang, op, 0);
            do
               message("TrfL", "TypeCheck", "operator_not_in_table", "Error",
                       s(subject, int 1), "", lang);
```

In this second rule, we first check that the operator belongs to the language, and we get information about its arity and type of sons. Then we check if the operator is used with a correct arity, and if the operator is of the expected type.

```
tc_node_arity(lang, Sons, symbol_table
                 |- node_patt(operator, sons) : symbol_table')
&  check_operator_type(type, lang |- operator)
----------------
type, lang, symbol_table
   |- node_patt(operator, sons) : symbol_table' ;
         provided
            operator_in_table(lang, op, 1)  &
            look_table(lang, operator, Sons, _);
```

In tc_node_arity we check the arity of the operator and call tc_pattern on each son with the appropriate type.

5 An integrated environment for program transformations

In this section, we describe two major tasks in the area of program transformation:

1. expressing transformation rules is the task of the designer;
2. transforming a given program using transformation rules is the task of the end-user.

We deliberately choose an interactive setting for both tasks, for convenience and safety reasons. On the one hand, a designer can be considerably helped by a graphical tool based on actual patterns in an example source program. Thus the transformation rule is not *hand-written* but *automatically generated* by the tool. On the other hand, the end-user can be driven by a selection tool which highlights applicable transformations in a given situation. We describe transformation tools in the context of syntactic editing within Centaur, and provide illustrations based on the example developed in Section 6. These transformation tools are already used in the Centaur system with the TransForm language [25]. We currently work on the adaptation of these techniques to the TrfL language

5.1 Building transformation rules by selection

The starting point for the designer of transformations is a syntactic description of the object language. From these syntactic specifications, Centaur automatically generates a parser and a pretty-printer and combines them in a structure editor. Abstract syntax is used to check the validity of editing operations. Centaur also provides an editing menu to fill place-holders (meta-variables) according to their expected type. Available patterns are

indicated; a simple mouse click on a particular abstract syntax operator will provoke the insertion of that pattern in the current hole, using concrete syntax in the editor.

This editing menu is automatically generated from the syntactic description of the language, and can be organized using hierarchical submenus. It can be augmented by a designer who wants to provide specific editing tools to the programming environment, such as transformation rules.

A designer can create a new transformation rule by selecting a particular fragment of a program in a Centaur editor, and clicking on the "Add Transformation" button of the "Editing-Tools" pulldown.

