Transformation Mechanisms in MT1

A.J.C. Bik P.J. Brinkhaus P.M.W. Knijnenburg H.A.G. Wijshoff

Leiden Institute of Advanced Computer Science,
Leiden University
Niels Bohrweg 1
2333 CA Leiden, the Netherlands

Abstract

In this report we describe a specification language for program transformations on the Fortran 77 source language level. Together with an application engine this specification mechanism allows for a flexible and highly tunable set of transformations to be applied to a program. These transformations are structured around a pattern match mechanism that allows for user defined function that can access the internal program representation directly. This approach allows the specification of a wide range of program transformations, including all commonly used loop level transformations. Next we discuss the Strategy Specification Language (SSL). The SSL is a specification mechanism to sequence elementary transformations. The SSL on the source language level is capable of the sequential composition of transformations, a conditional and a repetetive construct. The conditions for these last constructs consist of the success or failure of some arbitrary condition transformation. Since the condition of a transformation may contain user-defined functions that may access an arbitrary data structure, a mechanism is provided for feedback. This feedback information may be targetted towards specific parts of the program, like a particular loop.

Contents

Chapter 1

Introduction

Optimizing and restructuring compilers incorporate a number of program transformations that replace program fragments by semantically equivalent fragments [Wol91, Wol96, ZC90]. The aim is to obtain more efficient code for a given target architecture. To this end, a collection of suitable transformations and conditions under which to apply them needs to be defined. This collection is dependent upon characteristics of the target architecture. Furthermore, the order in which to apply them needs to be considered. This last problem is commonly known as the *phase ordering problem*.

Traditionally, compilers approach this problem rather statically: the tranformations and their application order are hard coded. This renders these systems rather inflexible. New transformations need to be hard coded on the internal data structure of the compiler. This is a difficult and error-prone process. Likewise, the compiler needs to be adapted to implement different application strategies. However, it is not at all obvious what the best strategy for a given architecture and application domain is. Therefore, experimentation is required to obtain the optimal strategy.

Within the MT1 compilation system [Bik92, BW93, Bri93] these problems are appoached in the following way. The system provides a Transformation Definition Language and a Strategy Specification Language. Transformations and strategies specified in these languages can be dynamically loaded into the compiler and executed. This yields a very flexible system that allows the user to easily add new transformations and experiment with their application strategy. In this report the Transformation Definition Language (TDL) is described.

The TDL is based on pattern matching. The user can specify an *input pattern*, a transformed *output pattern* and a *condition* can be legally and/or beneficially applied. The patterns may consists of sequences of DO loops, IF statements, assignments etc. They may also contain *expression* and *statement variables*. When such sequences are matched against the code under

consideration, these variables are bound to actual expressions and code fragments, respectively. The expression and statement variables can be used in turn in the specification of the output pattern and the condition. This mechanism allows one to specify a large number of transformations, like loop interchange, loop distribution or loop fusion. However, it is not powerful enough to express other important transformations, like loop unrolling. For loop unrolling, the loop body needs to be duplicated, and each ocurrence of the loop index `I` needs to be replaced by `I + 1` in the second copy of the loop body. Therefore, the TDL also allows for *user defined functions* in the output pattern. Such a function may implement for instance the replacement of one expression by another in a sequence of statements. In the TDL, user defined functions are the interface to the internal data structures of the compiler. In this way, any algorithm for transforming the code can be implemented and made accessible to the level of the TDL. Likewise, all kinds of tests on the structure and properties of the code can be implemented. For practical purposes, libraries with user defined function that perform certain elementary functions can be created and used in the formulation of more advanced transformations and conditions.

However, being able to specify transformations is only one part of the general problem of obtaining optimal code by means of program transformations. The order in which these transformations have to be applied needs consideration also [Wol96]. In the initial implementation of the TDL this order is fixed. However, in order to be able to experiment with different application orders for the transformations, a Strategy Specification Language (SSL) has been implemented. This language contains sequential composition of transformations, a choice construct and two repetitive constructs. Like the definition of transformations using the TDL, a file containing a strategy written in the SSL can be dynamically loaded by MT1 and the sequence of transformations specified in that file will be executed. This yields a very flexible system and users are free to change the strategy at any moment.

There are some other projects which separate the implementation of the optimizing strategy from the implementation of the rest of the compiler. Sage++ offers the possibility of specifying a strategy in the C++ language. Sage++ contains a parser that converts the source program into its intermediate format. Sage++ offers a library of routines to walk through the syntax tree, investigate properties of the source code, and apply restructuring transformations onto the source code. These restructuring transformations need to be defined by the user using the primitives offered by the Sage++ library. Although this approach allows a high degree of flexibility, writing strategies takes place at a fairly low level. Moreover, the user needs to hard code the order in which the transformations are applied. If he wants to change this order then part of compilation system needs to be modified and recompiled. This is in contrast to the present approach that offers separate languages to specify transformations and strategies. These specifications are dynamically loaded and executed which means that the system deals with changes in the specifications very flexibly.

This report is organized as follows. In the second chapter, the TDL on source language level is discussed. In section 2.1 a brief overview of the MT1 compilation system is given. In section 2.2 the syntax and semantics of the Transformation Definition Language is described. In section 2.3 it is described how the TDL and MT1 interface. Finally, in section 2.4 the syntax of the TDL is formally specified in BNF notation. In the third chapter, the Strategy Specification Language is discussed. Finally, in the fourth chapter some conclusions are presented.

Chapter 2

Transfromation Definition Language

2.1 The restructuring compiler MT1

MT1 is a Fortran restructuring source-to-source compiler, initially developed as an aid in vectorizing and/or parallelizing sequential programs [Bik92, BW93, Bri93, vDdVW+95]. The core of MT1 consists of a parser that constructs the internal program representation. To this core several modules are hooked that operate on this internal representation. MT1 has a command-driven interface. After it has been started it shows a prompt after which commands, such as loading a Fortran program or transformation file, can be given. Loading a program executes the following phases in MT1.

- Lexical scanning, syntax analysis, semantics checking and construction of internal data structure;
- Interprocedural analysis and optimization;
- Goto elimination;
- Data dependence analysis.

MT1 supports seperate compilation. If a Fortran program consists of several source files, each of these files can be processed independently. However, loading all source files at once will give better results, due to MT1's interprocedural analysis.

MT1 saves several data structures as human readable text in files, all starting with the prefix 'program.' in the current directory. These files can be shown during the execution of MT1, and will be left behind after exiting MT1. Table 2.1 shows the generated files, together with the command by which the file can be shown during execution.

File	Command	Data structure
`program.f`		Program after parsing
`program.txt`	**showprg**	Program after optimization and application of transformations
`program.sym`	**symtb**	Symbol table
`program.dep`	**showdep**	Data dependence graph
`program.cg`	**showcg**	Call graph
`program.cfg`	**showcfg**	Interprocedural control flow graph
`trafo.txt`	**showtrf**	Transformation definitions
`trafo.sym`		Symbol table of transformation definitions

Table 2.1: MT1's internal data structures

MT1 supports full Fortran. In addition, several extensions commonly supported by many other Fortran compilers are supported.

From the Military Standard Definition (MIL-STD-1753) MT1 implements the following features:

- Block `DO` loops
- `DO WHILE` statements.
- `INCLUDE` statements.
- `IMPLICIT NONE` statement.
- The bit manipulation intrinsics `IAND`, `IBSET`, `NOT`, `IBCLR`, `IOR`, `ISHFT`, `IEOR`, `ISHFTC`, `IBITS` and `BTEST`.

Other implemented extensions are:

- The use of binary, octal, and hexadimal constants in any place where integer constants are allowed[1]. Binary constant may be written as $B'b_1 \ldots b_n'$ or $'b_1 \ldots b_n'B$, octal constants as $O'o_1 \ldots o_n'$ or $'o_1 \ldots o_n'O$ and hexadecimal constants as $Z'x_1 \ldots x_n'$ or $'x_1 \ldots x_n'Z$ where b_i is a binary digit, o_i an octal digit and x_i a hexadecimal digit.
- The use of underscores ('_') and dollar-signs ('$') in symbolic names.
- Lower case letters as part of the FORTRAN character set.

