DIPLOMA THESIS

The WOOP Preprocessor An Implementation of Discrete Loops in Ada95.

carried out at the Department of Automation

of the Vienna University of Technology

under guidance of

Univ.Doz. Dr. Ulrich Schmid

and

Univ.Ass. Dr. Johann Blieberger

by

Bernd Burgstaller

Matr.Nr. 8925663

Mitterberg 11
8665 Langenwang

November 29, 1996

WIDMUNG

Eines Nachts hatte ich einen Traum-
mir träumte, daß ich mit dem HERRN am Ufer des Meeres entlangging.
Am Himmel flammten Szenen aus meinem Leben auf.
Bei jeder Szene entdeckte ich zwei Paar Fußabdrücke im Sand,
ein Paar von mir, das andere vom HERRN.
Als die letzte Szene aufflammte,
sah ich mich um nach meinen Fußspuren im Sand.
Ich bemerkte, daß oftmals auf meinem Lebenspfad
nur eine Fußspur zu sehen war.
Und es fiel mir auf, daß dies immer während der
dunklen und traurigen Zeiten meines Lebens geschehen war.
Dies bewegte mich sehr, und ich fragte den HERRN,
weshalb das so sei.
"HERR, als ich mich entschloß, dir nachzufolgen,
versprachst du mir, meinen ganzen Weg mit mir zu gehen.
Nun habe ich aber bemerkt, daß in den schwersten Zeiten
meines Lebens nur ein Paar Fußabdrücke zu sehen ist.
Ich verstehe nicht, warum du mich allein gelassen hast,
als ich dich am allermeisten nötig hatte." Der HERR antwortete:
"Mein teures, liebes Kind,
ich liebe dich und würde dich nie, nie allein gelassen haben
während den Zeiten des Leidens und der Anfechtung.
Wenn du nur ein Paar Fußabdrücke gesehen hast,
so war das deshalb,
weil ich dich getragen habe."

Diese Arbeit widme ich meinen Eltern, die mich fortwährend unterstützten und mir damit erst dieses Studium ermöglicht haben.

In Dankbarkeit,
Euer Sohn Bernd.

ACKNOWLEDGMENTS

I wish to thank Johann Blieberger, my supervisor, for his constant enthusiasm and guidance during the course of this work.

I would also like to thank my WOOP team-mates for their interesting comments and insights and for the fun we had, in particular: Bernhard Scholz, Roland Lieger, Astrid Koizar, and Mario Weilguni.

Thanks also to the GNAT team, especially Prof. Robert Dewar, for providing insight on some of the more sophisticated details of GNAT.

I also want to thank my brother Jörg for the countless hours and days we spent biking, skiing and hiking in the mountains of Styria.

Finally a big thank you to everyone at Haus Technik for putting up with my drumming during all those years.

Abstract

The WOOP Preprocessor
An Implementation of Discrete Loops in Ada95.

by Bernd Burgstaller

WOOP, as an acronym for *Worst-Case Performance of Object-Oriented Programs,* is a research project at the Vienna University of Technology that is aimed at the determination of the timing behavior of software for real-time systems[1]. One of WOOP's early achievements was the introduction of the so-called *discrete loop statement [Bli94]*, a loop construct which narrows the gap between general loops (e.g. repeat, while) and for-loops. Like for- but contrary to while-loops, discrete loops are known to complete in any case, but they provide much greater semantic power. Furthermore it is possible to determine the number of iterations of a discrete loop, which can be an extremely difficult task with general loops.

In order to utilize this new and up to this point solely theoretical concept for a real-world programming environment, it was necessary to provide a translation mechanism that would enable programmers to use discrete loops in their code. Although in principle there was no cause to favor any particular programming language, it was the reliability and maintainability of *Ada95* that finally lead to the decision to build a preprocessor capable of translating discrete loop augmented Ada95 code into standard Ada95 in a way that would preserve the *semantics* of discrete loops.

The goal of this thesis is to present the implementation of this preprocessor. It is in fact built on top of the source code of GNAT, the Gnu Ada Translator[2], because its analytical tasks blend well with the analysis that GNAT performs on its input. Calculations aimed at the estimation of upper bounds for discrete loops are carried out with help of Mathematica[3], a commercial computer algebra package. Parts of this preprocessor are therefore written in Mathematica's internal programming language. An Ada to C interface has also been implemented in order

[1]Project WOOP is supported by the Austrian Science Foundation (FWF) under grant P10188-MAT.

[2]GNAT is an Ada95 compiler that has been developed at New York University and its source code is distributed under terms of the GNU General Public License ([GPL91]).

[3]Mathematica is a trademark of Wolfram Research

to call Mathematica's kernel from within this preprocessor. Calculated upper bounds are inserted into the generated code and represent viable information for determining a program's worst-case timing behavior.

Note To The Reader On Reading Mathematical Equations

Despite the well-known fact that every mathematical formula cuts down the readership by half, I had to resort to their use at some places in this book. If you are a reader who finds any formula intimidating (and most people do), then I recommend a procedure that I normally adopt myself when such an offending line presents itself. The procedure is, more or less, to ignore that line completely and to skip over to the next actual line of text! Well, not exactly this; one should spare the poor formula a perusing, rather than a comprehending glance, and then press onwards. After a little, if armed with new confidence, one may return to that neglected formula and try to pick out some salient features. The text itself may be helpful in letting one know what is important and what can safely be ignored about it. If not, then do not be afraid to leave an offending formula behind altogether[PE].

Contents

Chapter 1
INTRODUCTION

The most significant difference between real-time systems and other computer systems is that the system behavior must not only be correct but the result of a computation must be available within a predefined deadline. It has turned out that a major progress in order to guarantee the timeliness of real-time systems can only be achieved if the *scheduling problem* is solved accordingly. Most scheduling algorithms assume that the runtime of a task is known a priori. Thus the *worst-case performance* of a task plays a crucial role. Determining the number of iterations of general loops is one of the most difficult tasks when estimating the timing behavior of a program. Most researchers try to ease the task of estimating the number of general loop iterations by *forbidding* general loops, i.e., by forcing the user to supply constant upper bounds for the number of iterations. Another approach is to let the user specify a time bound within which the loop has to complete. Project WOOP follows a different approach: The gap between general loops and for-loops is narrowed by defining *discrete loops*. These loops are known to complete and are easy to analyze (especially their numbers of iterations) and capture a large part of applications which otherwise would have been implemented by the use of general loops. These include *Heapsort*, a bottom-up version of *Mergesort* and *Euclid's Algorithm* to compute the greatest common divisor of two positive numbers. Furthermore all *divide and conquer* algorithms can be handled by discrete loops, e.g. *binary search* and tree traversing algorithms such as *weight-balanced trees* (BB[α]-trees) or AVL-trees.

1.1 Discrete Loops

The following is a brief summary on Discrete Loops that provides the necessary insight to understand their implementation. The exact theoretical treatment can be found in [Bli94].
The syntax of discrete loops is described using the notation in [Ada95](1.1.4).

In contrast to for-loops, discrete loops allow for a more complex dependency between two successive values of the loop-variable. In fact an arbitrary functional dependency between two successive values of the loop-variable is admissible, but this dependency must be constrained in order to ensure that the loop completes and to determine the number of iterations of the loop. Which values are assigned to the loop-variable is completely governed by the loop-body. The loop-header, however, contains a list of all those values that can possibly be assigned to the loop-variable during the next iteration. In fact each item of this list of values is a function of the loop-variable. A simple example is shown in Figure 1. In this example the loop-variable **K** will assume the values 1,2,4,8,16,32,64,.. until finally a value greater than **N** would be reached.

```
discrete K in 1 .. N new K := 2*K loop
--  loop body
end loop;
```

Figure 1: A simple example of a discrete loop

Of course the effect of this example can also be achieved by a simple for-loop, where the powers of two are computed within the loop-body. A more complex example is depicted in Figure 2. In this example the loop-variable **K** can assume the values 1,2,4,9,18,37,75,... until finally a value greater than **N** would be reached.

```
discrete K in 1 .. N new K := 2*K | 2*K+1 loop
--  loop body
end loop;
```

Figure 2: A more complex example of a discrete loop

But it is also possible that **K** follows the sequence 1,3,6,13,26,52,105,.... Here the same effect can not be achieved by a for-loop, because the value of the loop-variable cannot be determined exactly before the loop-body has been completely elaborated.

The reason for this is the *indeterminism* involved in discrete loops: Clearly the loop-body *determines* exactly which of the given alternatives is chosen, thus one can say that there definitely is no indeterminism involved. On the other hand, from an outside view of the loop one cannot determine which of the alternatives will be chosen, without having a closer look at the loop-body or without exactly knowing which data is processed by the loop. It is this "outside-view" indeterminism that is meant here. Furthermore this indeterminism enables us to estimate the number of loop iterations quite accurately without having to know all the details of the loop body.

1.1.1 Monotonical Discrete Loops

Monotonical Discrete Loops can be characterized best by the sequence of values the loop variable can take: if this sequence is strictly monotonically increasing (e.g. Figure 1), we speak of so-called monotonically increasing discrete loops. In the case of monotonically decreasing sequences we speak of monotonically decreasing discrete loops. The syntax of a monotonical discrete loop is given in conjunction with the syntax of *for*- and *while*-loops below.

```
loop_statement ::=
    [loop_simple_name:]
        [iteration_scheme] loop
            sequence_of_statements
        end loop [loop_simple_name];

iteration_scheme ::= while condition
    | for for_loop_parameter_specification
    | discrete discrete_loop_parameter_specification

for_loop_parameter_specification ::=
    identifier in [reverse] discrete_subtype_definition

discrete_loop_parameter_specification ::=
    identifier := initial_value in [reverse] discrete_subtype_definition
        new identifier := list_of_iteration_functions

list_of_iteration_functions ::=
    iteration_function { | iteration_function}

iteration_function ::= expression
```

Semantics of Monotonical Discrete Loops

For a loop with a **discrete** iteration scheme, the loop parameter specification declares a loop parameter, which is an object whose subtype is defined by the initial_value, the discrete_subtype_definition and the list_of_iteration_functions. Note that type information is collected from the list_of_iteration_functions although it is possible for the loop parameter to occur within the list_of_iteration_functions. The loop parameter is required to be of a *discrete* type.

Elaboration of the discrete_loop_parameter_specification creates the loop parameter and elaborates the inital_value and the discrete_subtype_definition. If the discrete_subtype_definition defines a subtype with a null range, execution of the loop is complete.

The optional keyword **reverse** defines a loop to be monotonically decreasing; if it is missing, the loop is considered to be monotonically increasing. Within the sequence of statements, the loop variable behaves like any other variable, i.e., it can be used on both sides of an assignment statement.

Before the sequence of statements is executed, the list of iteration functions is evaluated to produce a list of *possible successive values*. It is also checked whether all of these values are greater than the value of the loop variable if the keyword **reverse** is missing, or whether they are smaller than the value of the loop variable if **reverse** is present. If one of these checks fails, the exception **monotonic_error** is raised.

After the sequence of statements has been executed, it is checked whether the value of the loop variable is contained in the list of possible successive values. If this check fails, the exception **successor_error** is raised.

If the value of the loop variable is still within the discrete range stated in the loop header, the loop is iterated (at least) once more. If it is not within the range, the loop completes.

These semantics ensure that such a loop always completes, either because the value of the loop variable is outside the given range or because one of the above checks fails.

A lot of these runtime checks can be avoided by ensuring at compile time that the iteration functions are monotonical functions, or by means of data-flow analysis in order to make sure that **successor_error** will never be raised. Moreover we might even detect the number of iterations of the loop, which clearly depends on the initial_value of the loop variable, on the discrete_subtype_definition, and on the list_of_iteration_functions.

The following code shows an implementation of *Heapsort* using a discrete loop:

```
N : constant Positive := ??; -- Number of elements to be sorted.
subtype Index is Positive range 1 .. N;
type Sort_Array is array(Index) of Integer;

procedure Heapsort (Arr : in out Sort_Array) is

    N : Index := Arr'Length;
    T : Index;

    procedure Siftdown(N,K : Index) is
        J : Index;
        V : Integer;
    begin
        V := Arr(K);
        discrete H := K in 1 .. N/2 new H := 2*H | 2*H+1 loop
            J := 2*H;
            if J < N and then Arr(J) < Arr(J+1) then
                J := J+1;
            end if;
            if V >= Arr(J) then
                Arr(H) := V;
                exit;
            end if;
            Arr(H) := Arr(J);
            Arr(J) := V;
            H := J;
        end loop;
    end Siftdown;

begin --  Heapsort
    for K in reverse 1 .. N/2 loop
        Siftdown(N,K);
    end loop;
    for M in reverse 2 .. N loop
        T := Arr(1);
        Arr(1) := Arr(M);
        Arr(M) := T;
        Siftdown(M-1,1);
    end loop;
end Heapsort;
```

Figure 3: Heapsort

1.1.2 Discrete Loops with Remainder Functions

Although monotonical discrete loops are applicable to many problems where a general loop would have to be used otherwise, it is sometimes not desirable or even not possible to have the loop variable follow a monotonical iteration sequence. Many times this does not mean that the problem under consideration does not impose some upper bound on the number of iterations of the loop.

To be able to treat such cases, the concept of the *remainder loop variable* has been introduced. The remainder loop variable draws its name from the fact that it usually describes the amount of work that remains to be done at some stage of the loop[1]. The value of the remainder loop variable is computed during each iteration by the so-called *remainder function*, which must be a monotonically decreasing function. By that means we are able to guarantee upper bounds as well as termination in a similar way as for monotonical discrete loops.

Since the remainder loop variable must be of a discrete type, this restriction is not imposed on the loop variable anymore. Therefore the programmer has the freedom to iterate over whatever he chooses except *limited* [Ada95](7.5) or *abstract* [Ada95](3.9.3) types, which is considered a major advantage over the traditional for loop.

```
discrete Node_Pointer := Root
   new Node_Pointer := Node_Pointer.Left | Node_Pointer.Right
 with H := Height
   new H = H - 1 loop

   – loop body:
   – Here the node pointed at by node_pointer is processed
   – and node_pointer is either set to the left or right
   – successor.
   – The loop is completed if node_pointer = null.

end loop;
```

Figure 4: Binary Tree Traversal

Figure 4 shows an example of a discrete loop with a remainder function. Its purpose is the traversal of a binary tree. The loop variable points to the current node, whereas the remainder loop variable describes the height of the remaining subtree.

[1]e.g. the number of remaining data items

Syntactical and Semantical Issues of Discrete Loops with Remainder Functions

The syntax of discrete loops with remainder functions differs from the syntax given in Section 1.1.1 only in its loop_parameter_specification. Therefore only this part is given here. A comprehensive syntax for all kinds of discrete loops can be found in appendix A.

```
discrete_loop_with_remainder_function_parameter_specification ::=
  [identifier := initial_value
     new identifier := list_of_iteration_functions]
     with rem_identifier := initial_value new remainder_function

remainder_function ::=
    rem_identifier = expression |
    rem_identifier <= upper_bound_expression
      [and rem_identifier >= lower_bound_expression]
```

For a discrete loop with a remainder function, the corresponding loop parameter specification is the optional declaration of the loop variable with the given identifier. The type of the loop variable is derived from the initial_value and the list_of_iteration_functions. It can be anything but a *limited* or an *abstract* type. The initial value of the loop variable is given by initial_value. Within the sequence of statements, the loop variable behaves like any other variable, i.e., it can be used on both sides of an assignment statement.