This command creates a paned view with two Centaur editors, as illustrated in Figure 5, with loop unrolling.

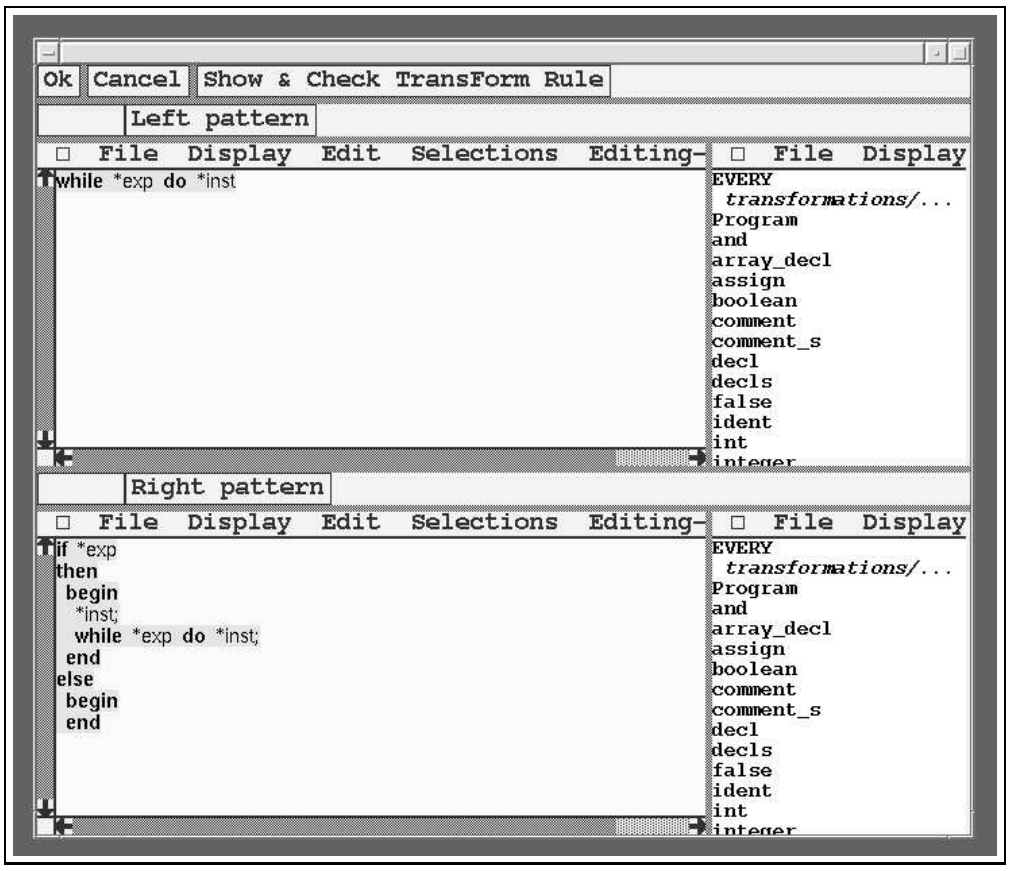

Figure 5: Building a transformation rule.

The upper editor allows one to build the left-hand pattern of the transformation, the lower editor corresponds to the right-hand side pattern. Each fragment of the program belongs to the manipulated language, so the designer is not required to know the abstract syntax of the language. Each editor (both sides of the transformation rule) is associated with editing menus that facilitate the editing of the transformation patterns, using these two fragments of program.

The "Show & Check Transformation Rule" button creates the transformation rule itself, generated from program fragments. This rule may be edited or refined as well (directly at the source level), compiled, and then inserted as a new item into the editing menu.

5.2 Transforming a program by selection

The end-user is guided by the editing menu (possibly augmented by the designer of transformations, sorted in alphabetic order, or re-organized using hierarchical submenus). Automatically, according to the current selection in the editor, this menu is updated with the corresponding available items (both patterns of the language and transformations). Each item of a menu has the same associated action: clicking on a transformation rule replaces the current subtree with the result of the transformation. This is illustrated in Figure 6 with the available item "unfold loop" (note that the item "switch if" is not available, since the current highlighted expression is a `while`).

A special mouse behavior allows one to iterate the application of a transformation at all possible occurrences within a program.

One advantage for the designer and the end-user is that the use of Centaur and syntax-directed editing ensures the syntactic correctness of the resulting program. Another advantage for the end-user is that he is always guided by the menu which interactively highlights available transformations at every stage of the program development.

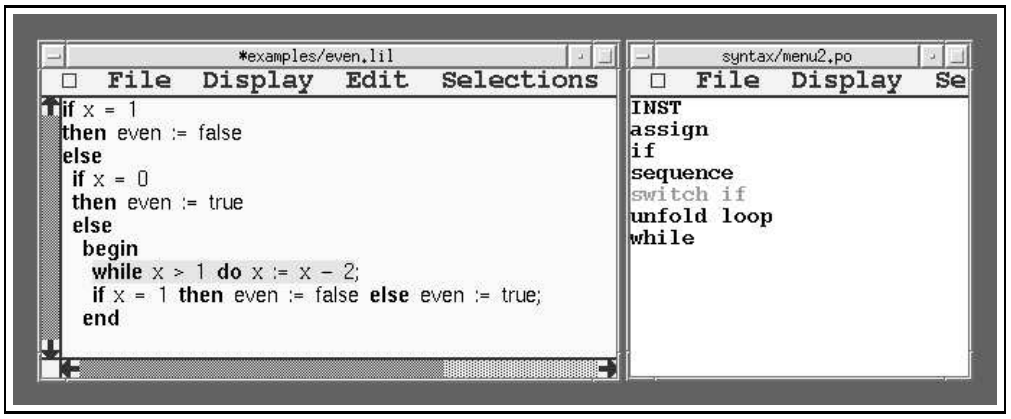

Figure 6: Transforming a program.

6 A complete example

Among many source-to-source program transformations (loop unrolling, inlining, procedure cloning, ...), constant propagation is a global transformation that takes advantage of static data (i.e. known at specialization-time) to propagate constant values and simplify the program, according to these values.

We illustrate our approach with the description of constant propagation on a toy imperative language named Little[1]. We first describe the language itself, give a program example and the residual program after constant propagation. We finally express the transformation in TrfL to show the expressiveness of our language.

Figure 7 shows the abstract syntax for Little, represented as a set of operators and signatures.