[1] MIL-STD-1753 also defines octal and hexadecimal constants. However, they are only allowed in `DATA` statements.

- Symbolic names longer than six characters.
- Byte length in type statements. For example, INTEGER*4 specifies an integer of four bytes. At the moment, byte lengths may only be the default type length, e.g. INTEGER*4 is allowed but INTEGER*2 is not. MT1 compiles types with byte length automatically to the appropriate type, e.g. `REAL*8` is converted into `DOUBLE COMPLEX` This feature has only been added for compatibility reasons.
- Double complex data type (`DOUBLE COMPLEX`)
- The intrinsics `DCMPLX`, `ZABS`, `DIMAG`, `DCONJG`, `ZSQRT`, `ZEXP`, `ZLOG`, `ZSIN`, `ZCOS` which are equivalent with the C-prefixed intrinsics but use double complex types.

2.2 The Transformation Definition Language

The Transformation Definition Language (TDL) enables the user to define its own transformations to be applied by the compiler. The language has been kept as simple as possible but is powerful enough to define a host program transformations and their conditions. For more advanced transformations, which cannot be expressed in the TDL directly, an interface to user defined functions written in C is provided. To make live more easy, MT1 comes with a shared library, `libtdl`, which contains a set of commonly needed user-defined functions.

Section 2.2.1 describes the TDL in detail. Section 2.2.2 explains some sample transformations to illustrate the features of the TDL. Section 2.2.3 describes the user defined functions in more detail. Finally, Section 2.2.4 describes all the functions in the shared library `libtdl`.

2.2.1 Structure of the TDL

The transformation file consists of several `import` and `transform` statements. The `import` statement describes the interface to user defined functions (comparable to function prototypes in C), while the `transform` statement describes an actual transformation. Appendix 2.4 summarizes the syntax rules of the TDL.

Comments

Any text after the characters '%' or '#' up to a new line is treated as a comment. However, the use of '#' is discouraged, because it gives conflicts with the C preprocessor `/lib/cpp`.

Reserved Keywords

A reserved keyword is a string of characters which have special significance to the compiler when used within the transformation file (except when they occur within a comment). The following keywords are reserved:

```
B           dobody   merge
E           follow   nil
S           head     not
and         if       tail
assign      ifbody   transform
condition   into     import
dep         isnil    from
do          issub    true
doall       list
```

In particular, reserved keywords may not be used as identifiers within a `transform` or an `import` statement. The case of the keywords is significant, i.e. all reserved keywords are in lowercase, except the `B`, `E` and `S` keywords.

Identifiers

An identifier is a string of characters used to refer to a user defined function. An identifier can contain any combination of lowercase or uppercase characters, digits or the underscore character ('_'). However, it must start with zero or more underscores, followed by at least one letter. An identifier may not be a reserved keyword (see Section 2.2.1). The case of identifiers is significant.

The `transform` Statement

A transformation is described with the `transform` statement. The statement has the following form:

```
tranform
    pattern1
into
    pattern2
condition
    condition
;
```

where both *pattern1* and *pattern2* must be statement lists.

Whenever *pattern1* (which will be referred to as the left-hand-side pattern) matches a fragment in the program, and *condition* holds, the fragment in the program will be replaced with *pattern2* (the input pattern) if this transformation is applied.

Statement List Patterns A statement list pattern may be one of the following patterns:

- a statement list variable (see Section 2.2.1);
- `list`(*statement*, *statement-list*)
  where *statement* is the head of the list, a statement pattern (see Section 2.2.1) and *statement-list* the tail of the list, a statement list pattern;
- `follow`(*stmt-var*, *statement-list*),
  a built-in function which matches a list of statements ending with *statement-list* while the start of the list is bound to the statement variable *stmt-var*. The `follow` function may be used only in the left-hand-side pattern of a transformation. The main purpose of this function is to split up a statement list in an arbitrary fashion. Consecutive matches of `follow` split up the statement list differently, e.g. given program fragment

  ```
  A(I) = B(I)
  C(I) = D(I)
  E(I) = F(I)
  ```

  and the statement list pattern `follow(!s1, !s2)` (here `!s1` and `!s2` are statement list patterns, see Section 2.2.1). The first match of `follow` will bind the first assignment to `!s1` as a statement list and the second and third assignment to `!s2`. On a second match, `!s1` will be bound to the first and the second assignment and `!s2` to the third. Finally, on a third match, `!s1` will be bound to all three statements and `!s2` will be bound to `nil`.
- `merge`(*statement-list-1*, *statement-list-2*)
  a built-in function which concatenates the two statement list patterns *statement-list-1* and *statement-list-2*. The `merge` function may be used only in the output pattern of a transformation;
- `follow`(*statement variable, statement list*)
  a built-in function that first binds the statement variable to a program fragment of minimal length such that *statement list* also can be bound to the next fragment, and if the transformation is not accepted binds the variable to a fragment of greater length and so on untill either the transformation is accepted or no fragement can be found such

that the variable and the list both can be matched. The `follow` function may be used only in the output pattern of a transformation;

- a user-defined function
  which must have been defined in an import statement (see Section 2.2.1) before the transformation in which the function is used. The return value of the function must be of type statement list (`S`, see section 2.2.3). User-defined function are not allowed in the left-hand-side of a transformation.

- `nil`
  the empty list.

Statement Patterns A statement pattern may be one of the following patterns:

- `assign`(*expr-1*, *expr-2*)
  which describes an assigment statement with *expr-1*, an expression pattern, as the left hand side of the assignment and *expr-2*, also an expression pattern, as the right hand side of the assignment;

- `if`(*expr*, *statement-list*)
  which describes either a logical or a general `IF` statement, with *expr*, an expression pattern, as the condition of the `IF` statement and *statement-list* as the body of the `IF statement`;

- `do`(*expr-1*, *expr-2*, *expr-3*, *expr-4*, *statement-list*)
  which describes a `DO` loop. *expr-1*, *expr-2*, *expr-3* and *expr-4* are expression patterns, where *expr-1* describes the index variable, *expr-2* the lowerbound, *expr-3* the upperbound and *expr-3* the stride of the `DO` loop;

- `doall`(*expr-1*, *expr-2*, *expr-3*, *expr-4*, *statement-list*)
  which describes a `DOALL` loop and is exactly the same as the `do` statement pattern, except that it matches a `DOALL` keyword in case the pattern occurs in the left hand side of a transformation statement or generates a `DOALL` keyword in case the pattern occurs in the output pattern of a transformation statement[2].

See Section 2.2.1 for a description of expression patterns.

[2]Note that `DOALL` loops are not accepted as an extension of standand FORTRAN 77 on input. However, `doall` patterns are provided to turn parallel `DO` loops into `DOALL` loops on output. So, currently MT1 is not always able to read in its own output again.

Expression Patterns A expression pattern may be one of the following patterns:

- an expression variable (see Section 2.2.1);
- a FORTRAN 77 integer, real or logical constant;
- *expr-1 bin-op expr-2*
  where *expr-1* and *expr-2* are expression patterns and *operator* is one of the FORTRAN 77 binary expression operators '*', '+', '-', '/', '**', `.EQ.`, `.NE.`, `.GE.`, `.GT.`, `.LE.`, `.LT.`, `.EQV.`, `.NEQV.`, `.AND.` or `.OR.`;
- *un-op expr*
  where *expr* is an expression pattern and *un-op* is one of the FORTRAN 77 unary expression operators '-' or `.NOT.`;
- (*expr*)
- `vectorize`(*expr-var-1*, *expr-var-2*, *expr-1* : *expr-2* : *expr-3*)
  a built-in function to generate array-sections. This expression pattern may only be used in the output pattern of a transformation statement;
- a user-defined function
  which must have been defined in an import statement (see Section 2.2.1) before the transformation in which the function is used. The return value of the function must be of type expression (`E`, see Section 2.2.3). User-defined function are not allowed in the left-hand-side of a transformation.