After the keyword **with** the remainder loop variable is declared by the given identifier (rem_identifier). It must be of subtype **natural** or of a subtype with a static lower bound of zero that has **natural** among its ancestors. The reaminder_function itself may have three different forms:

Case 1: If the remainder function can be determined exactly, it is given by an equation.

Case 2: If only an upper bound of the remainder function is available, it is given by an inequality (<=).

Case 3: If in addition to (2) a lower bound of the remainder function is known, it can be given by an optional inequality (>=). The second inequality must be separated from the first by the keyword **and**.

In case (1) the remainder loop variable behaves like a constant within the sequence of statements. In cases (2) and (3) the remainder loop variable behaves like any other variable within the sequence of statements. If the value of the remainder loop variable is changed during execution of the statements, we call the original value *previous value* and the new value *current value.*

Before the sequence of statements is executed, the list of iteration functions is evaluated if a loop variable is given. This results in a list of *possible successive values.* The remainder function or its bounds (depending on which are given by the programmer) are also evaluated.

After the sequence of statements has been executed, it is checked whether the loop variable is contained in the list of possible successive values. If this check fails, the exception **successor_error** is raised.

Thereafter the remainder loop variable is set to its new value. In case (1) this is the value calculated by the given expression, provided that it is smaller than the current value of the remainder loop variable. If not, exception **monotonic_error** is raised. Case (2) consists of two possibilities:

1. Previous value = current value (the code in the loop body did not touch the remainder loop variable): The remainder loop variable is set to the value calculated by the upper_bound_expression if this value is smaller than the current value. Otherwise exception **monotonic_error** is raised.

2. Previous value not equal current value: In this case the following condition must hold: *current value* <= *value of upper_bound_expression* < *previous value.* Otherwise exception **monotonic_error** is raised.

Case (3) is the same as case (2), but it is also checked that this iteration's lower_bound_expression is not greater than last iteration's lower_bound_expression and that the interval [*lower_bound_expression*, *current value*] contains at least one element. Any violation of the conditions above triggers a **monotonic_error** exception.

If in cases (1), (2), and (3) the value of the remainder loop variable is zero, the exception **loop_error** is raised, otherwise the loop is continued. The regular way to complete a discrete loop with a remainder function is to use an *exit* statement before the remainder loop variable reaches zero.

Remark 1.1.1 The semantics of discrete loops with remainder functions ensure that such a loop will always complete, either if the loop is terminated by an *exit* statement or because one of the above checks fails, i.e., one of the exceptions **monotonic_error**, **successor_error** or **loop_error** is raised.

1.2 Multi-Dimensional Discrete Loops

The concept of discrete loops can be extended easily in order to support more than one loop variable. Although this might contrast with the appearance of the traditional for-loop, it is justified by the following facts:

- Sometimes it is hard to pick a single entity of the problem domain that can serve as the loop variable. Instead, several entities together play the part of the loop variable.
- Many times the loop variable is not the only entity that changes its value during each iteration in a predefined and deterministic way. For the analysis of programs it can be of great value to have this regularity declared explicitly in the loop header.
- Static and dynamic checks can make use of the information specified in such a loop header, which in turn facilitates testing and verification of the underlying program.

Since one can think of several loop variables as one big aggregate containing those variables, we speak of them as discrete loops with *aggregate variables*, but they can also be refered to as *multi-dimensional* discrete loops.

1.2.1 Multi-Dimensional Monotonical Discrete Loops

The scenario given in Figure 5 may serve as an introductory example for the use of a multi-dimensional monotonical discrete loop. Suppose *Pac Man* wants to leave the building. Depending on the contents of each of the 16 fields he has several possibilities to make his way to the exit (a path is characterized as a series of tupels of x- and y- coordinates):

$$((1,1),(2,1),(3,1),(4,1),(4,2),(4,3),(4,4))$$

or

$$((1,1),(2,2),(3,3),(4,4))$$

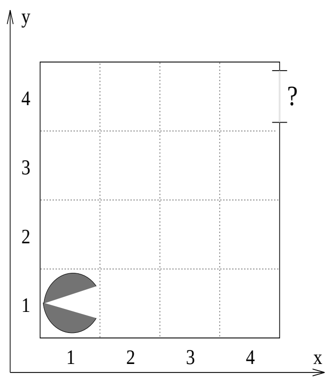

Figure 5: Introductory Example for Multi-Dimensional Discrete Loops

or

$$((1,1),(1,2),(1,3),(2,4),(3,4),(4,4)) \ldots$$

An auto-pilot could be coded as follows:

```
1   discrete (X,Y) := (1,1) in (1 .. 4 , 1 .. 4)
2    new (X,Y) := (X+1,Y)|(X,Y+1)|(X+1,Y+1)
3   loop
4   --  Eat, compute next field, move.
5   end loop;
```

In this example X and Y together play the role of an aggregate loop variable representing Pac Man's position. The iteration_functions in line 2 specify that within the predefined range of 1..4 Pac Man is allowed to advance to the right (X+1,Y), up (X,Y+1) or diagonal (X+1,Y+1). However, he is not allowed to move backwards or to stand still, since this would violate the loop's monotony. As it is the case with single-dimensional loops, the value of the loop variable is calculated in the loop body. The loop header only provides a definite forecast.

Another prominent example is Euclid's algorithm [SE88] (confer Figure 6). It computes the greatest common divisor (gcd) of two positive integers. Keyword *reverse* defines both dimensions of the loop to be monotonically decreasing. The loop variable (M,N) takes the following values

$$((M,N),\ (N, M\ mod\ N),\ (M\ mod\ N, N\ mod\ (M\ mod\ N))) \ldots$$

during subsequent iterations, until a value of (X,0) terminates the loop.

```
function Euclid (A, B : Positive) return Positive is
   H : Positive;
begin
   if B > A then
      return Euclid (B, A);       -- swap
   end if;
   discrete (M,N) := (A,B) in reverse (1..A, 1..B)
    new (M,N) := (N, M mod N) loop
      H := M;
      M := N;
      N := H mod N;
   end loop;
   return H;
end Euclid;
```

Figure 6: Euclid's Algorithm

Syntax of Multi-Dimensional Monotonical Discrete Loops

The syntax of multi-dimensional monotonical discrete loops differs from the syntax given in Section 1.1.1 only in its loop_parameter_specification. Therefore only this part is given here. A comprehensive syntax for all kinds of discrete loops can be found in Appendix A.

```
multi_dimensional_monotonical_discrete_loop_parameter_specification ::=
    identifier_aggregate := initial_value_aggregate in [reverse] range_aggregate
       new identifier_aggregate := iteration_function_specification

identifier_aggregate ::= ( identifier { , identifier } )

initial_value_aggregate ::= ( initial_value { , initial_value } )

range_aggregate ::= ( [reverse] discrete_subtype_definition
    {, [reverse] discrete_subtype_definition} )

iteration_function_specification ::= iteration_function_aggregate
    {| iteration_function_aggregate}

iteration_function_aggregate ::= (list_of_iteration_functions
    {,list_of_iteration_functions})
```

The identifier_aggregate determines the number of dimensions of the discrete loop. Initial_value_aggregate, range_aggregate and iteration_function_aggregates have to correspond to the number of dimensions of the identifier_aggregate. Note that an Iteration_Function_Aggregate of the form $(A|B,C)$ is meant as a shorthand notation for two aggregates (A,C) and (B,C).

Semantics of Multi-Dimensional Monotonical Discrete Loops

The elaboration of the loop parameter specification declares a parameter for every dimension of the loop. Types are derived from the initial_value of the corresponding initial_value_aggregate and the discrete_subtype_definition of the corresponding range_aggregate. All parameters together make up the *loop variable.* Within the loop body the loop variable is referenced through its parameters. They behave like any other variable, i.e., they can be used on both sides of an assignment statement.

Elaboration of the loop parameter specification also elaborates the initial_values of the initial_value_aggregate as well as the discrete_subtype_definitions of the range_aggregate. If one of the discrete_subtype_definitions defines a subtype with a null range, execution of the loop is complete.

The optional keyword **reverse** can appear before the range_aggregate. This defines all dimensions of the loop to be monotonically decreasing. If keyword **reverse** appears right before some discrete_subtype_definition within the range_-aggregate, this particular dimension of the loop is defined to be monotonically decreasing. Keyword **reverse** must not appear before *and* within the range_aggregate.

Before the sequence of statements is executed, the iteration functions in the iteration_function_specification are evaluated for every dimension of the loop. All dimensions together recruit the *possible successive values* of the loop variable. Like with the one-dimensional case, it has to be checked whether these values obey monotony. What monotony actually means in this context is subject of the next paragraph.

Monotony in the Case of Multi-Dimensional Discrete Loops

Definition 1.2.1 A multi-dimensional discrete loop L with N ($N \in \mathbf{N}$) identifiers in its identifier_aggregate is said to be of *dimension* N. This is denoted by $Dim(L) = N$.

Definition 1.2.2 As a shorthand notation for a loop variable $(a_1, a_2, .., a_i)$ we write (a_i). The i^{th} parameter of (a_i) is written as a_i.

Definition 1.2.3 $(a_i) = (b_i) \Leftrightarrow \forall i_{1 \le i \le Dim(L_{(a_i),(b_i)})}\ a_i = b_i$

Definition 1.2.4 $(a_i) < (b_i) \Leftrightarrow \forall i_{1 \le i \le Dim(L_{(a_i),(b_i)})}\ a_i \le b_i\ \wedge\ \exists j : a_j < b_j$

If discrete loops were required to be uniformly *reverse* or not *reverse* in all dimensions, Definition 1.2.4 would suffice in order to to define monotony.

Problem 1.2.1 It is possible for an N - dimensional loop ($N > 1$) to deploy discrete_subtype_definitions with and without keyword reverse at once e.g.

```
discrete (A,B) := (1,100) in (1..100, reverse 1..100)
 new (A,B) := (A*2, B-A)
loop
-- Body suppressed.
end loop;
```

In this example the loop variable (A,B) takes the values

$$((1,100),\ (2,99),\ (4,97),\ (8,93),\ (16,85),\ (32,69),\ (64,37))$$

until the loop completes. While the iteration sequence of the first dimension is monotonically increasing, the second dimension bears a monotonically decreasing iteration sequence. Therefore Definition 1.2.4 has to be modified slightly to be able to cope with such mixed cases.

Definition 1.2.5 We define relation 'M' as an extension of the common 'smaller'-relation '<' between two integers: For a given loop L and two values (a_i) and (b_i) of the loop variable

$$a_j\ \mathrm{M}\ b_j \Leftrightarrow \begin{cases} a_j < b_j & \textit{if } j^{th} \textit{ dimension of } L \textit{ not defined to be } \textbf{reverse} \\ a_j > b_j & \textit{else} \end{cases}$$

$$a_j\ \underline{\mathrm{M}}\ b_j \Leftrightarrow \begin{cases} a_j \le b_j & \textit{if } j^{th} \textit{ dimension of } L \textit{ not defined to be } \textbf{reverse} \\ a_j \ge b_j & \textit{else} \end{cases}$$

Definition 1.2.6 With definition 1.2.5 we are able to define monotony between two values of an mixed-case aggregate loop variable of loop L:

$$(a_i)\ \mathrm{M}\ (b_i) \Leftrightarrow \forall i_{1 \le i \le Dim(L_{(a_i),(b_i)})}\ a_i\, \underline{\mathrm{M}}\, b_i\ \wedge\ \exists j : a_j\, \mathrm{M}\, b_j$$

Before the sequence of statements is executed, the current value of the loop variable (a_i) is compared to the set of possible successive values $\{(psv_i)\}$. Every value (psv_i) has to fulfill the relation $(psv_i) \,\textsc{m}\, (a_i)$. Otherwise exeption monotonic_error is raised.

After the sequence of statements has been executed, it is checked whether the value of the loop variable is contained in the set of possible successive values $\{(psv_i)\}$. If this check fails, the exception **successor_error** is raised.

If the value of the loop variable is still within the bounds stated in the loop header, the loop is iterated (at least) once more. If not, the loop completes.

Remark 1.2.1 The semantics of multi-dimensional monotonical discrete loops ensure that such a loop will always complete, either because the value of the loop variable is outside its bounds or because one of the above checks fail, i.e., one of the exceptions **monotonic_error** or **successor_error** is raised.

1.2.2 Multi-Dimensional Discrete Loops with a Remainder Function

Monotonical discrete loops with remainder functions can also be extended to support aggregate loop variables. **Binary Search** is an algorithm that lends itself nicely to an implementation powered by a discrete loop (confer Figure 7).

The essential property of binary search is a sequence of intervals which become smaller and smaller with each iteration of the loop. The starting interval $[l_1, u_1] = [1, N]$ is changed with each iteration of the loop according to

$$[l_{i+1}, u_{i+1}] = \begin{cases} [l_i, \lfloor \frac{u_i+l_i}{2} - 1 \rfloor] & \textit{or} \\ [\lfloor \frac{u_i+l_i}{2} + 1 \rfloor, u_i] \end{cases}$$

depending on the sub-interval that contains the element being sought. The interval $[l_i, u_i]$ is represented by the loop variable $[L, U]$, the remainder loop variable I represents the number of remaining data items which equals the interval length.

Syntax of Multi-Dimensional Discrete Loops with Remainder Functions

The syntax of multi-dimensional discrete loops with remainder functions differs from the syntax given in Section 1.1.1 only in its loop_parameter_specification. Therefore only this part is given here. A comprehensive syntax for all kinds of discrete loops can be found in Appendix A.

multi_dimensional_discrete_loop_with_remainder_function_parameter_spec ::=

```
    [identifier_aggregate := initial_value_aggregate
        new identifier_aggregate := iteration_function_specification]
        with rem_identifier := initial_value new remainder_function

remainder_function ::=
    rem_identifier = expression |
    rem_identifier <= upper_bound_expression
      [and rem_identifier >= lower_bound_expression]

identifier_aggregate ::= ( identifier { , identifier } )

initial_value_aggregate ::= ( initial_value { , initial_value } )

iteration_function_specification ::= iteration_function_aggregate
    {| iteration_function_aggregate}

iteration_function_aggregate ::= (list_of_iteration_functions
    {,list_of_iteration_functions})
```

Semantics of Multi-Dimensional Loops with Remainder Functions

Essentially the same as in Section 1.1.2, with the difference that the loop variable is an aggregate variable as opposed to one of an elementary type.

```
N : constant Positive := ??; -- Number of elements.
subtype Index is Positive range 1 .. N;
type Sort_Array is array (Index) of Integer;

function Binary_Search
 (Item : in Integer;
  Arr : in Sort_Array)
return Index is
    M : Index;
begin --  Successful search or runtime error.
    discrete (L,U) := (Arr'First, Arr'Last)
      new (L,U) := (L , (L+U)/2 - 1) | ((L+U)/2 + 1 , U)
      with I := Natural (U-L+1) new I <= I/2
    loop
        M := (L+U)/2;
        if Item < Arr (M) then
            U := M - 1;
        elsif Item > Arr (M) then
            L := M + 1;
        else
            exit;
        end if;
        I := U-L+1;
    end loop;
    return M;
 end Binary_Search;
```

Figure 7: Binary Search

Chapter 2
IMPLEMENTATION CONSIDERATIONS

It has ever been seen as an integral part of Project WOOP to implement a tool that incorporates the theoretical results gathered. Concerning discrete loops it was therefore necessary to provide a translation-mechanism that would enable programmers to use discrete loops in their code. Although in principle there was no cause to favor any particular programming language, it was the reliability and maintainability of *Ada95* that finally lead to the decision to build a preprocessor capable of translating discrete loop augmented Ada95 code into standard Ada95 in a way that would preserve the *semantics* of discrete loops. The resulting code could then be compiled by any Ada compiler. The preprocessor would have to perform the following tasks:

- Scan its input.
- Perform syntax checks
- Perform semantic checks.
- Analyze discrete loops in order to find an upper bound for the number of iterations.
- Transform discrete loops into their equivalent in standard Ada, preserving the semantics of those loops.
- Generate the resulting Ada code.