```
program    : ID  x  PARAMS  x  DECLS  x  INST  -> PROGRAM
params     : PARAM +                          -> PARAMS
param      : ID  x  TYPE                      -> PARAM
integer    :                                  -> TYPE
boolean    :                                  -> TYPE
decls      : DECL +                           -> DECLS
decl       : ID  x  VAL                       -> DECL
assign     : ID  x  EXP                       -> INST
sequence   : INST *                           -> INST
ifthenelse : EXP  x  INST  x  INST            -> INST
while      : EXP  x  INST                     -> INST
```

Figure 7: The Little abstract syntax.

The Little program of Figure 8.a can be transformed into the residual version of Figure 8.b, using constant propagation.

The principle of the constant propagation is to determine which variable is known statically. Then, statements can be specialized and simplified. More precisely, we compute, at every step in the program, a list of pairs binding a variable and its value (merely assignments), if this value is the same for all possible executions. This list is empty at the beginning, then updated and propagated till the end of the program (see the rule `propagate_program`).

[1]This transformation was originally expressed in Typol in [2].

```
program MAIN (x: integer; b : boolean)
declarations
variable y = b ;
variable z = 3
in
begin
  if b
  then z := x + z
  else z := x - z ;
  if x = 1
  then even := false
  else
   if x = 0
   then even := true
   else
    begin
     while x > 1 do x := x - 2;
     if x = 1
     then even := false
     else even := true ;
    end
end
```

(a) A Little program.

```
program MAIN (b : boolean)
declarations
variable y = b ;
variable z = 3
in
begin
  if b then z := 8 else z := 2 ;
  x := 1;
  even := false;
end
```

(b) The residual program (x = 5).

Figure 8: Constant Propagation for Little.

We detail here the transformation rule for the `if` statement: three cases can occur:

- the expression partially evaluates to `true`, then constant propagation continues on the then-part;
- the expression partially evaluates to `false`, then constant propagation continues on the else-part;
- the expression cannot be statically determined, then constant propagation continues with the common binding list (call to the function `glb`, for greatest lower bound, for instance glb(B1' , B2' -> B' , B1'' , B2'') computes the following decompositions B1'= B' + B1'' and B2' = B' + B2'' where B1'' and B2'' are disjoint sequences) and the partially evaluated `if` statement.

The transformation rule for the if statement is expressed in TrfL in Figure 9 (the full constant propagation transformation is given in the Appendix).