Variables Variables may be used to denote an arbitrary expression or statement list within a statement pattern. Expression variables start with '!e' followed by a number, e.g. `!e1` or `!e10`. Statement list variables start with '!s' followed by a number, e.g. `!s2` or `!s10`.

If an expression or an statement list variable occurs in the output pattern of a transformation statement, it must occur also in the left hand side of that statement, otherwise it will be unbounded during the application phase. A statement list variable may be used only once in the left hand side pattern. Expression variable may be used several times in a left hand side pattern to indicate that the expressions bound to different occurences of the expression variable must be syntactically the same, e.g. the statement pattern

```
assign(!e1, !e1)
```

matches the FORTRAN 77 statement

```
A = A
```

but not

```
A = B
```

Conditions A condition in the `condition` clause of a transformation statement may be one of the following boolean expressions:

- `true`
  the boolean true value. Note that there is no constant `false` value. This value can be represented by `not true`;

- `dep` *kind* [*direction*] (*from-stmt-list*, *to-stmt-list*) [`>` *expr-var*]
  returns true if there exist a dependence of kind *kind* with direction *direction* from any statement in *from-stmt-list* to any statement in *to-stmt-list*. The optional [`>` *expr-var*] may be used to denote that the dependence must hold on a variable which occurs in the expression bound to *expr-var*. The *kind* of dependence may be one of `flow`, `anti`, `input`, `output` or '@' (any dependence except `input`). The direction vector *direction* is optional and may be used to denote a direction which should hold for the dependence. The elements of the vector maybe '`<`', '`>`' and '*' (meaning either '`<`' or '`>`'). The length of the vector should correspond to the common number of DO (WHILE) statements surrounding both *from-stmt-list* and *to-stmt-list*. The statement lists *from-stmt-list* and *to-stmt-list* must be references to the matched pattern.

- `isnil`(*statement-variable*)
  a built-in function which returns true, if *statement-variable* is bound to the empty statement list (the statement list pattern `nil`), or false otherwise.

- `issub`(*expr-var-1*, *expr-var-2*)
  a built-in function which returns true, if the expression bound to *expr-var-1* occurs as a subscript expression in the expression bound to *expr-var-2*, or false otherwise. Note that the `issub` always returns false if the *expr-var-1* occurs only within an expression which is either an (intrinsic) function call or an implied DO loop;

- a user-defined-function
  which must have been defined in an import statement (see Section 2.2.1). Within a condition, a function requiring a statement list as argument (of type `S`) may be passed a reference to the matched pattern as well as a regular statement list. The return value of the function must be of type boolean (`B`, see Section 2.2.3);

- `not` *condition*
  returns true if *condition* is false, otherwise true;
- *condition-1* `and` *condition-2*
  returns true if both *condition-1* and *condition-2* are true, otherwise false.

The `not` operator has higher precedence than the `and` operator.

Referencing the Matched Pattern Constructs *from-stmt-list* and *to-stmt-list* consist of references to statement lists in the left-hand-side pattern. Each reference starts with the whole matching program fragment, indicated by a '**$**', and uses the constructs '**.next**', '**.dobody**', and '**.ifbody**' to refer to the next statement list, DO-loop body, or IF-statement, respectively.

The following example illustrates how references in a `dep` construct can be used to specify specific statement lists in a program fragment that matches the following pattern:

```
list( do(!e1, !e2, !e3, !e4,
         list( if(!e5, !s1), !s2 )), !s3 )
```

For the fragment below, both variable `!s3` and **$.next** are bound to the 'rest of program'. Since a body consists of a single statement list, both **$.body.ifbody** and `!s1` are bound to the '. . .' statements inside the IF-body:

```
              $ →  DO I = 1, 100
      $.dobody →    IF (L2) THEN
$.dobody.ifbody →      ...          ← !s1
                    ENDIF
 $.dobody.next →    ...            ← !s2
                  ENDDO
         $.next →  ...              ← !s3
```

Construct '**.head**' can be used at the end of a reference to indicated that only the first statement of the statement list indicated by reference that precedes this construct must be considered. Note that if this single statement is a DO-loop or an IF statement, the statements inside the body are also considered (since they belong to that single statement). So, in the given example, **$.head** specifies the DO-loop with its body (consisting of the IF statement with its body and all statements in the list bound to `!s2`), while **$** specifies this DO-loop with all statements in its body and the following statement list (bound to `!s3` and specified with **$.next**).

The first structure associated with a `dep` construction specifies the statement list from which the source statements must be taken, while the second one specifies the list from which the sink statements must be taken. Note that the use of references (rather than using, for instance, statement variables) enables the programmer to specify arbitrary statements and statement lists within the matching fragment. The compiler verifies, however, whether the references match the specification of the left-hand side pattern (so that **$.next.dobody**, for example, cannot be used, even though `!s3` may be bound to a DO-loop for a particular matching fragment).

The `import` Statement

Before user-defined functions are used within a transformation statement, they have to be declared in an import statement. The statement has the following form:

```
import
    user-defined-function-1 : argument-types -> result-type
    ...
    user-defined-function-n : argument-types -> result-type
from
    shared-library, ...
;
```

Each declared function is identified by one of the identifiers *user-defined-function-1* to *user-defined-function-n*. A function takes arguments as specified by *argument-types*, which is a list of types, separated by white space. An argument type may be either `S`, a statement list or `E`, an expression. The type of the function result is specified by *result-type*, which may be either `S`, a statement list, `E`, an expression or `B`, a boolean value.

In the `from` clause of the import statement, a comma separated list of shared libraries must be specified in which the declared function are searched for. The library names, which must have an extension '.sl' or '.so' and may be preceded by a path, must be delimited by either double quotes ('"') or angle brackets ('<' and '>'). If the library name is delimited by double quotes, the library is searched for in the location as specified. Otherwise the environment variable `TRAFOPATH` is used to determine where to search for the library by prefixing the library name with each path defined by `TRAFOPATH`.

2.2.2 Some Samples Transformations

In this section, some examples of transformations will be explained, to illustrate the features of the TDL.

Loop Vectorization

For vectorization, the built-in function vectorize is used.

```
transform
    list(do(!e1, !e2, !e3, !e4,
         list(assign(!e5, !e6), nil)), !s1)
into
    list(assign(vectorize(!e5, !e1, !e2:!e3:!e4),
                vectorize(!e6, !e1, !e2:!e3:!e4)), !s1)
condition
    not dep flow < ($.dobody, $.dobody)
;
```

Loop Distribution

Loop distribution uses the `follow` construct.

```
transform
     list(do(!e1, !e2, !e3, !e4, follow(!s1, !s2)), !s3)
into
     list(do(!e1, !e2, !e3, !e4, !s1),
     list(do(!e1, !e2, !e3, !e4, !s2), !s3))
condition
     not isnil (!s2) and
     not dep @ < ($.dobody.follow, $.dobody)
;
```

Since a loop can be distributed between any two statements in the loop body, the `follow` construct provides a means to dynamically guide the user to search for this point. Firts `!s1` is bound to the first statement in the loop body and `!s2` to the rest of the loop body. The loop is distributed at this point and the user is asked to accept this distribution or not. If not, `!s1` is bound to the first two statements in the body and `!s2` to the others, and the user is prompted again. And so on, until `!s1` is bound to the entire loop body and `!s2` is `nil`. Then the condition fails and the transformation is not applied anymore.