Note 2.0.1 Syntactic and semantic analysis has to be done for the entire program, considering only those statements belonging to a discrete loop does by no means suffice! A complete description of Ada95's syntax and semantics can be found in [Ada95] p.1 - 551.

Two possibilities seemed to exist to get the job done:

- Build from scratch.
- Use a scanner/parser generator to simplify things a little bit.

In this situation it was the availability of GNAT that saved a lot of work and enabled me to focus on the main problems. It prevented me from coding a complete front-end for the whole Ada programming language (comp. note 2.0.1) while providing every facility I needed to build upon. GNAT itself is written entirely in Ada, which makes this huge piece of software (over 12 Megabyte of source code) very modular. In fact this is also an achievement of the GNAT Team, since good tools alone do not necessarily lead to good programs. The coupling between its modules is really loose. In this way one can apply modifications locally and need not know all the details of the program as a whole. The source code of GNAT is very well structured and fully documented. It left me with enough space to add where I needed and in the way I wanted.

2.1 Extending the GNAT System

GNAT itself is a front-end and runtime system that uses the back-end of GCC as a retargetable code generator. The front-end uses an Abstract Syntax Tree (AST)[1] as the underlying data structure and it comprises three phases:

- Syntactic Analysis
- Semantic Analysis
- Transformation of the AST to a representation suitable for the back-end

Since only a minor part of the third phase is coded in a language other than Ada, the strengths of the Ada programming language also come into play in the source code of GNAT which properly reflects the structure of the corresponding chapters of the Reference Manual [Ada95]. This leads to a very modular functional design with no coupling between unrelated units.

Although the current release of GNAT has already been validated, development of the compiler is not yet finished. This means that modifications of the current release of GNAT might have to be taken over to a future release. Therefore I have attempted to keep modifications of the original code as small as possible while providing 'extra'- functionality in separate units that are called where appropriate (e.g. on encountering keyword *discrete* during syntactic or semantic analysis). Because of arising dependencies it was on the other hand necessary to properly integrate extensions into the original code.

[1]Details can be found in Section 3.

Although these requirements seem to contradict each other, the hierarchical library mechanism of Ada95 actually made it possible to fulfill both of them in most cases:

- Code that is totally independent of GNAT is kept in separate *library units.*
- Code that extends the functionality of an existing library unit is usually kept in a *child* of that unit. In this way we not only inherit the context provided by the context clauses of the parent, but we are also able to directly access all the entities that are declared in the spec of the parent from within the child. The parent body itself may access the child by means of a *with clause.* Children however are not allowed for subprograms since [Ada95] requires the parent unit to be a (generic) package.
- Code that depends on entities declared in the body of a library unit or code that is meant to extend a library unit that is a subprogram can only be separated by means of *subunits.* Visibility in the subunit is the same as at the corresponding stub in the parent body, except for differences due to context clauses of the subunit itself. Coupling of subunits to the original code is usually tighter than in the previous cases.

The following data illustrates this approach:

- The whole GNAT system consists of about 300000 lines of code.
- Modifying it to build a preprocessor for discrete loops took 4772 lines of code.
- Only 107 out of 4772 lines had to be applied to the original code. Those modifications are always preceded by a comment line like '`-- BB:`', which should make it easy to locate them within the original code. The two-letter combination `´BB´` has something to do with the author's name :-)

2.2 A Math-Package to Count On

Estimating the number of iterations of a discrete loop often involves complicated recurrence relations which have to be solved accordingly. Checking iteration functions for monotonic sequences of loop values also requires solving linear and non-linear equations. Although it should be possible in theory to implement a computer algebra system from scratch, in practice it can be considered a rather time-consuming and costly task. The kernel of Wolfram Research's Mathematica alone, for instance, consists of 300K lines of C code. An additional package that extends Mathematica to solve recurrence relations takes another 2,5K lines of code, the whole Mathematica system consists of about 4-6 megabytes of compiled code.

With those figures in mind the decision was made to take some computer algebra system and incorporate it into the preprocessor. From all packages available at that time, only Mathematica offered a fully documented programming interface to the kernel. Only in this way it was possible to start the math-subsystem as a slave process and have it evaluate the expressions passed to it from the preprocessor. It is also a fact that Mathematica's programming interface is the *only* interface to its kernel and that therefore also Wolfram Research's own front-ends use it. This information directly adds to the performance and dependability of that interface. Some other highlights of Mathematica were

- Comprehensive calculation and analysis tools.
- Open and extendible architecture.
- High-level programming language.
- Documentation and programs available on the net.
- Available on a wide variety of platforms.

If for some reason another (i.e. non-commercial) computer algebra system should turn out to be superior to Mathematica, an additional level of abstraction has been implemented within the preprocessor to allow easy replacement of the math subsystem.

2.3 What Name?

Another contribution to the devastating cluttering of our world with acronyms... **WPP**, the **W**oop **P**re**p**rocessor, also known as **double(UP)** or simply **DoubleUP**.

$$WPP = double(U)PP = double(U)\,double(P) = double(UP)$$

Possible further transformations, suggested by my supervisor:

$$double(UP) = U^2P^2 = (UP)^2$$

Chapter 3
SHAKING THE TREE

The Abstract Syntax Tree (or AST for short) is perhaps GNAT's single most important data structure. It is constructed by the recursive descent parser and represents the input program in a tree-like form. Subsequent processing in the front end traverses the tree, transforming it in various ways and adding semantic information. Therefore no separate symbol table structure is needed.

3.1 The Internal Representation of the Abstract Syntax Tree

Package Atree defines the basic structure of the tree and its nodes and provides the basic abstract interface for manipulating the tree. Two other packages use this interface to define the representation of Ada programs using this tree format. The package Sinfo defines the basic representation of the syntactic structure of the program, as output by the parser. The package Einfo defines the semantic information which is added to the tree nodes that represent declared entities (i.e. the information which might typically be described in a separate symbol table structure).

3.1.1 Definition of a Single Tree Node

The representation of the tree is completely hidden, using a functional interface for accessing and modifying the contents of nodes. Logically a node contains a number of fields, much as though the nodes were defined as a record type. Those fields that are used for WPP-specific modifications are summarized here.

- *Nkind*: Indicates the kind of the node. This field is present in all nodes.
- *Sloc*: Location (Source_Ptr) of the corresponding token in the Source buffer.
- *Comes_From_Source*: This flag is present in all nodes. It is set if the node is built by the scanner or parser, and clear if the node is built by the analyzer or expander. It indicates that the node corresponds to a construct that appears in the original source program.

- *Field1 - Field5*: Five fields holding a union of the following values[1]:
  - *ListN*: Synonym for FieldN typed as List_Id, a list of nodes.
  - *NameN*: Synonym for FieldN typed as Name_Id, an entry into the *names* table. This table is used to store character strings for identifiers and operator symbols, as well as other string values such as unit names and file names.
  - *NodeN*: Synonym for FieldN typed as Node_Id, a link to another node.
  - *UintN*: Synonym for FieldN typed as Uint (GNAT's basic universal integer type)
- *Flag4 - Flag18*: Fifteen Boolean flags.
- *Link*: Pointer to parent node, i.e. node which points to the node which references this field.

The actual usage of FieldN (i.e. whether it contains a List_Id, Name_Id, Node_Id, Uint), depends on the value in Nkind. Generally the access to this field is always via the functional interface, so the field names ListN, NameN, NodeN, and UintN are used only in the bodies of the access functions (i.e. in the bodies of Sinfo and Einfo). These access functions contain debugging code that checks that the use is consistent with Nkind values.

The use of the Boolean flags also depends on Nkind and Ekind, as described for FieldN. Again the access is usually via subprograms in Sinfo and Einfo which provide high-level synonyms for these flags, and contain debugging code that checks that the values in Nkind and Ekind are appropriate for the access.

3.1.2 Extended Nodes

As stated above, the AST contains not only the full syntactic representation of the program, but also the results of semantic analysis. In particular, the nodes for defining identifiers, defining character literals and defining operator symbols, collectively referred to as entities, represent what would normally be regarded as the symbol table information. In order to store this additional information, a node containing an entity can be extended in order to provide extra space for additional fields and flags. The use of those fields and flags is defined in package Einfo. It

[1]Again, only those values important for WPP are listed

depends on the entity kind, as defined by the contents of the Ekind field. Some other important fields:

- *Etype*: Present in all entities. Represents the type of the entity, which is itself another entity. For a type entity, this pointer is selfreferential. For a subtype entity, Etype is the base type.

- *First_Subtype*: Applies to all types and subtypes. For types, yields the first subtype of the type. For subtypes, yields the first subtype of the base type of the subtype [Ada95](3.2.1(6)).

- *Homonym*: Present in all entities. Contains a link to chain entities that are homonyms and that are declared in the same or enclosing scopes. (Homonyms in the same scope are overloaded).

- *Is_Abstract*: Present in all types, and also for functions and procedures. Set for abstract types and abstract subprograms.

- *Is_Discrete_Type*: Defined for all entities, true for all discrete types and subtypes.

- *Is_Enumeration_Type*: Defined for all entities, true for enumeration types and subtypes.

- *Scope*: Present in all entities. Points to the entity for the scope (block, loop, subprogram, package etc.) in which the entity is declared.

- *Type_High_Bound* Defined for scalar types. Returns the tree node that contains the high bound of a scalar type.

- *Type_Low_Bound*: See previous field.

3.1.3 Abstract Interface to the Abstract Syntax Tree

Package Atree.Unchecked_Access provides generic field access routines for tree nodes. These routines are used by package Sinfo and Einfo to provide a high-level interface based on logical synonyms for fields and flags.

3.2 Basic Tree Structure

Package Sinfo defines how the tree structure provided by package Atree is used to represent the syntax of an Ada program. The layout of every syntactic construct is described in a comment at the beginning of the spec of package Sinfo. The specification and the body of that package follow the definitions given in the comment. As an example the comments associated with the loop statement are given below.

sinfo.ads

```
----------------------------------
-- 5.5  Loop Statement --
----------------------------------

--  LOOP_STATEMENT ::=
--    [loop_STATEMENT_IDENTIFIER :]
--      [ITERATION_SCHEME] loop
--        SEQUENCE_OF_STATEMENTS
--      end loop [loop_IDENTIFIER];

--  N_Loop_Statement
--  Sloc points to LOOP
--  Identifier (Node1) loop identifier (set to Empty if no identifier)
--  Iteration_Scheme (Node2) (set to Empty if no iteration scheme)
--  Statements (List3)

----------------------------------
-- 5.5 Iteration Scheme --
----------------------------------

--  ITERATION_SCHEME ::=
--    while CONDITION | for LOOP_PARAMETER_SPECIFICATION

--  N_Iteration_Scheme
--  Sloc points to WHILE or FOR
--  Condition (Node1) (set to Empty if FOR case)
--  Loop_Parameter_Specification (Node4) (set to Empty if WHILE case)

-------------------------------------------------
-- 5.5  Loop parameter specification --
-------------------------------------------------

--  LOOP_PARAMETER_SPECIFICATION ::=
--    DEFINING_IDENTIFIER in [reverse] DISCRETE_SUBTYPE_DEFINITION

--  N_Loop_Parameter_Specification
--  Sloc points to first identifier
--  Defining_Identifier (Node1)
--  Reverse_Present (Flag15)
--  Discrete_Subtype_Definition (Node4)
```

According to this definition, the layout of a tree fragment for a while- or for-loop looks as follows (note that a loop can only contain a condition (while) or a Loop_Parameter_Specification (for), but not both):

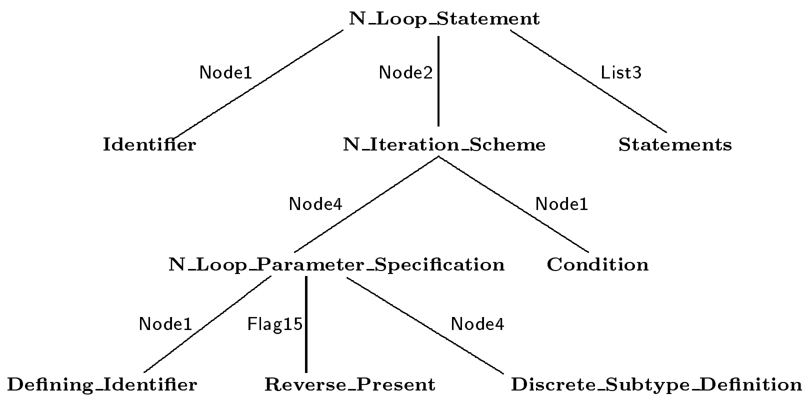

Figure 8: AST fragment for while- and for-loops.

For each node (e.g. N_Loop_Statement, N_Iteration_Scheme, N_Loop_Parameter_-Specification) package sinfo provides access functions that are named after the logical synonyms as stated in the comments. The bodies of those functions make use of the abstract interface mentioned in Section 3.1.3.

```
1    function Iteration_Scheme
2       (N : Node_Id) return Node_Id is
3    begin
4       pragma Assert (False
5           or else NT (N).Nkind = N_Loop_Statement);
6       return Node2 (N);
7    end Iteration_Scheme;

8    procedure Set_Iteration_Scheme
9       (N : Node_Id; Val : Node_Id) is
10   begin
11      pragma Assert (False
12          or else NT (N).Nkind = N_Loop_Statement);
13      Set_Node2_With_Parent (N, Val);
14   end Set_Iteration_Scheme;
```

Figure 9: Logical Synonyms based on the Abstract Interface

Lines 4-5 and 11-12 of Figure 9 are part of the debugging code that checks that the use of the logical synonyms is consistent with Nkind values (you cannot convince GNAT to provide you with a loop-body out of a case-statement, for that reason). Since GNAT is very picky about this, we created our own high-level interface for WOOP-specific parts of the AST. It is defined in package WoopSinfo and it is discussed in the next section.