```
set propag (SEQUENCE::INST -> SEQUENCE::INST) is
...
    propag_if:
    D :: #if(E, I1, I2)
        ->
    let
        V := propag_eval(D::E);
        < D'1::I'1 > := propag(D::I1);
        < D'2::I'2 > := propag(D::I2);
        < D', D"1, D"2 > := glb(D'1, D'2)
    in
        case V is
            true() : < D'1::I'1 >;
            false() : < D'2::I'2 >;
            others : < D'::#if(V, sequence(I'1, D"1), sequence(I'2, D"2)) >
        end case
    end let;
...
end propag;
```

Figure 9: Constant propagation in TrfL.

7 Conclusion and Future Work

We have presented a language and an integrated environment for program transformations. The TrfL language is intended to be powerful and easy to use for both designers and end-users. The interactive environment includes support tools to design transformation rules, without actually knowing object language constructors, just by a selection of program fragments. Our environment also provide a typechecker for the language which ensures the correct typing of the transformations. Finally, end-users perform transformations in a guided manner, selecting available transformations in a menu.

We will pursue this definition work for TrfL with an operational semantics. We also want to provide stand-alone efficient transformation tools, working in batch, and independent from Centaur.

Application domains in the area of program transformation, besides partial evaluation and compiling techniques, are numerous: we are currently investigating restructuration and adaptation of scientific Fortran programs, and also parallelization of object-oriented programs [1].

Our final goal, with a complete formal description of TrfL, is to be able to provide proof tools in order to help designers to ensure the correctness of their transformations.

References

[1] I. Attali, D. Caromel, S. O. Ehmety, and S. Lippi. Semantic-based visualization for parallel object-oriented programming. In *Proceedings of the 11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (*OOPSLA'96*), ACM, New-York, October 1996.

[2] Y. Bertot and R. Fraer. Reasoning with executable specifications. In *International Conference on Theory and Practice of Software Development*. Springer-Verlag LNCS 915, May 1995.

[3] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the System. In *The Third Symposium on Software Development Environments (SDE3)*, Boston, 1988. ACM SIGSOFT'88. Also appears as Inria Research Report no. 777 (December 1987).

[4] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24-1:44–67, 1977.

[5] C. Consel. A tour of schism: A partial evaluation system for higher-order applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 145–154. New York: ACM, 1993.

[6] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993*, pages 493–501. ACM, New York: ACM, 1993.

[7] J. R. Cordy and I. H. Carmichael. the TXL programming language – syntax and informal semantics – version 7. Technical Report 93-355, Queen's University at Kingston, Software Technoloy Laboratory – Dept. of Computing and Information Science, June 1993.

[8] P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28-2:324–328, 1996.

[9] M. Das, T. Reps, and P. Van Hentenryck. Semantic foundation of binding-time analysis for imperative programs. In *Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, June 1995*, pages 100–110. New York: ACM, 1995.

[10] T. Despeyroux. TYPOL: A formalism to implement natural semantics. Technical Report 94, INRIA Sophia-Antipolis, March 1988.

[11] J. Gallagher. Tutorial on specialization of logic programs. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 88–98. New York: ACM, 1993.

[12] J. Grosch. Transformation of attributed trees using pattern matching. Compiler Generation Report No. 26, GMD, Forschungsstelle an der Universität Karlsruhe, July 1991.

[13] R. Heckmann. A functional language for the specification of complex tree transformations. In *2nd European Symposium on Programming, Nancy*, pages 175–190. Springer Verlag, New York, NY, 1988. Lecture Notes in Computer Science 300.

[14] B. Hoffmann and B. Krieg-Brueckner. *Program Development by Specification and Transformation*. Number 680 in LNCS. Springer-Verlag, Berlin, 1993.

[15] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *1991 International Conference on Supercomputing, Cologne, Italy*, 1991.

[16] N.D. Jones. The essence of program transformation by partial evaluation and driving. In *Logic, Language and Computation*, volume LNCS 792, pages 206–224. Springer Verlag, 1994.

[17] N.D. Jones. Mix ten years later. In *Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, June 1995*, pages 24–44. New York: ACM, 1995.

[18] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[19] N.D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pages 124–140. Berlin: Springer-Verlag, 1985.

[20] M. Jourdan, C. Le Bellec, and D. Parigot. The Olga attribute grammar description language: Design, implementation and evaluation. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lecture Notes in Computer Science*, pages 222–237. Springer-Verlag, New York–Heidelberg–Berlin, September 1990. Paris.

[21] M. Jourdan and D. Parigot. *The FNC-2 User's Guide and References Manual release 1.18.* INRIA, Rocquencourt, Nov 1994. This manual is periodically updated.

[22] G. Kahn. Natural Semantics. In *Proc. of Symposium on Theoretical Aspects of Computer Science, Passau, Germany, LNCS 247*, 1987.

[23] P. Lipps, U. Möncke, and R. Wilhelm. OPTRAN: A language/system for the specification of program transformations—system overview and experiences. In Dieter Hammer, editor, *Compiler Compilers and High Speed Compilation*, volume 371 of *Lecture Notes in Computer Science*, pages 52–65. Springer-Verlag, New York–Heidelberg–Berlin, October 1988. Berlin.

[24] U. Meyer. Techniques for partial evaluation of imperative languages. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 94–105. New York: ACM, 1991.

[25] V. Pascual. *The Editing and Transformation manual.* INRIA, Sophia–Antipolis, July 1994. in Centaur 2.0 Manual.

[26] T. Reps and T. Teitelbaum. *The Synthesizer Generator: a system for constructing language-based editors.* Springer Verlag, New York, 1988. third edition.

[27] S. Skedzielewski and M. Welcome. Data-flow graph optimization in if1. In *International Conference on Functional Programming Languages and Computer Architecture.* Springer-Verlag LNCS 201, 1985.

[28] D. R. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[29] J. Vollmer. Experiences with Gentle: Efficient compiler construction based on logic programming. In J. Maluszynski and M. Wirsing, editors, *Proc. 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP 91)*, number 528 in Lecture Notes in Computer Science, pages 425–426. Springer Verlag, August 1991.

[30] W. Reif. The KIV Approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, volume 1009 of *Lecture Notes in Computer Science.* Springer Verlag, 1995.

[31] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. Accurate static estimators for program optimization. In *International Conference on Programming Languages Design and Implementation*, pages 85–96. ACM Sigplan Notices 29 (6), 1994.

[32] R. Wilhelm. Tree transformations, functional languages, and attribute grammars. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lecture Notes in Computer Science*, pages 116–129. Springer-Verlag, New York–Heidelberg–Berlin, September 1990. Paris.

Appendix: Constant Propagation in TrfL.