Loop Unrolling

This transformation uses two user defined functions that come with the standard library and that are described in section 2.2.4. Also, the built-in function `merge` is used. First, the

functions are declared by an `import` statement.

```
import
    tdl_isint : E -> B
    tdl_replace : S E E -> S
from
    <libtdl.sl>
;

transform
    list(do(!e1, !e2, !e3, 1, !s1), !s2)
into
    list(do(!e1, !e2, (!e2 - 1) + ((!e3 - !e2 + 1) / 2) * 2, 2,
            merge(!s1, tdl_replace(!s1, !e1, !e1 + 1))),
    list(if((((!e3 - !e2 + 1) / 2) * 2 .neq. (!e3 - !e2 + 1),
         tdl_replace(!s1, !e1, !e3)), !s2))
condition
    tdl_isint(!e2) and tdl_isint(!e3)
;
```

2.2.3 User defined functions

Because the expressiveness of the transformation language is rather limited, the possibility to call user defined functions, written in C, is implemented. These functions should be stored in one or more shared libraries, that can be dynamically loaded during run-time, i.e. during parsing of the transformation file. Type definitions of user defined functions and libraries to be loaded must be specified by the `import` clause in the transformation file. User defined functions may be called from the output pattern and the condition of a transformation. Usually, boolean functions are used in conditions while functions returning an expression or a statement are used within the output pattern of a transformation (although a call to such a function may also occur as an argument to a boolean function in a condition).

From within a user defined C function, it is possible to make calls to many functions used internally by MT1 to access and maintain the symbol table and the abstract syntax tree representing the Fortran input source.

Parameter Passing

An argument to a user defined function must be either an expression (type 'E') or a statement (type 'S'). These arguments are pointers to copies of nodes in MT1's abstract syntax tree.

Arguments are not passed directly to the user defined function. That means that the C definition of the function should have an empty argument list (i.e. `void` in ANSI C). Instead, expression arguments are passed through an expression stack and statement arguments though a statement stack. The arguments are moved in in order of appearance on these stack, i.e. the first argument is the lowest on one of the stacks and the last argument is on top on one of the stacks. Note that in general the arguments are copied before they are passed to the function. Therefore, the user defined function is responsible for cleaning up the arguments if they are not used as a function result.

The exception is that a function may be passed a statement list which is not a copy but a direct pointer into the abstract syntax tree of the input source (in this case the argument in the transformation definition starts with a '$'). In this case the argument must not be deleted by the function nor may it be returned as a function result.

Return Values

User defined function must return either an expression, a statement or a boolean value. If the function returns a boolean value, the type of the user defined function should be `int` and the function should return zero in case the return value is false or any nonzero value in case the return value is true. As with the arguments of the user defined function, expressions and statements are returned on top of either the expression or statement stack. The C definition of the function should therefore be of type `void`.

2.2.4 LIBTDL Functions

Because writing user-defined functions is not an easy task, MT1 comes with a library, `libtdl`, which provides the transformation writer with a set of commonly needed user-defined functions. This section describes each function in the library.

- **tdl_isint**

  `tdl_isint : E` $\rightarrow$ `B`

  returns `true` if expression `E` is of type integer, otherwise `false`.

- **tdl_isreal**

  `tdl_isreal : E` $\rightarrow$ `B`

  returns `true` if expression `E` is of type real, otherwise `false`.

- **tdl_isdreal**

  `tdl_isdreal : E → B`

  returns `true` if expression `E` is of type double real, otherwise `false`.

- **tdl_iscomplex**

  `tdl_iscomplex : E → B`

  returns `true` if expression `E` is of type complex, otherwise `false`.

- **tdl_isdcomplex**

  `tdl_isdcomplex : E → B`

  returns `true` if expression `E` is of type double complex, otherwise `false`.

- **tdl_islogical**

  `tdl_islogical : E → B`

  returns `true` if expression `E` is of type logical, otherwise `false`.

- **tdl_ischar**

  `tdl_ischar : E → B`

  returns `true` if expression `E` is of type character, otherwise `false`.

- **tdl_iseq**

  `tdl_iseq : E E → B`

  returns `true` if both expression have te same value, otherwise `false`. Both arguments must be constant expressions, otherwise an error messages is given and `false` is returned.

- **tdl_islt**

  `tdl_islt : E E → B`

  returns `true` if the value of the first expression is lower than the value of the second

expression, otherwise `false`. Both arguments must be constant expressions, otherwise an error messages is given and `false` is returned.

- **tdl_isleq**

  `tdl_isleq : E E → B`

  returns `true` if the value of the first expression is lower than or equal to the value of the second expression, otherwise `false`. Both arguments must be constant expressions, otherwise an error messages is given and `false` is returned.

- **tdl_isgt**

  `tdl_isgt : E E → B`

  returns `true` if the value of the first expression is higher than the value of the second expression, otherwise `false`. Both arguments must be constant expressions, otherwise an error messages is given and `false` is returned.

- **tdl_isgeq**

  `tdl_isgeq : E E → B`

  returns `true` if the value of the first expression is higher than or equal to the value of the second expression, otherwise `false`. Both arguments must be constant expressions, otherwise an error messages is given and `false` is returned.

- **tdl_isneq**

  `tdl_isneq : E E → B`

  returns `true` if the values of both expressions are not the same, otherwise `false`. Both arguments must be constant expressions, otherwise an error messages is given and `false` is returned.

- **tdl_isdummy**

  `tdl_isdummy : E → B`

  returns `true` if variable expression `E` is a a dummy argument variable, otherwise `false`. The argument must be a variable expression, otherwise an error message is given and `false` is returned.

- **tdl_iscommon**

  `tdl_iscommon : E → B`

  returns `true` if expression `E` is a member of a COMMON block, otherwise `false`. The argument must be a variable expression, otherwise an error message is given and `false` is returned.

- **tdl_isconst**

  `tdl_isconst : E → B`

  returns `true` if expression `E` is a constant expression, otherwise `false`.

- **tdl_isscalar**

  `tdl_isscalar : E → B`

  returns `true` if expression `E` is a scalar variable, otherwise `false`. The argument must be a variable expression, otherwise an error message is given and `false` is returned.

- **tdl_replace**

  `tdl_replace : S E E → S`

  replace each occurence of the second argument in the first argument with the third argument.

- **tdl_intvar**

  `tdl_intvar : E → E`

  create a local integer variable `IV_`x where x is the constant integer argument. If the argument is not a constant integer argument, an error is given. If the variable already exist, an error message is given also.

- **tdl_realvar**

  `tdl_realvar : E → E`

  create a local real variable `IV_`x where x is the constant integer argument. If the argu-

ment is not a constant integer argument, an error is given. If the variable already exist, an error message is given also.

- **tdl_mod**

  `tdl_mod : E E → E`

  generates a call to intrinsic function `MOD` with both arguments.

- **tdl_min**

  `tdl_min : E E → E`

  generates a call to intrinsic function `MIN` with both arguments.

- **tdl_max**

  `tdl_max : E E → E`

  generates a call to intrinsic function `MAX` with both arguments.

2.3 Incorporation of TDL in MT1

In this section we briefly descirbe how the TDL can be invoked from within MT1. First, a file containing a collection of `IMPORT` and `TRANSFORM` statements is read in. Then these transformations can be applied interactively. MT1 comes with a default strategy for applying transformations. This default strategy is described below. As mentioned in the Introduction, a Strategy Specification Language has been implemented in the MT1 system. This language is described in the Oceans Deliverable D1.2a.

The commands to be given to MT1 for applying transformations are the following.

Readtrf The **readtrf** command reads in a transformation file. The name of the tranformation file must be supplied as an argument. If the environment variable 'CPP' is set to a filter, **readtrf** feeds the transformation file to the filter and then parses the output of the filter.