3.3 WPP-Specific Leaves

It is necessary for the abstract syntax tree to grow a little extra-branch in order to accommodate the syntactic representation of a discrete loop. Node N_Loop_-Parameter_Specification can therein serve as a starting point, because down to the Loop_Parameter_Specification the syntax for for-loops and discrete loops is the same (confer Appendix A). Tree fragments for discrete loops can be divided in two categories:

- Syntactic constructs related to the *loop variable*.
- Syntactic constructs belonging to the *remainder* loop variable.

The purpose of the following two subsections is to grow subtrees for the syntactic categories given above. For the sake of clarity and understanding they are given in the same graphical notation that has already been used in Figure 8. The third subsection shows how those subtrees are connected (*grafted*) to the abstract syntax tree.

3.3.1 Constructs Related to the Loop Variable

According to Appendix A this includes the following entities:

- Identifier
- Initial_Value
- Discrete_Subtype_Definition (in case of a monotonical discrete loop)
- List_Of_Iteration_Functions

A suitable data structure capable of storing those entities might look as follows:

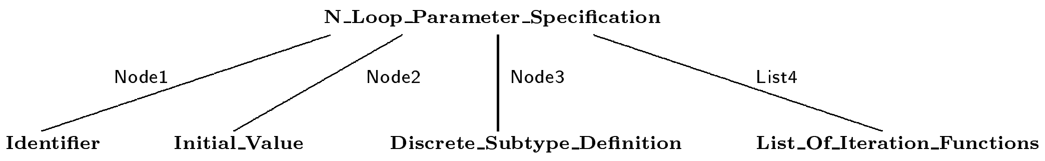
N_Loop_Parameter_Specification
Node1
Node2
Node3
List4
Identifier
Initial_Value
Discrete_Subtype_Definition
List_Of_Iteration_Functions

Figure 10: Simple Data Structure

Multi-dimensional discrete loops need a list that can store such a subtree for each of their N dimensions. Although this works well for Identifier, Initial_Value and Discrete_Subtype_Definitions, it won't work for iteration functions.

Problem 3.3.1 If iteration functions are stored on a per-dimension basis, the information that maps a certain iteration function to a certain Iteration_Function-_Aggregate gets lost. Consider the algorithm for Binary Search given in Figure 7 on page 16. The possible successive values for the loop variable *(L, U)* are $(L, (L+U)/2-1)$ and $((L+U)/2+1, U)$ respectively. Storing the iteration functions on a per-dimension basis, we get $\{L, (L+U)/2+1\}$ for the first dimension and $\{(L+U)/2-1, U\}$ for the second one. We have now successfully discarded the information on how the iteration functions of the two dimensions were connected. Building all possible combinations e.g. $(L, (L+U)/2-1)$, (L, U), $((L+U)/2+1, (L+U)/2-1)$, $((L+U)/2+1, U)$ does not help either, because not all combinations have been specified in the loop header.

In this way the data structure given in Figure 10 cannot be used to store iteration functions of multi-dimensional loops. Nevertheless it is suitable for Identifier, Initial_Value, and Discrete_Subtype_Definition:

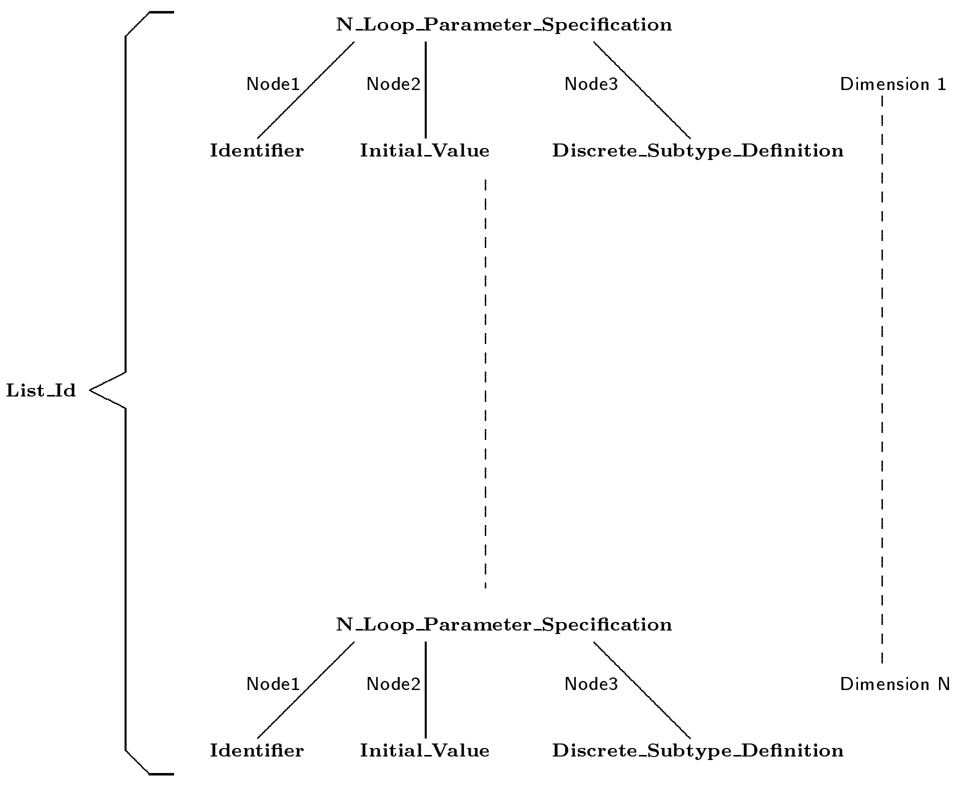
N_Loop_Parameter_Specification
Node1
Node2
Node3
Dimension 1
Identifier
Initial_Value
Discrete_Subtype_Definition
List_Id
N_Loop_Parameter_Specification
Node1
Node2
Node3
Dimension N
Identifier
Initial_Value
Discrete_Subtype_Definition

Figure 11: Data Structure without Iteration Functions

Iteration Functions

Before the tree segment for iteration functions can finally be defined, a short paragraph on the nature of GNAT's *lists* is needed.

Package Nlists provides facilities for manipulating lists of nodes. A node list is a list that is threaded through nodes (using the Link field). This means that it takes minimum space, but a node can be on at most one such node list. For each node list, a list header is allocated in the lists table, and a **List_Id** value references this header which may be used to access the nodes in the list using the set of routines that define the interface.

Problem 3.3.2 A node list can contain nodes or extended nodes, but it is not possible to build a list of lists.

To overcome this little shortage, one can build a list of nodes, where each node serves as a pointer to another list. This list of nodes-to-lists is equivalent to a list of lists.

Now consider an N-dimensional loop with M Iteration_Function_Aggregates. Due to the shorthand notation defined in Section 1.2.1 we also have to take into account that dimension n $(1 \leq n \leq N)$ of Iteration_Function_Aggregate m $(1 \leq m \leq M)$ contains k $(k \geq 1)$ iteration functions. This is denoted by $K_{m,n}$. The iteration functions are indexed as $IF_{m,n,k}$.

- m: m^{th} Iteration_Function_Aggregate, $1 \leq m \leq M$
- n: n^{th} dimension, $1 \leq n \leq N$
- k: k^{th} iteration function of dimension n of Iteration_Function_Aggregate m, $1 \leq k \leq K_{m,n}$

Figure 12 makes use of this notation in order to define a subtree that can accommodate the common case

$$\begin{gathered}(IF_{1,1,1}|\ldots|IF_{1,1,K_{1,1}},\ldots,IF_{1,N,1}|\ldots|IF_{1,N,K_{1,N}})\ | \\ \ldots \\ |(IF_{M,1,1}|\ldots|IF_{M,1,K_{M,1}},\ldots,IF_{M,N,1}|\ldots|IF_{M,N,K_{M,N}})\end{gathered}$$

of an Iteration_Function_Specification.

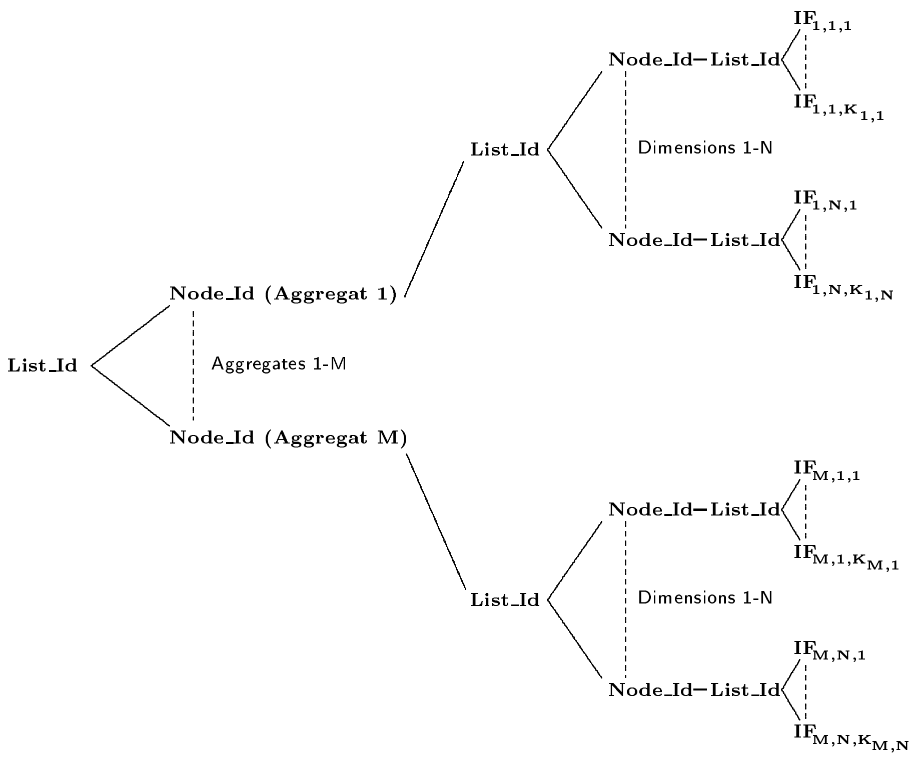

Figure 12: Data Structure for an Iteration_Function_Specification

Since every discrete loop consists of at least one dimension, the iteration function's subtree is connected to the data structure given in Figure 11 via *Node4* of the first N_Loop_Parameter_Specification node.

3.3.2 Constructs Related to the Remainder Loop Variable

Due to the fact that only one remainder_function is allowed for each discrete loop, this subtree is rather simple (Figure 13). The field '**Token**' (Uint3) in fact contains

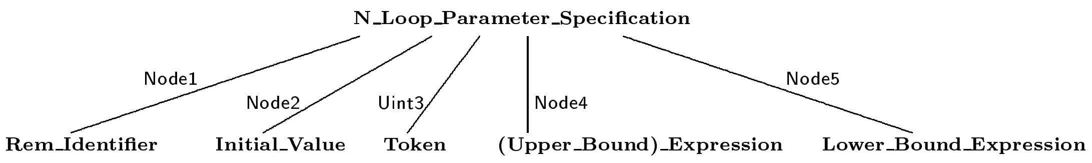

Figure 13: Data Structure for a Remainder Function Specification

one of the values Tok_Equal or Tok_Less_Equal of type Token_Type (package scans) depending on the token ('=' or '<=') given in the remainder_function.

3.3.3 Connecting Subtrees to the Abstract Syntax Tree

The constructs related to the loop variable (Figure 11) as well as those related to the remainder loop variable (Figure 13) have to be connected to the AST through node N_Loop_Parameter_Specification. Since the exact layout of the abstract syntax tree as it is given in package sinfo is subject to change[2], it is wise to implement an additional layer of abstraction between GNAT's AST and WPP's amendments. Figure 14 illustrates the approach chosen. The nodes within the 'cloud' represent

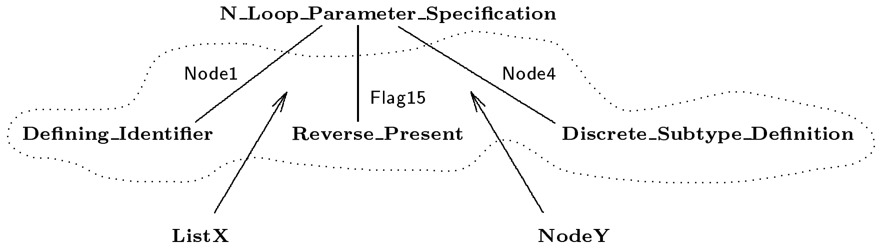

Figure 14: Connection to the AST.

[2]Only GNAT-specific implementational details may change, the big picture is more or less defined in [Ada95]

what usually gets stored in the tree for a for-loop. For a discrete loop this cloud is replaced by two fields:

- ListX: This is the field containing the constructs related to the loop variable (the subtree illustrated in Figure 11).
- NodeY: This field contains the subtree related to the remainder loop variable as illustrated in Figure 13.

Instead of assigning specific nodes (Node1, Node2,...) to ListX and NodeY, which might conflict with future releases of GNAT, their actual names are hidden in the body of package woopsinfo. There is no need for the outside world to reference these names anyway, because an interface based on logical synonyms is provided. From within package woopsinfo, only two functions named 'ListX' and 'NodeY' are used to access WPP-specific subtrees. In this way it should be of very little effort to adjust the GNAT-WPP tree-connection to further releases of GNAT.

3.3.4 Flags

WPP also makes use of a couple of *Boolean* flags. The following flags are stored within the tree:

- *Case1*: Needed in conjunction with remainder function loops. True if the new value of the remainder function is specified as '*rem_identifier = expression*'.
- *Case2*: Needed in conjunction with remainder function loops. True if the new value of the remainder function is specified as '*rem_identifier <= upper_bound_expression*'.
- *Case3*: Needed in conjunction with remainder function loops. True if the new value of the remainder function is specified as '*rem_identifier <= upper_bound_expression and rem_identifier >= lower_bound_expression*'.
- *Custom_Reverse_Present*: True if keyword 'reverse' appears within a range_aggregate of an $N > 1$ dimensional discrete loop.
- *Implicit_Type_Required*: True if some range of a discrete loop is given in the form of a constrained_discrete_subtype_indication ([Ada95](3.2.2(3)). This means that during code generation an implicit subtype has to be declared (see Chapter 9).

- *Is_Discrete_Loop:* True for an N_Loop_Parameter_Specification that contains a discrete loop.

- *Needs_Package_WoopDefs*: False till the first occurrence of a discrete loop is detected. At this point this flag becomes 'True.' It is then checked whether package WoopDefs is already visible within the current compilation unit. If this is not the case, flag Needs_WoopDefs_Context_Clause is set.

- *Needs_WoopDefs_Context_Clause*: Set if in the code-generating phase a with_clause for package WoopDefs has to be issued.

The remaining attributes are synthesized from the tree.

- *Compilation_Unit*: For any node 'N' of the AST returns the node N_Compilation_Unit to which it belongs.