```
program const_propag transform little is
import glb(SEQUENCE, SEQUENCE ->
        SEQUENCE, SEQUENCE, SEQUENCE),
      not_val(VAL),
      interpr(VAL, BINOP, VAL -> EXP),
      dynamic(PARAMS -> PARAMS)
      from Annex
rules

 propagate_program (PROGRAM -> PROGRAM):
  #program(ID, PARAMS, DECLS, I)
   ->
  let
    < D1 :: I1 > := propag(sequence()::I);
    PARAMS' := dynamic(PARAMS)
  in
    #program(ID, PARAMS', DECLS, sequence(I1, D1))
  end let;

set propag_eval (SEQUENCE::EXP -> EXP) is
 dont_touch:
  D :: x
   provided x in {true(), false(), int _}
   ->
  x;

 propag_eval_ident:
  D :: I:=ident N
   ->
  propag_bound(I::D);

 propag_eval_binop:
  D :: bexp(E1, bop, E2)
   ->
  let
    V1 := propag_eval(D::E1);
    V2 := propag_eval(D::E2)
  in
   if V1 in VAL & V2 in VAL
      then let V := interpr(V1, bop, V2) in
             V
           end let
      else bexp(V1, bop, V2)
   end if
  end let
end propag_eval;

set propag_bound (ID::SEQUENCE -> EXP) is

 propag_bound_found:
  I :: sequence(_*, assign(I, V), _*)
   ->
  V;

 propag_bound_not_found:
  I :: sequence(L*)
    provided ! I in {sequence(_*, assign(I, V), _*)}
   ->
  I
end propag_bound;
```

```
set propag (SEQUENCE::INST -> SEQUENCE::INST) is
 propag_if:
  D :: #if(E, I1, I2)
   ->
  let V := propag_eval(B::E);
      < D'1 :: I'1 > := propag(D::I1);
      < D'2 :: I'2 > := propag(D::I2);
      < D', D''1, D''2 > := glb(D'1, D'2)
  in
     case V is
       true() : < D'1 :: I'1 >;
       false() : < D'2 :: I'2 >;
       others : < D' :: #if(V, sequence(I'1, D''1),
                            sequence(I'2, D''2)) >
     end case
  end let;

 propag_while:
  D :: while(E, I)
   ->
  let V := propag_eval(D::E);
      false := false()
  in
    if V == false
       then < D :: sequence() >
       else
         < sequence() :: sequence(D, while(E, I)) >
    end if
  end let;

 propag_sequence_end:
  D :: sequence()
   ->
  < D :: sequence() >;

 propag_sequence_rec:
  D :: sequence(I1, I2*)
   ->
  let < D1 :: I11 > := propag(D::I1);
      < D2 :: I22 > := propag(D1::I2*)
  in
     < D2 :: sequence(I11, I22) >
  end let;

 propag_assign:
  D :: assign(I, E)
   ->
  let val := propag_eval(D::E) in
     if not_val(val)
     then
        let D' := remove(I:: D) in
         < D' :: assign(I, val) >
        end let
     else
        let D' := partial_update(<I, val>::D) in
         < D' :: sequence() >
        end let
   end if
  end let
end propag;
```

```
set remove (ID::SEQUENCE -> SEQUENCE) is
 remove_found:
  I :: sequence(L1*, assign(I, _), L2*)
    ->
  sequence(L1*, L2*);

 remove_not_found:
  I :: P:=sequence(L*)
    provided ! P in {sequence(_*, assign(I, _), _*)}
    ->
  sequence(L*)
end remove;
```

```
set partial_update (<ID, VAL>::SEQUENCE -> SEQUENCE) is
 upd_found:
  <I, V> :: sequence(L1*, assign(I, _), L2*)
    ->
  sequence(L1*, assign(I, V), L2*);

 upd_not_found:
  <I, V> :: P:=sequence(L*)
    provided ! P in {sequence(_*, assign(I, _), _*)}
    ->
  sequence(L*, assign(I, V))
end partial_update
end program
```

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399