Start The **start** command starts the application of transformations to the Fortran program. If a fragment of the program matches the lefthand-side of a transformation and the condition of the transformation evaluates to true, the matched and the new fragment, described by the righthand-side of the transformation, are displayed and you are asked for conformation to apply the transformation by a prompt

```
** ACCEPT (y/n/q/e/s) ===>
```

Five answers (either in uppercase or in lowercase) are allowed:

- 'y' - yes, apply this transformation and proceed searching with this transformation;
- 'n' - no, do not apply this transformation but proceed searching with this transformation;
- 'q' - quit, do not apply this transformation but proceed searching with the next transformation starting at the beginning of the program unit;
- 'e' - exit, do not apply this transformation but return to the command prompt;
- 's' - skip program unit, do not apply this transformation but restart the transformation phase on the next program unit.

The answer should be terminated by a carriage-return. If only a carriage-return is entered, the default answer is the first one listed ('y').

The default strategy for applying transformations can be described by the following steps:

1. Select the first program unit textually occuring in the Fortran input.
2. Select the first transformation textually occuring in the transformation file.
3. Search through the program unit for a fragment matching the lefthand-side of the transformation.
4. If there's no such fragment goto step 5, otherwise ask for conformation. If the answer is
   - 'y' apply the transformation and proceed with step 3;
   - 'q' proceed goto step 5;
   - 'e' return to the command prompt;
   - 's' goto step 6.
   - otherwise, proceed with step 3;
5. Select the next transformation textually occuring in the transformation file. If there are no more transformations, proceed with the next step, otherwise goto step 3.
6. Select the next program unit textually occuring in the Fortran input. If there are no more program units, return to the command prompt,

Auto The **auto** command applies all matching transformations without asking for conformation to the user. The same strategy is used for the **auto** command as the strategy for the **start** command. In case of the default strategy, step 4 in this strategy should be replaced by

4. If there's no such fragment proceed with step 5, otherwise apply the transformation and goto step 3.

Query If determination of the exact solutions is not feasible or possible, dependence analysis must result in conservative estimates of dependences. This is done to prevent application of transformations that change the semantics of the program, even if this causes some valid applications to be overlooked. Therefore, the command **query** has been implemented in MT1 to toggle between two modes in which either all dependences are assumed to hold or a mode in which dependences can be ignored. In the latter mode, the compiler prompts all dependences causing a 'dep' condition to hold on a match. The user can instruct the compiler to ignore certain dependences, if it is known that these dependences do not actually hold, which might disable or enable further application. In the following fragment, for example, the occurrence of complex subscript '`IND(I)`' results in the recording of the loop-carried dependence $S_1 \delta^o_< S_1$, which prevents concurrentization. However, if array `IND` contains a permutation of the index values, concurrentization is valid. After reply 'y', the transformation is enabled:

```
** MATCH ON
      L1: DO I = 1, 100, 1
        S2: A(IND(I)) = B(I)
      ENDDO
      .....
      ** Dependence S2 d-outp < S2 A: IGNORE (n/y/q)=> y
** TRANSFORM INTO
      L5: DOALL I = 1, 100, 1
        S6: A(IND(I)) = B(I)
      ENDDOALL
      .....
      ** ACCEPT (y/n/q/e) ===>
```

2.4 Syntax of the Transformation Definition Languge

This section shows the syntax rules for the transformation language. Lowercase words are non-terminal tokens, while uppercase words and single quoted strings are terminal tokens. The uppercase tokens may be instantiated as follows:

- `ID` an identifier consisting of lower and uppercase letters, digits and underscores. The identifier must start with zero or more underscores followed by a letter;

- `SLID` the name of a shared library file (optionally preceded by a path). The name of the file must either have the extension '.sl' or '.so'.
- `REALCONST` any floating point number;
- `INTCONST` any integer value;
- `BOOLCONST` either '.true.' or '.false.';
- `STMTVAR` a string '!s' followed by an integer;
- `EXPVAR` a string '!e' followed by an integer.

The empty rule is denoted by `<empty>`. Here are the syntax rules in BNF-form.

```
definitions -> definitions definition

definition -> udf_def | trafo_def

udf_def -> 'import' decls 'from' sl_libs ';'

decls -> decl | decls decl

decl -> ID ':' arg_list '->' type

type -> 'S' | 'B' | 'E'

arg_list -> <empty> | arg_list type

sl_libs -> sl_id | sl_libs sl_id

sl_id -> '"' SLID '"' | '<' SLID '>'

trafo_def -> 'transform' pattern 'into' xpattern 'condition' conditions ';'

pattern -> 'list' '(' stmt ',' xpattern ')'
         | STMTVAR
         | 'follow' '(' STMTVAR ',' pattern ')'
         | 'merge' '(' xpattern ',' xpattern ')'
         | 'nil'
```

```
stmt -> 'do' '(' xexp ',' xexp ',' xexp ',' xexp ',' xpattern ')'
      | 'doall' '('  xexp ',' xexp ',' xexp ',' xexp ',' xpattern ')'
      | 'assign' '('  xexp ',' xexp ')'
      | 'if' '(' xexp ',' xpattern ')'

xpattern -> pattern | func

exp -> EXPVAR
     | xexp '+' xexp
     | xexp '-' xexp
     | xexp '*' xexp
     | xexp '/' xexp
     | xexp '**' xexp
     | xexp '.eq.' xexp
     | xexp '.ne.' xexp
     | xexp '.ge.' xexp
     | xexp '.gt.' xexp
     | xexp '.le.' xexp
     | xexp '.lt.' xexp
     | xexp '.eqv.' xexp
     | xexp '.neqv.' xexp
     | xexp '.and.' xexp
     | xexp '.or.' xexp
     | '.not.' xexp
     | '(' xexp ')'
     | '-' xexp
     | INTCONST
     | REALCONST
     | BOOLCONST
     | 'vectorize' '(' EXPVAR ',' EXPVAR ',' xexp ':' xexp ':' xexp ')'

xexp -> exp | func

conditions -> condition | condition 'and' conditions

condition -> 'true'
           | 'dep' depkind dirvec '(' s_indic ',' s_indic ')' onclause
           | 'isnil' '(' STMTVAR ')'
           | 'issub' '(' EXPVAR ',' EXPVAR ')'
           | 'not' condition
```

```
            | func

depkind -> 'flow' | 'anti' | 'output' | 'input' | '@'

onclause -> <empty> | '>' EXPVAR

dirvec -> <empty> | dir dirvec

dir -> '=' | '<' | '>' | '*'

s_indic -> '$' attribs

attribs -> '.' 'tail' attribs
          | '.' 'dobody' attribs
          | '.' 'ifbody' attribs
          | '.' 'follow' attribs
          | '.' 'head'
          | <empty>

func -> ID '(' act_arg_list ')'

act_arg_list -> <empty> | act_args

act_args -> act_args ',' act_arg

act_arg -> exp | pattern | func | s_indic
```