- *Dimensions*: Returns the number of parameters of the loop variable.

- *Fake_Initial_Value*: If an N-dimensional loop variable comprises M>N initial_values in its initial_value_aggregate, the first of those superfluous initial_values is returned (used for error messages during semantic analysis).

- *Fake_Discrete_Subtype_Definition*: Similar purpose as Fake_Initial_Value.

- *Fake_List_Of_Iteration_Functions*: Similar purpose as Fake_Initial_Value.

- *Has_Loop_Variable*: Needed in conjunction with remainder function loops. True if the loop contains a loop variable besides the remainder loop variable.

- *Has_Several_PSVs*: True in case of an N dimensional loop variable where the iteration_function_specification defines more then one vector of values. In case of a one-dimensional loop variable this is true if the list_of_iteration_functions includes more than one element.

- *Is_Loop_With_Remainder_Function*: True in case of a discrete loop with a remainder function.

- *Is_Monotonical_Loop*: True in case of a monotonical discrete loop.

- *LOIF_Nr_Of_Aggregates*: Returns the number of aggregates of an iteration_function_specification.

- *Needs_Lower_Range_Constant*: True if the discrete_subtype_definition of a monotonical discrete loop is given as a *range* with a non-static lower bound. In this case the code generator needs to generate a constant that is initialized with the lower bound of the range before execution of the loop. In this way the evaluate-once semantics of the range is ensured (see Chapter 9).
- *Needs_Upper_Range_Constant*: Similar to Needs_Lower_Range_Constant.
- *Nr_Of_Iteration_Functions*: Returns the number of iteration functions for a given dimension of a given aggregate of the loop variable.
- *Reverse_Present*: True if a given dimension of the loop variable is of a 'reverse' nature (either Custom_Reverse_Present or keyword **reverse** before range_aggregate).

Chapter 4
THE SCANNER

Instead of using freely available scanner- and parser generators like Lex and Yacc, the developers of GNAT chose to hand-code their own scanner and parser due to reasons of increased performance and user-friendliness. Since the source-files belonging to the scanner have a name starting with 'sc', they can be identified easily within the about 800 different files GNAT consists of. Concerning discrete loops, it takes only minor modifications to make the scanner aware of the additional token '*discrete*'.

Package scans defines an enumeration-type 'Token_Type' that is used to identify the tokens returned by the scanner. Subsequent subtype declarations define token-classes. Type 'Token_Type' is given in a table, where the classes applying to a token are given as comment lines. We have to add an enumeration-literal for token 'discrete'.

scans.ads

```
type Token_Type is (

--   Token name              Token type    Class(es)

    Tok_Integer_Literal, -- numeric lit   Literal, Lit_Or_Name
    Tok_Real_Literal, -- numeric lit   Literal, Lit_Or_Name

    ...

    Tok_Begin, -- BEGIN        Eterm, Sterm, After_SM, Labeled_Stmt
    Tok_Declare, -- DECLARE     Eterm, Sterm, After_SM, Labeled_Stmt

--   BB: entered Token Tok_Discrete for the keyword 'discrete'.
    Tok_Discrete, -- DISCRETE   Eterm, Sterm, After_SM, Labeled_Stmt

    Tok_For, -- FOR          Eterm, Sterm, After_SM, Labeled_Stmt
    Tok_Loop, -- LOOP         Eterm, Sterm, After_SM, Labeled_Stmt
    Tok_While, -- WHILE        Eterm, Sterm, After_SM, Labeled_Stmt

    ...

    );
```

From the scanner's point of view, there is no difference between a discrete loop and any other kind of loop provided by the Ada programming language. Therefore our token belongs to the same classes that the tokens 'for', 'loop' and 'while' do:

- Eterm which means that the token is an expression terminator (it can never appear within a simple expression). This is used for error recovery purposes (if GNAT encounters an error in an expression, it simply scans to the next Eterm token).
- Sterm: Terminator for simple expressions.
- After_SM: Tokens which always, or almost always, appear after a semicolon. Of significance only for error recovery.
- Labeled_Stmt: Tokens which start labeled statements.

The specification of package Snames contains definitions of standard names (i.e. entries in the Names table) that are used throughout the GNAT compiler. Upon initialization of the scanner, they are loaded into the Names table. These also include all of Ada95's reserved words, since they ought to be recognized by the scanner. Names are defined as constants that derive their values depending on some predefined constants plus their respective positions within this spec. Every additional token increases the respective positions of all the following tokens in the spec. In this way not only their positions but also their values are changed. In order to keep the number of obsolete definitions low, the name for token 'discrete' is inserted at the end of the reserved words:

snames.ads

```
Name_Abort : constant Name_Id := N + 313;
Name_Abs : constant Name_Id := N + 314;
Name_Accept : constant Name_Id := N + 315;
Name_And : constant Name_Id := N + 316;
Name_All : constant Name_Id := N + 317;
Name_Array : constant Name_Id := N + 318;

  ...

Name_When : constant Name_Id := N + 368;
Name_While : constant Name_Id := N + 369;
Name_With : constant Name_Id := N + 370;
```

```
Name_Xor : constant Name_Id := N + 371;

--  BB: Add constant Name_Id for keyword 'discrete'.
Name_Discrete : constant Name_Id := N + 372;
```

As stated above, one has to be added to the declaration of all the constants that follow Name_Discrete in the file. Note that constants with the syllable 'First_' pass their value over to the following constant, whereas constants with the syllable 'Last_' take their value from the previous constant. This scheme is used for grouping purposes.

The string-representation of the new token must be provided in the package body of Snames. It is important that the token appears at the same position as in the spec. Note also that every string is terminated by '#'.

snames.adb

```
Preset_Names : constant String :=
  "_abort_signal#" &
  "_assign#" &
  "_chain#" &

   ...

  "when#" &
  "while#" &
  "with#" &
  "xor#" &

--  BB: added string representation of keyword 'discrete'.
  "discrete#" &

   ...;
```

It is the task of the scanner to return the appropriate token for each sequence of characters that occurs in the input. To put it GNAT-specific, it has to establish a mapping from the input to the values of type Token_Type. In the case of reserved words, this is done with help of the names table. At scanner initialization time, the whole set of reserved words is loaded into the names table. Every entry that holds the string representation of such a reserved word uses an additional component of the GNAT-internal type *Byte* to identify the position number of the reserved word's enumeration literal within type Token_Type. For every other entry in the names table, the value of this additional component is zero.

Whenever the scanner encounters a sequence of characters, it consults the names table looking for an entry already representing this sequence. If there exists an entry, this can either be a reserved word or a previously stored identifier. In case of a reserved word, Token_Type'Val of the Byte-component mentioned above returns the appropriate token.

Therefore the only thing that remains to be done is to modify the initialization routine of the scanner that loads the names table.

scn.adb

```
procedure Initialize_Scanner
  (Unit : Unit_Number_Type;
   Index : Source_File_Index)
is
begin
--  Set up Token_Type values in Names Table entries for reserved keywords
--  We use the Pos value of the Token_Type value. Note we are relying on
--  the fact that Token_Type'Val (0) is not a reserved word!

--  BB: For Name_Discrete: Set the corresponding position for
--  'Tok_Discret' in the Name_Table.
   Set_Name_Table_Byte (Name_Discrete, Token_Type'Pos (Tok_Discrete));

   Set_Name_Table_Byte (Name_Abort, Token_Type'Pos (Tok_Abort));
   Set_Name_Table_Byte (Name_Abs, Token_Type'Pos (Tok_Abs));

   ...

end Initialize_Scanner;
```

Since for this operation the sequence of statements is insignificant, the code has been added right at the beginning of procedure Initialize_Scanner.

Chapter 5
SYNTACTIC ANALYSIS

The Par function and its subunits contain all the parsing routines for GNAT's top-down recursive descent parser that constructs the Abstract Syntax Tree. Within function Par the parsing routines are grouped by chapters as given in [Ada95]. Every chapter is reflected as a package containing parse-functions for its constructs. The bodies of those packages are given as stubs, which means that they reside in separate files. The naming convention for such files is defined in [LIB94]. To put it in a nutshell, the file-layout for the parser looks as follows:

- *par.ads*: Spec of function Par.
- *par.adb*: Body of function Par.
- *par-ch2.adb* ... *par-ch13.adb*: Parse-functions for constructs of corresponding chapters of [Ada95].

Loops belong to statements, which are treated in Chapter 5. Therefore we add function P_Discrete_Statement in the body of package Par.Ch5 and make it a subunit:

par-ch5.adb

```
function P_Loop_Statement (Loop_Name : Node_Id := Empty) return Node_Id;
--  Parse loop statement. If Loop_Name is non-Empty on entry, it is
--  the N_Identifier node for the label on the loop. If Loop_Name is
--  Empty on entry (the default), then the loop statement is unlabeled.

function P_For_Statement (Loop_Name : Node_Id := Empty) return Node_Id;
--  Parse for statement. If Loop_Name is non-Empty on entry, it is
--  the N_Identifier node for the label on the loop. If Loop_Name is
--  Empty on entry (the default), then the for statement is unlabeled.

function P_Discrete_Statement (Loop_Name : Node_Id := Empty) return Node_Id;
--  Parse discrete statement. If Loop_Name is non-Empty on entry, it is
--  the N_Identifier node for the label on the loop. If Loop_Name is
--  Empty on entry (the default), then the discrete statement is unlabeled.

function P_Discrete_Statement (Loop_Name : Node_Id := Empty) return Node_Id is
separate;
```

On encountering token *Tok_Discrete*, the parser has to call this function. This happens two times within function P_Sequence_Of_Statements.

par-ch5.adb

```
-- Loop_Statement (labeled Loop_Statement)

if Token = Tok_Loop then
 Append_To (Statement_List,
  P_Loop_Statement (Id_Node));

-- BB: Entered function call to P_Discrete_Statement.
-- (labeled discrete statement)
elsif Token = Tok_Discrete then
 Append_To (Statement_List, P_Discrete_Statement (Id_Node));
```

The first call deals with labeled discrete loops, a similar call deals with loops without a label. It is the duty of function P_Discrete_Statement to parse the whole discrete loop and to return a subtree as specified in Chapter 3 which is then appended to the Abstract Syntax Tree. The following tasks are carried out within this function:

1. Push a new entry on the scope stack [Ada95](3.1(8)). This is necessary for all nested constructs in order to deal with 'END' nesting errors.

2. Create a new node N_Loop_Statement.

3. Connect a new N_Iteration_Scheme node to node N_Loop_Statement.

4. Parse the discrete_loop_parameter_specification and connect the resulting tree to node N_Iteration_Scheme. This is a tedious task that has to take into account all syntactic variations between monotonical loops and loops with remainder functions, both single- and multi-dimensional. Due to this fact the declaration of a separate function within function P_Discrete_Statement seemed appropriate. Its name is P_Discrete_Loop_Parameter_Specification and it is also made into a subunit.

5. Parse the loop body and connect the resulting tree to node N_Loop_Statement. This task is rather simple, because the body of a discrete loop does not syntactically differ from any other loop body. It is therefore sufficient to call function P_Sequence_Of_Statements that is already provided in package Par.Ch5.

6. Set attribute Is_Discrete_Loop.

Chapter 6
SEMANTIC ANALYSIS

Semantic analysis in general performs name and type resolution, decorates the AST with various attributes and performs all static legality checks on the program. Type resolution is done using a two-pass algorithm. During the first (bottom-up) pass, each node within a complete context is labeled with its type, or if overloaded with the set of possible meanings of each overloaded reference. During the second (top-down) pass, the type of the complete context is used to resolve ambiguities and choose a unique meaning for each identifier in an overloaded expression [GNAT]. In the case of a loop statement the loop's *iteration_scheme* as well as the *sequence_of_statements* have to be analyzed.

The body of a discrete loop differs from the body of a for-loop only in one sense: The discrete loop variable behaves like a variable rather than a constant within the body of the loop. It is therefore possible to assign a value to a discrete loop variable within the loop body. With that minor difference in mind we can leave the latter task to GNAT using procedure Analyze_Statements of package Sem_Ch5 in order to analyze the loop body.

The sole purpose of the semantic analysis of the *iteration_scheme* of a discrete loop is to determine the type of the loop variable (and/or remainder loop variable) and to verify that the type is appropriate for the given type of loop.

6.1 Semantic Analysis of One-Dimensional Monotonical Discrete Loops

The discrete_loop_parameter_specification of an iteration_scheme contains three entities that provide type information:

- The expression representing the *initial_value*.
- The *discrete_subtype_definition*.
- The iteration functions contained in the *list_of_iteration_functions*.

The algorithm that is used for type resolution is depicted in pseudo-code in Figure 15. It starts with calls to procedure *Analyze* to carry out the bottom-up pass for the loop's *init_id* and *discrete_subtype_definition*. This leaves us with at least one possible type for both arguments. Strictly speaking, we get one type for non-overloaded entities and N ($N \in \mathbf{N}, N > 1$) for overloaded ones. Code lines (3) to (17) aim at computing the intersection between these two sets of types. To avoid ambiguity, we require that the resulting set contains at most one type. An empty set means that *init_id* and *discrete_range* were type-incompatible which is equivalent to finding a bad type. To recover from such an error, bad types are treated as 'type wild-cards' that match any given type (they are said to be of type 'Any_Type'). This is exactly the difference between statement *issue_error* (lines 2 & 15) and *return_error* (lines 21 & 31): the first one complains put continues execution, whereas the latter one quits issuing an error-message. Note that the algorithm does in no way terminate before having passed line 19! This ensures that the loop variable is visible during subsequent analysis of the loop body. Furthermore it allows a loop variable to appear in an *iteration_function.*

The remaining part of the algorithm (lines 22 - 40) distinguishes between two cases:

- We have found type 'Universal_Integer', which makes us consult the *iteration functions* to find some specific integer-type (line 29). If the iteration functions do not yield a specific type either, then 'Universal_Integer is casted to 'integer'.

- We have already found a specific type that has to fit the *iteration_functions.*

In order to avoid anonymous base types (which cannot be referenced in the source code that WPP creates) we have to compute their so-called *first named subtypes*[1] (lines 23 & 35). What remains to be done is to resolve all three entities *(init_id, discrete_subtype_definition & list_of_iteration_functions)* with the type derived. This corresponds to pass two (top-down) of GNAT's type resolution algorithm.

6.2 Semantic Analysis of Multi-Dimensional Monotonical Discrete Loops

Unfortunately this is a bit more complicated than the previous one. First of all, multi-dimensional loops add a new dimension of errors on the programmer's side. New checks are introduced accordingly:

- An identifier must not occur twice within a loop variable.

[1]First named subtypes are somewhat explained in [Ada95](3.2.1(7))

- The number of dimensions of the initial_value_aggregate, range_aggregate, identifier_aggregate after 'new', and iteration_function_specification must be the same as that of the identifier_aggregate at the beginning of the loop.
- The names that occur in the N dimensions of the identifier_aggregate after keyword 'new' must match their counterparts in the identifier_aggregate at the beginning of the loop statement.
- Keyword 'reverse' can occur right before the range_aggregate that contains N discrete_subtype_definitions. This means that keyword 'reverse' stands for all N dimensions of the loop variable. On the other hand keyword 'reverse' can also appear inside the range_aggregate, right before any of the discrete_subtype_definitions. In this case it *must not* occur outside the range_aggregate too.