2.5 Example transformations

In this section we present the TDL formulation of all commonly used loop level transformations as can be found in e.g. [Pol88, Wol96, Wol91, ZC90]. This section serves to show the expressive power of the TDL defined above.

```
%
% Loop collapsing
%

transform
    list(do(!e1, 1, !e3, 1,
         list(do(!e5, 1, !e7, 1, !s1), nil)), !s2)
```

```
into
    list(do(tdl_intvar(1), 1, !e3 * !e7, 1,
         list(assign(!e1, ((tdl_intvar(1) - 1) / !e7) * !e7 + 1),
         list(assign(!e5, tdl_mod(tdl_intvar(1) - 1, !e7) + 1),
         !s1))), !s2)
condition
    tdl_isint(!e3) and
    tdl_isint(!e7)
;


%
% Loop Distribution
%

transform
     list(do(!e1, !e2, !e3, !e4, follow(!s1,!s2)),!s3)
into
     list(do(!e1, !e2, !e3, !e4, !s1),
        list(do(!e1, !e2, !e3, !e4, !s2), !s3))
condition
      not isnil (!s2) and
      not dep @ < ($.dobody.follow, $.dobody)
;



%
% Loop fusion
%

transform
    list(do(!e1, !e2, !e3, !e4, !s1),
    list(do(!e1, !e2, !e3, !e4, !s2), !s3))
into
    list(do(!e1, !e2, !e3, !e4, merge(!s1,!s2)), !s3)
condition
    not dep @ ($.dobody, $.tail.dobody)
;


%
% Loop interchange
%

transform
    list(do(!e1, !e2, !e3, !e4,
         list(do(!e5, !e6, !e7, !e8, !s1), nil)), !s2)
into
```

```
    list(do(!e5, !e6, !e7, !e8,
         list(do(!e1, !e2, !e3, !e4, !s1), nil)), !s2)
condition
    not dep @ <> ($.dobody.dobody, $.dobody.dobody) and
    not dep flow <= ($.dobody.dobody, $.dobody.dobody) and
    not dep flow ($.head, $.dobody) > !e1
;


%
% Transformations to normalize loop with non-unit stride.
%

#include <tdl.i>

%
% Turn a loop with a negative stride into a loop with a stride with
% a positive stride
%

transform
    list(do(!e1, !e2, !e3, !e4, !s1), !s2)
into
    list(do(!e1, !e3, !e2, -!e41,
            tdl_replace(!s1, !e1, !e2 + !e3 - !e1)), !s2)
condition
    tdl_isconst(!e4) and
    tdl_isint(!e4) and
    tdl_islt(!e4, 0)
;

%
% Turn a loop with a non-unit stride into a loop with
% with lowerbound 1 and unit stride.
%

transform
    list(do(!e1, !e2, !e3, !e4, !s1), !s2)
into
    list(do(!e1, 1, (!e3 - (!e2 - !e4)) / !e4, !e4), 1,
            tdl_replace(!s1, !e1, !e2 - !e4 + !e4 * !e1)), !s2)
condition
    tdl_isint(!e2) and
    tdl_isint(!e3) and
    tdl_isint(!e4) and
    tdl_isconst(!e4) and
    tdl_isneq(!e4, 1)
;
```

```
%
% Loop peeling
%

#include <tdl.i>

transform
    list(do(!e1, !e2, !e3, !e4, !s1),
    list(do(!e1, !e5, !e3, !e4, !s2), !s3))
into
    merge(tdl_replace(!s1, !e1, !e2),
          list(do(!e1, !e2 + 1, !e3, !e4, !s1),
          list(do(!e1, !e2 + 1, !e3, !e4, !s2), !s3)))
condition
    tdl_iseq(!e5 - !e2, 1)
;

%
% Loop reversal
%

#include <tdl.i>

transform
    list(do(!e1, !e2, !e3, !e4, !s1), !s2)
into
    list(do(!e1, !e3, !e2, -!e4, tdl_replace(!s1, !e1, !e2 + !e3 - !e1)), !s2)
condition
    tdl_isconst(!e4) and
    tdl_islt(!e4, 0)
;
import
    tdl_intvar: E -> E
    tdl_isconst: E -> B
from
    "./src/libtdl.sl"
;

transform
    list(do(!e1, !e2, !e3, !e4,
follow(!s1, list(assign(!e5, !e6*!e1), !s2))), !s3)
into
    list(assign(tdl_intvar(1), !e2 * !e6),
         list(do(!e1, !e2, !e3, !e4,
 merge(!s1, list(assign(!e5, tdl_intvar(1)),
         merge(!s2, list(assign(tdl_intvar(1), tdl_intvar(1) +
                                               !e6),nil))))),!s3))
condition
```

```
    tdl_isconst(!e6)
;

%
% Unroll zero trip loops
%

#include <tdl.i>

transform
    list(do(!e1, !e2, !e3, !e4, !s1), !s2)
into
    !s2
condition
    tdl_isint(!e2) and
    tdl_isint(!e3) and
    tdl_isint(!e4) and
    tdl_isleq(((!e3 - !e2 + !e4) / !e4), 0)
;
%
% Unroll one trip loops
%

#include <tdl.i>

transform
    list(do(!e1, !e2, !e2, !e3, !s1), !s2)
into
    merge(tdl_replace(!s1, !e1, !e2), !s2)
condition
    true
;
%
% Loop unrolling
%

#include <tdl.i>

transform
    list(do(!e1, !e2, !e3, 1, !s1), !s2)
into
    list(do(!e1, !e2, (!e2 - 1) + ((!e3 - !e2 + 1) / 2) * 2, 2,
            merge(!s1, tdl_replace(!s1, !e1, !e1 + 1))),
    list(if(((!e3 - !e2 + 1) / 2) * 2 .neq. (!e3 - !e2 + 1),
         tdl_replace(!s1, !e1, !e3)), !s2))
condition
    tdl_isint(!e2) and
    tdl_isint(!e3) and
    not tdl_isconst(!e2)
```

```
;

transform
    list(do(!e1, !e2, !e3, 1, !s1), !s2)
into
    list(do(!e1, !e2, (!e2 - 1) + ((!e3 - !e2 + 1) / 2) * 2, 2,
            merge(!s1, tdl_replace(!s1, !e1, !e1 + 1))),
    list(if(((!e3 - !e2 + 1) / 2) * 2 .neq. (!e3 - !e2 + 1),
         tdl_replace(!s1, !e1, !e3)), !s2))
condition
    tdl_isint(!e2) and
    tdl_isint(!e3) and
    not tdl_isconst(!e3)
;

transform
    list(do(!e1, !e2, !e3, 1, !s1), !s2)
into
    list(do(!e1, !e2, (!e2 - 1) + ((!e3 - !e2 + 1) / 2) * 2, 2,
            merge(!s1, tdl_replace(!s1, !e1, !e1 + 1))), !s2)
condition
    tdl_isint(!e2) and
    tdl_isint(!e3) and
    tdl_isconst(!e2) and
    tdl_isconst(!e3) and
    tdl_iseq(((!e3 - !e2 + 1) / 2) * 2, (!e3 - !e2 + 1))
;

transform
    list(do(!e1, !e2, !e3, 1, !s1), !s2)
into
    list(do(!e1, !e2, !e3 - 1, 2,
            merge(!s1, tdl_replace(!s1, !e1, !e1 + 1))),
         merge(tdl_replace(!s1, !e1, !e3), !s2))
condition
    tdl_isint(!e2) and
    tdl_isint(!e3) and
    tdl_isconst(!e2) and
    tdl_isconst(!e3) and
    tdl_isneq(((!e3 - !e2 + 1) / 2) * 2, (!e3 - !e2 + 1))
;

%
% loop vectorization
%

transform
    list(do(!e1, !e2, !e3, !e4,
         list(assign(!e5, !e6), nil)), !s1)
```

```
into
    list(assign(vectorize(!e5, !e1, !e2:!e3:!e4),
                vectorize(!e6, !e1, !e2:!e3:!e4)), !s1)
condition
    issub(!e1, !e5) and
    not dep flow < ($.dobody, $.dobody)
;
```

Chapter 3

Strategy Specification Language

In this chapter we discuss the syntax and semantics of the Strategy Specification Language. We also discuss how this language is incorporated in the MT1 compilation system. Finally, we present a novel strategy for transforming imperfectly nested loops.

In the initial implementation of MT1, the order in which the transformations were applied, was specified by the order in which the transformations appeared in the transformation definition file. Each transformation was taken in turn and applied to every statement in the current Fortran unit. If one such pass caused the Fortran code to be changed, another pass with the same transformation was executed on the same piece of code. Only when a certain pass did not change anything, the next transformation was taken and applied in the same way.

In this section a Strategy Specification Language (SSL) is presented, that allows the specification of the order in which the transformations from the transformation definition file are to be applied. The language implicitly defines the set of statements against which the transformations are applied. This construction allows the specification of an optimizing strategy to be at a much higher abstract level than the source code level as in most other compilers.

In Section 3.1 de transformation engine of MT1 is described. Next, Section 3.2 describes the syntax of our prototype SSL. Section 3.3 then describes the way MT1 interprets the SSL using the transformation engine.

3.1 The original MT1 transformation engine

This section first describes the transformation engine as it was originally implemented in the MT1 system. These issues are dealt with, because they are essential to describe strategy execution. For a full discussion of the Transformation Definition Language and the original

mechanism for applying transformations, see Oceans Deliverable D1.1 [BBK+97]. In the next section the alterations to the transformation application engine to include strategies are described.

Transformations in MT1 consist of an input pattern against which FORTRAN statements are matched, conditions checking various properties of the matched code, and an output pattern which is used to derive the new code from. So the basic syntax for specifying transformations can be given as follows.

```
trafo     := TRANSFORM input_pattern
             INTO output_pattern
             CONDITION conditions;
```

When a FORTRAN program is read in by MT1, it is split into separate units each containing one subroutine. MT1 then applies each transformation on each unit separately.

The application of a transformation consists of iterating over all FORTRAN statements in the associated scope of the transformation, trying to match the input pattern of the transformation against a block of statements starting with the current statement. The notion of the associated scope of a transformation will be explained later. For now, consider the scope of any transformation to be the entire unit. This means that the transformations are applied to every statement in the current unit. Note that the input pattern of a transformation needs to be matched against statements which are within its scope. During matching, the statement- and expression-variables are bound to matching fragments of FORTRAN code.

After having matched the input pattern, MT1 checks whether the conditions specified in the transformation hold. To do this, it uses the bindings of the variables to test the original code for the properties specified in the condition of the transformation.

If the conditions hold, the output pattern and the variable bindings are used to compute the resulting code. After replacing the original code with the newly computed code, MT1's internal data structures are updated to reflect the new code, after which the execution of the strategy continues. Note that the next application of a transformation may thus work on the result of a previous transformation.

3.2 Strategy syntax

In this section we give the syntax of the Transformation Definition Language. The semantics of these constructs is given in the next section. A strategy consists of a (possibly empty) list of semicolon (;) separated SSL statements:

```
strategy   := ssl_stats;
ssl_stats  := ssl_stat ';' ssl_stats | ;
```

An SSL statement can be a single transformation, a conditional statement, or one of two repetitive statements.

```
ssl_stat   := seq_stat | if_stat | while_stat | until_stat;
seq_stat   := trafo_id | roll_back_stat;
```

An IF-statement consists of a transformation that acts as a condition, a THEN-part and an optional ELSE-part. The transformation in the condition can applied successfully or not. If it is, the transformations in the THEN-part are to be executed. Optionally, in the ELSE-part a statement list can be given which should be executed in case the transformation matched but was not applied successfully due to failing conditions.

```
condition  := trafo_id;
if_stat    := IF condition THEN ssl_stats elsepart ENDIF;
else_part  := ELSE ssl_stats | {empty} ;
```

The two repetitive constructs consist of a transformation to be checked and a statement list to be executed if the condition is true or false, respectively.

```
while_stat := WHILE condition ssl_stats ENDWHILE;
until_stat := UNTIL condition ssl_stats ENDUNTIL;
```

The language contains a means for applying sequences of transformations only if they can all be applied, by means of the `roll_back_stat`.

```
roll_back_stat := trafo_id AND trafo_id | trafo_id AND roll_back_stat
```

3.3 Semantics

This section describes the semantics of the Strategy Specification Language. We will describe what actions are undertaken by MT1

- when a certain transformation did not match (either because MT1 found no match or because the user chose to ignore the match),
- when the conditions were not satisfied (or the user chose not to apply an applicable transformation),

- and when a transformation is successfully applied.

The interaction with the user has undergone some modifications over the original system. The system no longer checks whether the application of a transformation has changed anything in order to initiate another pass. Every transformation is applied only once to every statement. The user has to indicate whether to accept or ignore a match and whether to really apply a transformation.

When MT1 has found a match between the current transformation and a block of FORTRAN statements, beginning with the current statement, MT1 reports this fact by showing the matching code.

The conditions specified in the transformation are then checked. If the conditions are not satisfied, the user is prompted as follows:

```
** Condition not satisfied. Accept match (y/n)
```

If the user does not accept the match, MT1 will continue as if the match was not found. If the user does accept the match, MT1 will consider the result of the application of the transformation to be false. The result of the application of a transformation is used when deciding the flow of control through the strategy. In particular, it is used to select the THEN- or the ELSE-part in a conditional construct.

If the conditions are satisfied, MT1 computes and shows the new code fragment and prompts the user as follows:

```
** ACCEPT (y/n/i) ==>
```

If the user accepts, the matching code is replaced by the new code fragment. The result of the application of the transformation is considered to be true. In this case, we consider the transformation to be applied successfully. If the user replies 'n', the match is considered to be accepted, but the conditions are considered not to be satisfied. As in the case where MT1 decided the conditions not to be satisfied and the user accepted the match, the result of the transformation is considered to be false. If the user replies 'i', the match is ignored.

When we describe the semantics of the various constructs in our SSL, we will indicate which actions are undertaken by MT1 in case a certain transformation did not match, in case the conditions were not fulfilled, and in case the transformation was successfully applied. As described above, the user may choose to ignore a match if MT1 found one. The user may also choose to indicate the conditions not to be satisfied, even if MT1 found them to be true. MT1 will respond to these choices in the same way it would in case it itself did not find a match or found the conditions not to be satisfied, respectively.

The possibility for a user to ignore the matching of a certain transformation allows the user to influence the flow of control. If a certain transformation would match a FORTRAN loop, while the transformation is the condition of an IF-THEN-ELSE construct, matching of this transformation would never enter the loop, since after having executed the THEN- or ELSE-part, transformation would continue after the loop.

In order to describe the semantics, Section 3.3.1 first describes some notational conventions, and defines the level of transformations. There are four general constructs in our prototype SSL. These constructs indicate general sequential application (much like the original MT1 system), conditional application, and repetitive repetition, of which there are two forms. Each of these four constructs will be described in the next four sections.

3.3.1 General definitions

To describe these semantics we will use *trafo_n* to indicate a single transformation. To specify a list of transformations, possibly containing arbitrary complex strategy constructs, we will use the notation *ssl_stats_n*.

In the original MT1 system, all transformations were applied on every statement in the current unit. In the present context where we want to execute strategies, we need to reconsider to scope on which transformations act. For instance, in an IF-THEN-ELSE construct, we want the transformations in the THEN-part to act on the fragment selected by the condition. In order ot do this, MT1 implicitly associates a list of statements to each transformation. We will call this list of statements the *associated scope of a transformation*. The transformations are only applied to the statements in its scope. MT1 derives the scope of a transformation from its location in the strategy and the contents of the current unit.

In order to be able to derive the scope of a certain transformation, we first need to define the level of a transformation. To do this, we will first define the level of SSL statements. The level of an SSL statement is essentially its nesting depth. SSL statements which are not located in a body of another SSL statement are at level one. The level of an SSL statements located in the body of another SSL statement is one higher than the level of its containing construct. Using the level of SSL statements we can formulate the level of the individual transformations in these statements to be the same as the level of these statements. Hence the transformations in the THEN-part of a conditional SSL statement have a level one higher than the condition of this conditional statement.

The scope of transformations at level one is the entire unit under consideration. The scope of transformations at higher levels can nformally be given as:

- if the transformation is located in the THEN-part of a conditional SSL statement, its associated scope is the fragment that resulted from the application of the condition

transformation;

- if the transformation is located in the ELSE-part of a conditional SSL statement, its associated scope is the fragment that matched the condition transformation but did not satisfy the conditions of this transformation;
- if the transformation is located in the body of a WHILE-statement, its associated scope is the fragment that resulted form the application of the condition transformation in the WHILE-statement;
- if the transformation is located in the doby of a REPEAT-statement, its associated scope is the fragment that matched the condition transformation but did not satisfy its conditions.