Note that one can argue whether these are syntactic or semantic checks. The reason that they are carried out during semantic analysis is that we can give better user-feedback from here.

The second reason that makes semantic analysis of multi-dimensional monotonical discrete loops a bit more complicated is that they allow mutual dependencies between single parameters of the loop variable within the iteration functions. Consider the following loop which has been taken from Euclid's Algorithm (Figure 6, page 11).

```
1  discrete (M,N) := (A,B) in reverse (1..A, 1..B)
2   new (M,N) := (N, M mod N) loop
3     H := M;
4     M := N;
5     N := H mod N;
6  end loop;
```

Figure 16: Discrete Loop taken from Euclid's Algorithm

The loop variable of this loop consists of two parameters namely *M* and *N*. The iteration_function_specification (line 2) has it that parameter M depends on parameter N (it is assigned N's value), whereas parameter N depends on M and N. If type resolution was done on a per-dimension basis using the algorithm for one-dimensional monotonical discrete loops, this circular dependency could not be resolved, because at the time the first dimension is analyzed, the type system knows nothing about parameter N. Switching dimensions does not help either because of the dependency of parameter N on M and N. For this reason it is necessary to make all parameters of a multi-dimensional loop variable visible before any of the iteration functions can be analyzed.

Figure 17 may serve as a (hypothetical) example for another (hypothetical) problem that can occur with multi-dimensional discrete loops. The point with this example is that initial_value and discrete_subtype_definition of all parameters of the loop variable are of type 'Universal_Integer'.

```
1   discrete (A,B,C) := (1,2,3) in (1..10,1..10,1..10)
2    new (A,B,C) := (A+1,
3                    Short_Integer (B) + A,
4                    Long_Integer (C) + A)
5   loop
6   --  loop body
7   end loop;
```

Figure 17: Hypothetical Discrete Loop

Our type resolution algorithm for one-dimensional discrete loops suggests to consult the iteration functions for more specific type information in such a case. However, because of possible mutual dependencies of parameters this information may be spread across all N dimensions of the iteration_function_specification. Consider parameter A of Figure 17:

- The first dimension (line 2) proposes A to be of type 'Universal_Integer' (casted to 'Integer').
- Dimension 2 casts B to 'Short_Integer', which also requires A to be of type 'Short_Integer', provided that '+' is not overloaded with a function capable of adding objects of type 'Short_Integer' and 'Integer'.
- Dimension 3 finally requires A to be of type 'Long_Integer', provided that '+' is not overloaded either.

Obviously the requirements of the second and third dimension contradict each other, but this example shows that not only data but also type-information is propagated between different dimension's iteration functions. A type resolution algorithm capable of such cases must implement some sort of backtracking that verifies that typing is still consistent within the iteration_function_specification after a new type has been derived. However this is only necessary for loops where a parameter's type is only determined in the iteration_function_specification. Since this requires the initial_value and the discrete_subtype_definition to be of type 'Universal_Integer', this seldomly is the case. For this reason the current implementation of WPP requires the type of a *multi*-dimensional discrete loop's parameter to be determined solely from the initial_value and the discrete_subtype_definition. If the type of both constructs is 'Universal_Integer', then the type is casted to 'Integer'. If this introduces a type clash with the iteration_function_specification, then the programmer has to convert the initial_value or discrete_subtype_definition to the desired type manually (confer Figure 18).

```
discrete (A,B,C) := (1, Short_Integer (2), Long_Integer (3))
 in (1..10,1..10,1..10) new (A,B,C) := (A+1,
  Short_Integer (B) + Short_Integer (A),
  Long_Integer (C) + Long_Integer (A))
loop
--  loop body
end loop;
```

Figure 18: Type Conversation for Hypothetical Discrete Loop

The algorithm that is used for type resolution of multi-dimensional monotonical discrete loops is depicted in pseudo-code in Figure 19.

- Part one (lines 1 to 33) consists of a loop that iterates over all dimensions of the discrete loop under consideration. For every dimension it attempts to derive a type from the initial_value and the discrete_subtype_definition. Thereafter the loop parameter for the current dimension is made visible. In case of an appropriate type the initial_value and the discrete_subtype_definition are resolved.

- Part two (lines 34 to 39) analyzes each dimension's iteration functions and resolves them with the type derived for that dimension in Part one.

6.3 Semantic Analysis of Discrete Loops with Remainder Functions

6.3.1 Semantic Analysis of the Loop Variable

Discrete loops with remainder functions can (but need not) contain the declaration of a loop variable. Contrary to monotonical discrete loops this declaration does not contain a discrete_subtype_definition. The loop variable of a remainder function loop may be of any type except *limited* or *abstract*. Again type resolution distinguishes between one- and multi-dimensional loops.

One-Dimensional Loops

The algorithm used for type resolution is depicted in pseudo-code in Figure 20. In line 1 it is checked whether the loop variable and the remainder loop variable denote the same entity which of course is not allowed. Lines 4 to 10 attempt to derive a type from the initial_value construct. The call to function Yields_Appropriate_Type (line 5) removes all limited as well as abstract types from the typeset computed by the call to procedure Analyze. If the resulting typeset contains more than one possible type we have to issue an error, because unlike monotonical discrete loops the initial_value construct is the only source of type-information here. Only in the case of a universal type (Universal_Integer or Universal_Real) the list_of_iteration_functions is consulted to find a more specific type (lines 13 - 26).

Multi-Dimensional Loops

The following checks have to be introduced with multi-dimensional loops:

- An identifier must not occur twice within a loop variable.
- The number of dimensions of the initial_value_aggregate, identifier_aggregate after 'new', and iteration_function_specification must be the same as that of the identifier_aggregate at the beginning of the loop.
- The names that occur in the N dimensions of the identifier_aggregate after keyword 'new' must match their counterparts in the identifier_aggregate at the beginning of the loop statement.

Type resolution is depicted in pseudo-code in Figure 21. Because of possible mutual dependencies between parameters of the loop variable within the iteration functions all the parameters of the loop variable have to be visible before any of the iteration functions can be analyzed. Therefore the type of a loop parameter is solely determined by the initial_value construct.

- The first part of the type resolution algorithm (lines 1 to 22) consists of a loop that iterates over all dimensions of the discrete loop under consideration. For each dimension it attempts to derive a type from the initial_value construct.

Thereafter the loop parameter for the current dimension is made visible. In case of an appropriate type the initial_value is resolved.

- Part two (lines 23 to 28) analyzes each dimension's iteration functions and resolves them with the type derived for that dimension in part one.

6.3.2 Semantic Analysis of the Remainder Loop Variable

The purpose of semantic analysis of the remainder loop variable is the derivation of a type for the remainder loop variable. As stated in Section 1.1.2 we expect the remainder loop variable to be of subtype natural or of a subtype that has natural among its ancestors. Type resolution of the remainder loop variable makes use of function Yields_Natural several times. The purpose of this function is to take the typeset of an expression-node computed by a call to procedure Analyze and to remove all types except natural or subtypes that have natural among its ancestors. If the resulting typeset contains more than one type, flag Is_Overloaded is set on the node. Yields_Natural returns true if the resulting typeset contains at least one such type.

The algorithm used for type resolution of the remainder loop variable is depicted in pseudo-code in Figure 22. It starts with the initialization of *rem_init_id* (which is set to the remainder loop variable's initial_value), *exp1* (which is set to the expression (or upper_bound_expression) of the remainder_function), and *exp2* (which is set to the lower_bound_expression of the remainder_function). Thereafter a call to Analyze on the rem_init_id is made. Lines 3 to 10 attempt to pick an appropriate type from the resulting typeset. If it turns out to be of type Universal_Integer we have to postpone type derivation until exp1 (and exp2) have been analyzed in order to find a more specific discrete type. A rem_init_id that is overloaded with two or more specific types causes an error, because no other construct containing further type information is available.

Note 6.3.1 Exp1 and exp2 cannot be used for that purpose, because the remainder loop variable is allowed to occur in those constructs. This makes it necessary to set the remainder loop variable to an appropriate type (at least Universal_Integer) before exp1 and exp2 can be analyzed.

Remainder loop variables of *Case1* remainder loops are declared as being *constant*. In this way semantic analysis will not allow the user to assign a value to the remainder loop variable in the loop body. *Case2* and *Case3* remainder loop variables are treated as *variables* (lines 11 & 12). Note that the algorithm does in no way terminate before having passed line 12. This ensures that the remainder loop variable is visible during subsequent analysis of the loop body. Furthermore it allows a remainder loop variable to appear within a remainder_function. The remaining part of the algorithm distinguishes between two cases:

- We have found 'Universal_Integer' which makes us consult *exp1* (and *exp2*) in order to find some specific type (lines 19 to 39).
- We have already found a specific type that has to fit the *remainder_function* (line 17).

Calls to procedure Resolve finally finish the derivation of a type for the remainder loop variable.

6.4 Pragma Restrictions (No_General_Loops)

Package *restrict* is solely devoted to the implementation of pragma restrictions. It defines an enumeration type called *Restriction_Id* which contains all possible restrictions as literals. In conformance to [Ada95, 13.12] these are called restriction identifiers. In order to make WPP recognize a new restriction, a restriction identifier has to be added to this type.

restrict.ads

```
type Restriction_Id is (
   Immediate_Reclamation,                         -- (RM H.4(10))
   No_Abort_Statements,                    -- (RM D.7(5), H.4(3))
   No_Access_Subprograms,                         -- (RM H.4(17))
   No_Allocators,                                  -- (RM H.4(7))

     ...

--  BB: added pragma No_General_Loops
   No_General_Loops,                                       -- WPP

     ...

   No_Unchecked_Deallocation,                      -- (RM H.4(9))
   Not_A_Restriction_Id);
```

All requests for restrictions are stored in a data structure defined in package restrict. It also provides the following functions which operate on this data structure. Their aim is to check whether the restriction corresponding to the given restriction identifier has been requested by the user. If this is the case, these functions post an appropriate error message.

restrict.ads

```
procedure Check_Restriction (R : Restriction_Id; N : Node_Id);
--  Checks that the given restriction is not set, and if it is set,
--  an appropriate message is posted on the given node.

procedure Check_Restriction
   (R : Restriction_Parameter_Id;
    N : Node_Id);
--  Checks that the given restriction parameter identifier is not set to
--  zero. If it is set to zero, then the node N is replaced by a node
--  that raises Storage_Error, and a warning is issued.

procedure Check_Restriction
   (R : Restriction_Parameter_Id;
    V : Uint;
    N : Node_Id);
--  Checks that the count in V does not exceed the maximum value of the
--  restriction parameter value corresponding to the given restriction
--  parameter identifier (if it has been set). If the count in V exceeds
--  the maximum, then post an error message on node N.
```

What remains to be done is to make WPP call one of the above functions whenever it encounters a loop that might be rejected in the presence of pragma restrictions (No_General_Loop). This can be achieved at that point during semantic analysis where the kind of a given loop is determined. Procedure *Analyze_Loop_Statement* performs semantic analysis of a loop. If the given loop has no iteration_scheme, we have encountered a general loop that certainly is subject to our restriction:

sem_ch5.adb

```
if No (Iteration_Scheme (N)) then
--  Infinite loop
--  BB: In case of a pragma restrictions (No_General_Loops)
--  we have to complain now!
    Check_Restriction (No_General_Loops, N);
else
    Analyze_Iteration_Scheme (Iteration_Scheme (N));
end if;
```

In case of an iteration_scheme the loop can either be a *discrete*,- *while*- or *for*-loop. If the loop turns out to be a *while*- loop, we have to check for the presence of pragma restrictions (No_General_Loop).

sem_ch5.adb

```
if Present (Cond) then
--  BB: In case of pragma restrictions (No_General_Loops)
--  we don't appreciate while-loops either.
    Check_Restriction (No_General_Loops, N);
    Analyze_And_Resolve (Cond, Any_Boolean);
else
--  Else we have a FOR loop
    ...
end if;
```

```
Analyze (init_id); Analyze (discrete_subtype_definition);

if they do not yield discrete types then issue_error; end if;

if Is_Overloaded (init_id) and Is_Overloaded (discrete_subtype_definition) then
    T := Compute_the_Intersection_of_both_Typesets;
elsif Is_Overloaded (init_id) and not Is_Overloaded (discrete_subtype_definition) then
    if Typeset(init_id) contains Type(discrete_subtype_definition) then
        T := Type(discrete_subtype_definition);
    end if;
elsif not Is_Overloaded (init_id) and Is_Overloaded (discrete_subtype_definition) then
    if Typeset(discrete_subtype_definition) contains Type(init_id) then
        T := Type(init_id);
    end if;
else
    if Type(init_id) not_compatible_to Type(discrete_subtype_definition) then
        issue_error;
    end if;
end if;

Make the entity of the loop variable visible;

if no suitable type found then return_error; end if;

if T / = Universal_Integer then
    T := First_Named_Subtype (Base_Type (T));
    Resolve_Init_Id (T);
    Resolve_Discrete_Subtype_Definition (T);
    Analyze_List_Of_Iteration_Functions;
    Resolve_List_Of_Iteration_Functions (T);
else
    Analyze_List_Of_Iteration_Functions;
    T := Determine_Type_of_Iteration_functions;
    if T = Any_Type then return_error; end if;
    if T = Universal_Integer then
        T := Standard_Integer;
    else
        T := First_Named_Subtype (Base_Type (T));
    end if;
    Resolve_Init_Id (T);
    Resolve_Discrete_Subtype_Definition (T);
    Resolve_List_Of_Iteration_Functions (T);
end if;
```

Figure 15: Pseudo-code type resolution of a one-dimensional monotonical discrete loop.

```
for Index in 1 .. Dim (L) loop

    init_id := Initial_Value (L, Index);
    discrete_subtype_definition := Discrete_Subtype_Definition (L, Index);

    Analyze (init_id); Analyze (discrete_subtype_definition);
    if they do not yield discrete types then issue_error; end if;

    if Is_Overloaded (init_id) and Is_Overloaded (discrete_subtype_definition) then
        T (L, Index) := Compute_the_Intersection_of_both_Typesets;
    elsif Is_Overloaded (init_id) and not Is_Overloaded (discrete_subtype_definition) then
        if Typeset(init_id) contains Type(discrete_subtype_definition) then
            T (L, Index) := Type(discrete_subtype_definition);
        end if;
    elsif not Is_Overloaded (init_id) and Is_Overloaded (discrete_subtype_definition) then
        if Typeset(discrete_subtype_definition) contains Type(init_id) then
            T (L, Index) := Type(init_id);
        end if;
    else
        if Type(init_id) not_compatible_to Type(discrete_subtype_definition) then
            issue_error;
        end if;
    end if;

    Make the parameter of the Ith dimension of the loop variable visible;

    if suitable type found then
        if T (L, Index) = Universal_Integer then
            T (L, Index) := Standard_Integer;
        else
            T (L, Index) := First_Named_Subtype (Base_Type (T (L, Index)));
        end if;
        Resolve (Initial_Value (L, Index), T);
        Resolve (Discrete_Subtype_Definition (L, Index), T);
    end if;

end loop;

for Aggregate in 1 .. LOIF_Nr_Of_Aggregates (L) loop
    for Index in 1 .. Dimensions (L) loop
        Analyze (List_Of_Iteration_Functions (L, Index, Aggregate));
        Resolve (List_Of_Iteration_Functions (L, Index, Aggregate), T (L, Index));
    end loop;
end loop;
```