The precise association of scopes to transformations in a strategy is described below in the semantics of the various constructs.

3.3.2 Sequential application

The simplest strategy consist of a list of transformations to be applied one after the other:

```
trafo_1;
ssl_stats_1;
```

MT1 will first try to apply *trafo_1* on all statements in its associated scope. If the input pattern of this transformation does not match a certain statement, the next statement is considered. If a certain statement does match, but the conditions do not hold, the application of transformations continues with the next statement as well. If the input pattern matched and the conditions hold, the code is replaced. Transformation then continues with the statement that follows the last statement in the match. When application of *trafo_1* reaches the last statement of its associated scope, application of the sequential construct continues with the first statement in *ssl_stats_1*. The associated scope of transformations in *ssl_stats_1* at the same level as *trafo_1* all have the same associated scope as *trafo_1*. Hence application of the transformations in *ssl_stats_1* starts with the first statement in this scope.

3.3.3 Conditional application

With the *IF-THEN-ELSE-ENDIF* construct it is possible to apply certain transformations conditionally. Whether the transformations in the THEN- or ELSE-part are to be applied is dependent on the result of the transformation in the condition:

```
IF trafo_1 THEN
    ssl_stats_1;
ELSE
    ssl_stats_2;
ENDIF;
```

If a certain statement does not match the input pattern of *trafo_1* the next statement is tried. If a statement does match the input pattern, the conditions are checked. Depending on whether the conditions do or do not hold *ssl_stats_1* or *ssl_stats_2* is applied, respectively.

The statements in *ssl_stats_1* and *ssl_stats_2* are one level higher that *trafo_1*. The associated scope of statements at a higher level is always refined. The associated scope of the transformations in *ssl_stats_1* consists of all statements inserted by *trafo_1*. That is, the transformed code fragment resulting from the application of *trafo_1*. The associated scope of the transformations in *ssl_stats_2* consists of all statements in the match of *trafo_1*.

After having applied one of the statement lists, application continues on the statement following the last statement in the match of the previous application of *trafo_1*.

3.3.4 Repetitive application: WHILE

It is possible to apply a list of transformations repeatedly while a certain transformation can be applied successfully.

```
WHILE trafo_1
    ssl_stats_1;
ENDWHILE;
```

If a certain FORTRAN statement matches the input pattern of *trafo_1* and the conditions are satisfied, the transformations in *ssl_stats_1* are applied. The associated scope of the transformations in *ssl_stats_1* consists of the transformed fragment resulting from the successful application of *trafo_1*.

After having applied all transformations in *ssl_stats_1*, the WHILE constructs starts over and *trafo_1* is applied again. Since we want the WHILE-construct to repeatedly act on a single program fragment, *trafo_1* will be matched against the same fragment as it has matched the first time. That is, its associated scope is refined to be the same as the associated scope of the transformations in *ssl_stats_1*. Because the transformation in the condition need not match the first statement in its associated scope, it will search its scope to find a fragment against which it can be matched. Therefore, this scope may be further refined on each iteration. The end of its scope is given by the first statement after the first match of *trafo_1*.

When the conditions of *trafo_1* are not satisfied, transformation continues with the statement following the last statement in the match and with *trafo_1*.

3.3.5 Repetitive application: REPEAT

It is also possible to apply certain transformations as long as a certain transformation cannot be applied, because its conditions are not fullfilled.

```
UNTIL trafo_1
    ssl_stats_1;
ENDUNTIL;
```

The transformations in *ssl_stats_1* are executed if *trafo_1* matched, but its conditions were not satisfied. The associated scope of the transformations in *ssl_stats_1* consists of all statements in the match of *trafo_1*.

When all transformations in *ssl_stats_1* have been applied, transformation continues with *trafo_1*. Its associated scope is refined to the associated scope of the transformations in *ssl_stats_1*. Again the associated scope of *trafo_1* may be refined on each iteration.

When the conditions in *trafo_1* are satisfied, transformation continues on the statement following the last statement in the match of *trafo_1*.

3.3.6 Roll back construct

When writing a strategy, we may want to try out a certain transformation, which should enable another transformation. If the second transformation cannot be applied, we may want to undo the first transformation. such strategy cannot be expressed in our SSL. The most intuitive way to express such a construct is by grouping certain transformations together:

```
trafo_1 and trafo_2;
```

In this construct, *trafo_1* will be applied on a certain fragment of FORTRAN code. If this transformation is successfully applied, *trafo_2* will be applied on the resulting code from *trafo_1*. That is, the associated scope of *trafo_2* is the resulting fragment from *trafo_1*. If *trafo_2* fails to be applied, the effect of *trafo_1* is rolled back: the entire construct fails and no changes are made to the program under consideration.

We allow for an arbitrary number of transformations to be grouped together in a roll back construct. If any of these transformations fails, the entire construct fails and no changes are made to the program under consideration.

If all transformations in the group have been succesfully applied, the result is rolled forward. This means that the fragment that matched *trafo_1* is replaced in the program under consideration by the result of all transformations on this fragment.

In case such a group of transformations appears as the condition of a conditional or repetitive construct, the result would be true only if all transformations in the group were applied successfully.

Chapter 4

Conclusion

In this report we have described a specification language for program transformations on the Fortran 77 source language level. Together with the strategy specification language this specification mechanism allows for a flexible and highly tunable set of transformations to be applied to a program. These transformations are structured around a pattern match mechanism that allows for user defined function that can access the internal program representation directly. Using this mechanism we may specify an input pattern containing meta-variables that is matched against the source program thereby binding these meta-variables to actual expressions and statements. An output pattern can be specified using these meta-variables. Based on this output pattern and the actual bindings of the meta-variables, a new program fragment is constructed that will replace the fragment that was matched by the input pattern. We may also specify conditions under which the transformation may be applied. These conditions typically check for the legality of the transformation. However, the conditon may also contain user defined functions that may inspect arbitrary auxilary data structures. Using this mechanism, a hook is provided for the feedback from low to high level. This feedback can be stored in an auxilary data structure and inspected by user defined functions in the condition.

Next we have discussed our notions of Strategy Specification Language. The SSL is a specification mechanism to sequence elementary transformations. The SSL on the source language level is capable of the sequential composition of transformations, a conditional and a repetetive construct. The conditions for these last constructs consist of the success or failure of some arbitrary condition transformation. In this way, enabling transformations can be applied before the main transformation. Also, since the condition of a transformation may contain user-defined functions that may access an arbitrary data structure, a mechanism is provided for feedback. This feedback information may be targetted towards specific parts of the program, like a particular loop. Now the 'enabling' transformation may consist of a match against an arbitrary loop, transforming that loop into the same loop and having the condition check the

identity of the loop. Only the sought after loop, the number of which is given in the feedback information, will match this condition that checks for this number.

Bibliography

[BBK+97] A.J.C. Bik, P.J. Brinkhaus, P.M.W. Knijnenburg, P. Touber, and H.A.G. Wijshoff. Transformation Definition Language. Oceans Deliverable D1.1, 1997. Available through `www.wi.leidenuniv.nl/~peterk`.

[Bik92] Aart J.C. Bik. A prototype restructuring compiler. Master's thesis, Utrecht University, 1992. INF/SCR-92-11.

[Bri93] Peter Brinkhaus. Compiler analysis of procedure calls. Master's thesis, Utrecht University, 1993. INF/SCR-93-13.

[BW93] A.J.C. Bik and H.A.G. Wijshoff. MT1: A prototype restructuring compiler. Technical Report no. 93-32, Department of Computer Science, Leiden University, 1993.

[Pol88] C. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Boston, 1988.

[vDdVW+95] E. van Dis, R.D. de Vreugd, A.P. Wulms, P. Brinkhaus, and P.M.W. Knijnenburg. A vector transformation library. Technical Report no. 95–29, Department of Computer Science, Leiden University, 1995.

[Wol91] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, 1991.

[Wol96] M.J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

[ZC90] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.