Figure 19: Pseudo-code type resolution of a multi-dimensional monotonical discrete loop.

```
if loop_variable = remainder_loop_variable then return_error; end if;

init_id := Initial_Value (L, 1);
list_of_iteration_functions := List_Of_Iteration_Functions (L, 1);

Analyze (init_id);
if not Yields_Appropriate_Type (init_id) then issue_error; end if;
if Is_Overloaded (init_id) then
    issue_error;
else
    T := Type (init_id);
end if;

Make the entity of the loop variable visible.

if no suitable type found then return_error; end if;

if Is_Universal_Type (T) then
    Analyze (list_of_iteration_functions);
    T := Determine_Type (list_of_iteration_functions);
    if T = Any_Type then
        return_error;
    elsif T = Universal_Integer then
        T := Standard_Integer;
    elsif T = Universal_Real then
        T := Standard_Float;
    else
        T := First_Named_Subtype (Base_Type (T));
    end if;
    Resolve (init_id, T);
    Resolve (list_of_iteration_functions, T);
else
    T := First_Named_Subtype (Base_Type (T));
    Resolve (init_id, T);
    Analyze (list_of_iteration_functions);
    Resolve (list_of_iteration_functions, T);
end if;
```

Figure 20: Pseudo-code type resolution of a one-dimensional loop variable of a remainder function loop.

```
for Index in 1 .. Dim (L) loop

    if loop_parameter (index) = remainder_loop_variable then return_error; end if;

    init_id := Initial_Value (L, Index);

    Analyze (init_id);
    if not Yields_Appropriate_Type (init_id) then issue_error; end if;
    if Is_Overloaded (init_id) then
        issue_error;
    else
        T (L, Index) := Type (init_id);
    end if;

    Make the parameter of the Ith dimension of the loop variable visible;

    if suitable type found then
        if T (L, Index) = Universal_Integer then
            T (L, Index) := Standard_Integer;
        elsif T (L, Index) = Universal_Real then
            T (L, Index) := Standard_Float;
        else
            T (L, Index) := First_Named_Subtype (Base_Type (T (L, Index)));
        end if;
        Resolve (Initial_Value (L, Index), T (L, Index));
    end if;

end loop;

for Aggregate in 1 .. LOIF_Nr_Of_Aggregates (L) loop
    for Index in 1 .. Dimensions (L) loop
        Analyze (List_Of_Iteration_Functions (L, Index, Aggregate));
        Resolve (List_Of_Iteration_Functions (L, Index, Aggregate), T (L, Index));
    end loop;
end loop;
```

Figure 21: Pseudo-code type resolution of a multi-dimensional loop variable of a remainder function loop.

```
rem_init_id := Rem_Initial_Value (L); exp1 := Expression1 (L); exp2 := Expression2 (L);

Analyze (rem_init_id);
if Type (rem_init_id) / = Universal_Integer and then not Yields_Natural (rem_init_id) then
    issue_error;
end if;
if Is_Overloaded (rem_init_id) then
    issue_error;
else
    T := Type (rem_init_id);
end if;
if Case1 (L) then declare entity for remainder loop variable as constant;
else declare entity for remainder loop variable as variable; end if;
if no suitable type found then return_error; end if;

Analyze (exp1);
if Case3 (L) then Analyze (exp2); end if;
if T / = Universal_Integer then
    Resolve (rem_init_id, T);
else
    if Case1 (L) or Case2 (L) then
        if not Yields_Natural (exp1) or Is_Overloaded (exp1) then return_error; end if;
        T := Type (exp1);
    else --  Case3
        if not Yields_Natural (exp1) or not Yields_Natural (exp2) then return_error; end if;
        if Is_Overloaded (exp1) and Is_Overloaded (exp2) then
            T := Compute_the_Intersection_of_both_Typesets;
        elsif Is_Overloaded (exp1) and not Is_Overloaded (exp2) then
            if Typeset (exp1) contains Type (exp2) then
                T := Type (exp2);
            end if;
        elsif not Is_Overloaded (exp1) and Is_Overloaded (exp2) then
            if Typeset (exp2) contains Type (exp1) then
                T := Type (exp1);
            end if;
        else
            if Type (exp1) = Type (exp2) then
                T := Type (exp1);
            end if;
        end if;
        if no suitable type found then return_error; end if;
        Resolve (rem_init_id, T);
    end if;
end if;

Resolve (exp1, T);
if Case3 (L) then Resolve (exp2, T); end if;
```

Figure 22: Pseudo-code type resolution of a remainder loop variable.

Chapter 7

ESTIMATING THE NUMBER OF ITERATIONS OF DISCRETE LOOPS

7.1 Methods of Choice

According to [Bli94] there exist three methods suitable for the calculation of lower and upper bounds for the number of iterations of discrete loops:

- Solving involved recurrence relations.
- Exploiting theoretical foundations that apply to certain kinds of iteration- and remainder-functions ([Bli94], Section 4 & 6.3).
- Extensive Enumeration

7.1.1 Solving Involved Recurrence Relations

As an example we take the discrete loop of the Heapsort-algorithm depicted in Figure 3, page 5, derive the underlying recurrence relation and solve it. Clearly

```
discrete H := K in 1 .. N/2 new H := 2*H | 2*H+1 loop
-- Loop body
end loop;
```

Figure 23: Discrete loop taken from Heapsort

the number of iterations of this loop is bounded above by the length of $(h_\nu^{(min)})$ which fullfills the recurrence relation

$$\begin{aligned} h_1^{(min)} &= k \\ h_{\nu+1}^{(min)} &= 2h_\nu^{(min)} \end{aligned}$$

since the length of any loop sequence containing two successive elements that fulfill $h_{\nu+1} = 2h_\nu + 1$ will be smaller than that of $(h_\nu^{(min)})$. Solving the above recurrence relation we get

$$h_\nu^{(min)} = k2^{\nu-1}$$

We want to determine the value of ω such that

$$h_\omega^{(min)} \le N/2 \le h_{\omega+1}^{(min)}$$

Taking logarithms we obtain

$$\omega = \lfloor \mathrm{ld} N - \mathrm{ld} k \rfloor$$

for the number of iterations of the discrete loop. The method of choice for solving (more complicated) recurrence relations are generating functions and exponential generating functions. The same can be achieved with help of Mathematica (transcript of Mathematica session follows).

```
In[1]:= <<DiscreteMath`RSolve` -- Load package RSolve.

In[2]:= RSolve[{h[n+1]== 2 h[n], h[0] == k}, h[n], n]

Out[2]= {{h[n] -> (2^n k)/2}}

In[3]:= Solve [2^(n-1)k==N/2,n]

Out[3]= {{n -> Log[N/k]/Log[2]}}
```

7.1.2 Theoretical Foundations for Certain Iteration- and Remainder-Functions

Proofs for the following theorems can be found either in [Bli94] or in Appendix D. Here $f(x)$ is used to denote an iteration function. For convenience of the reader the numbering of the theorems is that of [Bli94].

Theorem 4.2

If $f(x) = \lceil \alpha x + \beta \rceil$, $\alpha > 1$, $\beta \geq 0$, the length of the corresponding loop sequence is bounded above by

$$\left\lfloor \log_\alpha \left(\frac{N(\alpha - 1) + \beta}{\alpha + \beta - 1} \right) + 1 \right\rfloor .$$

The corresponding lower bound is

$$\left\lfloor \log_\alpha \frac{N(\alpha - 1) + \beta + 1}{\alpha + \beta} + 1 \right\rfloor .$$

Theorem 4.3

If $f(x) = \lceil \alpha x^\gamma + \beta \rceil$, $\alpha > 1$, $\beta \geq 0$, $\gamma > 1$, the length of the corresponding loop sequence is bounded above by

$$\lfloor \log_\gamma((\gamma - 1)\log_\alpha N + 1) + 1 \rfloor.$$

The corresponding lower bound is

$$\left\lfloor \log_\gamma((\gamma - 1)\log_{\alpha+\beta+1} N + 1) + 1 \right\rfloor.$$

Theorem 4.4

If $f(x) = \lceil q(x) + \beta \rceil$, where $\beta \geq 0$, and $q(x) = \sum_i \alpha_i x^{\gamma_i}$, $\alpha_i > 1$, $\gamma_i > 1$, the length of the corresponding loop sequence is bounded above by

$$\lfloor \log_{\gamma_m}((\gamma_m - 1)\log_{\alpha_m} N + 1) + 1 \rfloor,$$

where the index m is defined such that $\gamma_m = \max_i \gamma_i$.

Theorem 4.5

If $f(x) = x + \beta, \beta > 0$ with initial value $k_1 = K$ the length of the corresponding loop sequence is bounded above by

$$\left\lfloor \frac{N - K}{\beta} + 1 \right\rfloor.$$

Theorem 6.1

If a loop sequence of remaining items fulfills

$$\begin{aligned} r_1 &= N \\ r_{\nu+1} &= \lfloor r_\nu / \mu \rfloor, \end{aligned}$$

where $\mu > 1$, then the length of the corresponding remainder loop sequence is bounded above by

$$\lfloor \log_\mu N + 2 \rfloor.$$

7.1.3 Extensive Enumeration

Extensive enumeration is the so-called 'brute-force' method we resort to when everything else fails. The extensive enumeration algorithm can handle a set of iteration functions (as opposed to the previous methods which can only be applied to single iteration functions. Its major drawback (besides being extensive) is that it cannot handle symbolic values. The algorithm is depicted in pseudo-code in Figure 24. For 'reverse' discrete loops and for remainder function loops the

```
1   c := 0;
2   cur := initial_value;
3   while cur in discrete_subtype definition loop
4       c := c+1;
5       if reverse loop or remainder function loop then
6           cur := max($f_i$(cur));
7       else
8           cur := min($f_i$(cur));
9       end if;
10  end loop;
```

Figure 24: Algorithm used for extensive enumeration of discrete loops.

minimum *min* must be replaced by the maximum (line 5). Variable c represents the counter for the number of iterations. It is initialized with zero. The current value of the loop variable is represented by *cur*. As long as the current value of the loop variable stays within the discrete_subtype_definition, the iteration functions are evaluated. Depending on the kind of loop, the greatest or the smallest value is added to the current value of the loop variable. After termination of the loop, variable c contains an upper bound for the number of iterations of the loop under consideration.

The methods explained so far work with one-dimensional monotonical discrete loops and with remainder function loops. For multi-dimensional monotonical discrete loops they have to be extended. The reason for this lies in the fact that with multi-dimensional monotonical discrete loops the behavior of the loop variable is determined by the *multi-dimensional* iteration_function_specification. An upper bound for the number of iterations of a multi-dimensional monotonical discrete loop is

$$\sum_{i=1}^{Dim(L)} \max UB(f_{i,j}),\ 1 \leq j \leq \sum_{m=1}^{M} K_{m,i}$$

where the $\max UB(f_{i,j})$ term denotes the maximum upper bound of all iteration functions of all aggregates of dimension i. By means of symbolic analysis of the loop body it should be possible to derive more accurate bounds.

7.2 The Mathematical Subsystem of WPP

Every piece of mathematics that WPP throws at discrete loops has its roots in package Math. The interface of this package is used as an abstraction between the implementation of the math-subsystem and its client. It provides a given functionality via a set of functions. The most prominent among them are:

- *Math_Initialize*: Initialization of the math subsystem.
- *Math_Evaluate (S : String; Kind : Evaluation)*: Have string 'S' evaluated by the math subsystem. If no error occurs during evaluation, Math_Evaluate returns true. The result can be fetched with function Math_Get_Expression. Parameter 'Kind' determines the kind of evaluation: numeric or symbolic.
- *Math_Get_Expression*: Returns the result of the last query.
- Cannot_Raise_Monotonic_Error: Takes an iteration function and a discrete interval [Lb,Ub]. Returns 'true' if the given iteration function cannot raise exception monotonic_error within the given interval. Note that the current implementation of the math subsystem does not support symbolic values for this function.
- *Monotonic_UB*: This function tries to derive an upper bound for the number of iterations of a monotonical discrete loop from the given list of iteration functions, the initial value, and the discrete subtype definition. If it succeeds, it returns 'true' and a string representing the upper bound can be fetched with function Math_Get_Expression. Note that the current implementation of the math subsystem does not support symbolic values for this function.
- *Math_End*: Finalizes the math subsystem.

7.2.1 Implementation

It has been stated in the section about implementation considerations that WPP's math subsystem utilizes the kernel of Mathematica. Communication takes place via *MathLink*, a general mechanism for exchanging mathematical expressions between

Mathematica and other programs. MathLink can use various data transport systems such as pipes or TCP. In this way it is even possible to talk to a Mathematica kernel hosted on a remote computer.

In fact MathLink is a set of library functions capable of initializing MathLink connections, of sending and receiving data, and of shutting down connections. Since the whole Mathematica system is coded in C, we had to bridge the gap between Ada and C by means of a binding (see Appendix C).

The first goal of the math subsystem is to trade expressions back and forth between Mathematica and WPP. This of course is implemented within the package body of Math. The second goal, however, is to carry out the computations that it takes in order to derive upper bounds or track down monotonic error exceptions. This part is implemented in Mathematica's internal Lisp-like programming language [WR4]. Its code is kept in a separate Mathematica-package that is loaded during initialization (function Math_Initialize) of the math subsystem. In this way it is possible to modify functions of the math subsystem without recompiling or even relinking of the preprocessor. Note also that by means of executing our code within the kernel of Mathematica we have the power of all of Mathematica's features [WO92] at hand.

Monotonic_UB

Semantic analysis is the last preprocessor phase that extracts information about program entities from the underlying source code. After semantic analysis is completed we know facts like the kind of the loop, the type of the loop (or remainder loop) variable, and whether the expressions contained are static or not. Any attempts aimed at determining the upper bound for the number of iterations of a discrete loop are therefore carried out right after semantic analysis of the given loop. The following code calls function Monotonic_UB for every dimension of a monotonical discrete loop:

sem_ch5-woopsem_ch5.adb

```
for Index in 1 .. Dimensions (L) loop
   if Monotonic_UB
      (List_Of_Iteration_Functions (L, Index),
       Identifier (L, Index),
       Discrete_Subtype_Definition (L, Index),
       Initial_Value (L, Index),
       Reverse_Present (L, Index))
   then
```

```
-- Fetch upper bound.
    else
-- No upper bound found.
    end if;
end loop;
```

At this point we are still talking about tree fragments that are passed to the math subsystem. List_Of_Iteration_Functions (L, Index) contains a List_Id with all the iteration functions of dimension 'index', identifier contains the loop variable and so on. Within function Monotonic_UB this call is transformed into an expression suitable for Mathematica.

As an example we take procedure Siftdown from the Heapsort algorithm (Figure 3):

```
N : constant Positive := 100;
subtype Index is Positive range 1 .. N;

...

procedure Siftdown(N,K : Index) is
    J : Index;
    V : Integer;
begin
    V := Arr(K);
    discrete H := K in 1 .. N/2 new H := 2*H | 2*H+1 loop
        J := 2*H;
        if J < N and then Arr(J) < Arr(J+1) then
            J := J+1;
        end if;
        if V >= Arr(J) then
            Arr(H) := V;
            exit;
        end if;
        Arr(H) := Arr(J);
        Arr(J) := V;
        H := J;
    end loop;
end Siftdown;
```

Figure 25: Procedure Siftdown from the Heapsort algorithm

If function Monotonic_UB is invoked for the discrete loop in procedure Siftdown, it transforms its arguments into expression

```
TimeConstrained[MonUB[{2 * h, 2 * h + 1},h,1,100],10,$Timeout]
```

that is then sent to Mathematica. `TimeConstrained[expr, t, failexpr]` is a built-in function of Mathematica. It attempts to evaluate *expr* within deadline *t*. If the deadline is missed, it returns *failexpr*. We use this function in order to put down extensive enumeration that turns out to be infeasible to be carried out in a reasonable amount of time. `MonUB[List, lv, lb, ub]` is a function that belongs to our Mathematica-package. It takes its arguments and tries to derive an upper bound for it. Note that the initial_value and the upper bound of the discrete_subtype_definition (line 9) have been replaced by the bounds of subtype *index* (line 2). The reason for this is that the current implementation of our Mathematica-package cannot handle symbolic values. In this way we replace every non-static entity by the corresponding bound of its type in order to derive a somewhat reasonable upper bound. An alternative would be to take *-Infinity* and *+Infinity*, but this would result in less accurate upper bounds...

Mathematica-package: function Mon_UB

```
MonUB[LoIF_List,lv_,lb_,ub_] :=
  If[IsPolynomialQ[LoIF,lv] && EnumerateQ[LoIF,lv,lb,ub],
     Block[{IFS,SmIF},
           IFS=RemoveDummies[Union[LoIF],lv];
           SmIF=SmallestIF[IFS,lv,lb,ub];
           (* Is there one smallest
              iteration function in [lb,ub] ? *)
           If[SmIF===$Failed,
              (* There is no single smallest iteration function,
                  so we enumerate all iteration functions *)
              Enumerate[LoIF,lv,lb,ub],
              (* There is a single smallest iteration function,
                 check whether a theorem applies to it *)
              If[TMon[SmIF,lv]===$Failed,
                 (* No theorem applicable
                    to single iteration function. *)
                 Enumerate[{SmIF},lv,lb,ub],
                 (* Theorem applicable
                    to single iteration function. *)
                 CalculateUB[SmIF,lv,lb,ub]
                ]
22              ]
23          ],
24      $Failed
25    ]
```

The above code is executed by Mathematica's kernel as soon as it gets our request from the link. It ensures that the given iteration functions are polynomials of the loop variable and that they do not contain any symbolic coefficients (line 2). Function RemoveDummies (line 4) deletes any iteration function f from the list $LoIF$ for which the following property holds:

$$f(x) \textsc{m}\ g(x)\ \forall x \in [-\infty,\ \infty],\ f(x), g(x) \in LoIF$$

This simply means that we forget about all iteration functions that cannot contribute to the upper bound because they compute a value which is greater (smaller if 'reverse' loop) than the value of some other iteration function (iteration function $2 * h + 1$ of procedure Siftdown is such a dummy). Function SmallestIF (line 5) tries to extract the iteration function that computes the smallest (greatest) value within the discrete interval $[lb,\ ub]$. If such an iteration function exists, we check whether some theorem applies to it (Theorem 4.2 for iteration function $2 * h$ of the above example). If this is the case, the upper bound is computed using this theorem, otherwise the iteration function(s) have to be enumerated. Although it has been stated at the beginning of this chapter that one could also attempt to solve the arising recurrence relation, our current implementation does not support it. The reason for this shortcut is that although Mathematica is good at solving recurrence relations, the solutions often involve transcendental functions as soon as they are put into equations. Unfortunately Mathematica is not quite as good with that...

The result of function Mon_UB (7 in the above example) is put back on the link by the kernel where it is read by function Monotonic_UB.

Cannot_Raise_Monotonic_Error

The aim of this function is to determine whether a given iteration function can raise exception monotonic_error or not. An iteration function $f(x)$ cannot raise exception monotonic_error in a given discrete interval [lb, ub], if

$$x \textsc{m}\ f(x)\ \forall x \in [lb, ub].$$

Consider the iteration function $2x^4 - 3x^2 + x$ depicted in Figure 26. Within the discrete intervals $[-\infty, -2]$ and $[2, \infty]$ the values computed by the iteration function are smaller than x (the graph of the iteration function is above the line denoting x. This means that the iteration function cannot raise monotonic error in case of a monotonically increasing discrete loop ($f(x) > x$). However, this does not hold for the interval $[-1, 1]$.

In order to determine whether a given iteration function $f(x)$ can raise exception monotonic_error we therefore have to intersect the iteration function with x and examine the resulting intervals: If we can show that the bounds of the discrete_subtype_definition of the loop are within an interval where x ᴍ $f(x)$ holds, then it is save to omit the run-time checks for exception monotonic_error.

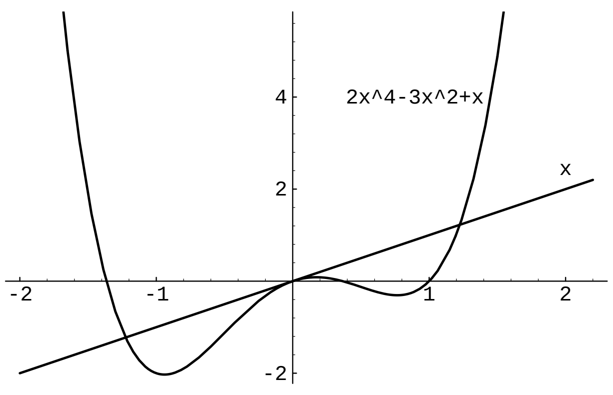

Figure 26: Plot of x and $f(x) = 2x^4 - 3x^2 + x$

Mathematica-package: function NoMonotonicError

```
(* NoMonotonicError[LoIF_List,lv_,lower_,upper_,cond_]
   Returns ´True´ if the the iteration functions in LoIF cannot
   raise monotonic_error in the given interval [lower,upper].
   ´cond´ is used to distinguish between increasing and
   decreasing (keyword ´reverse´) iteration functions. It is
   either FXGreaterX (increasing) or FXSmallerX (decreasing).
   Requirements: LoIF : must be polynomials in lv, no symbolic
   coefficients or exponents.
   lower, upper : non-symbolic values .
*)

NoMonotonicError[LoIF_List,lv_,lower_,upper_,cond_] :=
 If[!(IsPolynomialQ[LoIF,lv]
      && EnumerateQ[LoIF,lv]
      && BEnuQ [lower,upper]),
   False,
   Block[{CommonInterval := IntersectIntervals
          [Map[ToInterval[Intervals[#,lv,cond]]&,LoIF]]},
          IntervalMemberQ[CommonInterval,Interval[{lower,upper}]]
        ]
   ]
```

It should be stated that functions *IntersectIntervals*, *ToInterval* and *Intervals* are also part of the implementation of the math-subsystem. *Interval* is one of Mathematica's built-in functions. The purpose of those functions is expressed through their names, the implementational details are of minor importance and are omitted due to space considerations.

Chapter 8
TRANSFORMATION OF DISCRETE LOOPS INTO STANDARD ADA

Two prerequisites have to be met under all circumstances when transforming discrete loops into Ada:

1. For every discrete loop that has to be transformed the semantics given in Section 1.1 have to be preserved.

2. The enclosing program's semantics must not be changed.

The overall approach is to replace every discrete loop by a standard Ada loop construct the body of which is extended to behave in a 'discrete' way. The representation of a discrete loop in Ada95 comprises two related parts:

- Declarations of entities that are needed by the extensions in the loop body. In order to meet prerequisite (2) stated above, everything is declared locally by means of Ada's *block_statement*. In this way these declarations are not visible from outside of the block. However, they might hide other entities with the same name from direct visibility ([Ada95](8.3(5)) within the block (e.g.: in the loop body). For that reason a name-generation algorithm has been implemented that checks for possible conflicts with existing names (confer Section 9.1).

- Standard Ada loop construct. The body of this loop contains the discrete loop's body enclosed by extensions that ensure 'discrete' behavior of the loop.

The next sections illustrate how the various kinds of discrete loops are transformed into Ada95.

8.1 Transformation of Monotonical Discrete Loops into Standard Ada

8.1.1 One-Dimensional Discrete Loops

The following discrete loop has been taken from the Heapsort algorithm given in Figure 3, page 5.

Heapsort: Source

```
discrete H := K in 1 .. N/2 new H := 2*H | 2*H+1 loop
    J := 2*H;
    if J < N and then Arr(J) < Arr(J+1) then
        J := J+1;
    end if;
    if V >= Arr(J) then
        Arr(H) := V;
        exit;
    end if;
    Arr(H) := Arr(J);
    Arr(J) := V;
    H := J;
end loop;
```

The following code represents the transformation of the loop.

Heapsort: Transformation

```
declare
    type PSV_Type_1 is record
        out_of_range : Boolean := false;
        value : integer;
    end record;
    PSV1_1 : PSV_Type_1;
    PSV1_2 : PSV_Type_1;
    h : integer := k;
    Range_UB_1 : constant integer := n / 2;
begin
    while h in 1 .. Range_UB_1 loop
        if not PSV1_1.Out_Of_Range then
            begin
                PSV1_1.Value := 2 * h;
                if PSV1_1.Value <= h then
                    raise WoopDefs.MONOTONIC_ERROR;
                end if;
            exception
                when CONSTRAINT_ERROR =>
                        PSV1_1.Out_Of_Range := True;
                when others =>
                        raise;
            end;
        end if;
        if not PSV1_2.Out_Of_Range then
            begin
                PSV1_2.Value := 2 * H + 1;
```

```
28                  if PSV1_2.Value <= h then
29                      raise WoopDefs.MONOTONIC_ERROR;
30                  end if;
31              exception
32                  when CONSTRAINT_ERROR =>
33                         PSV1_2.Out_Of_Range := True;
34                  when others =>
35                         raise;
36              end;
37          end if;

38  --  Sequence of Statements:
39          j := 2 * h;
40          if j < n and then arr (j) < arr (j + 1) then
41              j := j + 1;
42          end if;
43          if v >= arr (j) then
44              arr (h) := v;
45              exit;
46          end if;
47          arr (h) := arr (j);
48          arr (j) := v;
49          h := j;

50          if (PSV1_1.Out_Of_Range or else PSV1_1.Value / = h)
51            and then (PSV1_2.Out_Of_Range or else PSV1_2.Value / = h)
52          then
53              raise WoopDefs.SUCCESSOR_ERROR;
54          end if;
55      end loop;
56  end;
```

The discrete loop has been replaced by a while loop with the condition

$$identifier\ in\ discrete\_subtype\_definition$$

The body of the discrete loop has not been touched during transformation (lines 39 - 49). The code before and afterwards corresponds to that 'extra work' it takes to make a while-loop behave in a *'discrete'* way. It consists of three steps:

1. Declare the types and objects needed.

2. Compute the possible successive values (done on a per iteration basis).

3. Check whether the new value of the loop variable is contained in the list of possible successive values and whether it is still within its range (done on a per iteration basis).

The following paragraphs elaborate on the topics given above.

Step 1: Declaration of Types and Objects.

Type *PSV_Type_1* is declared to hold possible successive values of the loop variable (lines 2 - 5). It consists of two components:

- *out_of_range*: A Boolean flag that is set if an iteration function computes a value that is out of the range of the type of the loop variable.
- *value*: Object used to hold a possible successive value while the loop body is executed. It is of the type of the loop variable.

For every iteration-function of a one-dimensional loop we declare an object of type PSV_Type_1 (lines 6 and 7). Those objects are indexed as PSV1_i where i stands for the i^{th} iteration function of the discrete_subtype_definition.

The loop variable is declared and initialized with the initial_value at line 8. Its type has been derived by the algorithm in Figure 15.

Line 9 declares a so-called *range_constant* for the upper bound of the discrete_subtype_definition. Range_constants are necessary in case of non-static bounds to ensure evaluate-once semantics of the discrete_subtype_definition.

Note that for any declaration except the loop variable that we introduce it has to be ensured that it does not hide some other entity declared in an enclosing scope of the program.

Step 2: Computation of the Possible Successive Values.

A possible successive value has to be computed for each iteration function *before* each iteration of the loop (conf. lines 12 - 24 and lines 25 - 37). The reason for the exception handler (lines 18 - 23 and 31 - 36) is that we do not want an overflowing iteration function to alter the flow of execution by means of a constraint_error exception. If such an overflow occurs, we catch the corresponding exception and exclude the iteration function from further evaluation by setting its out_of_range flag. We also have to ensure that the values that an iteration function computes are monotonically increasing (or decreasing if keyword 'reverse' is present).

Step 3: Consistency Checks.

An iteration of a discrete loop cannot be called complete until we have ensured that the loop variable has been assigned a value in the loop body that conforms to the iteration functions. This of course has to be done after the loop body has been executed (conf. lines 50 & 51). If no iteration function predicted the value of the loop variable, or if all iteration functions are already out of range, the exception successor_error is raised.

8.1.2 Multi-Dimensional Discrete Loops

The following discrete loop has been taken from Euclid's algorithm given in Figure 6, page 11.

Euclid's Algorithm: Source

```
discrete (M,N) := (A,B) in reverse (1..A, 1..B)
 new (M,N) := (N, M mod N) loop
   H := M;
   M := N;
   N := H mod N;
end loop;
```

Euclid's Algorithm: Transformation

```
declare
   type PSV_Type_1 is record
      out_of_range : Boolean := false;
      value : integer;
   end record;
   PSV1_1 : PSV_Type_1;
   PSV2_1 : PSV_Type_1;
   m : integer := a;
   n : integer := b;
   Range_UB_1 : constant integer := a;
   Range_UB_2 : constant integer := b;
begin
   while (m in 1 .. Range_UB_1) and then (n in 1 .. Range_UB_2) loop
      if not PSV1_1.Out_Of_Range then
         begin
            PSV1_1.Value := n;
            if PSV1_1.Value > m then
               raise WoopDefs.MONOTONIC_ERROR;
            end if;
         exception
            when CONSTRAINT_ERROR =>
                  PSV1_1.Out_Of_Range := True;
            when others =>
                  raise;
            end;
        end if;
        if not PSV2_1.Out_Of_Range then
            begin
                PSV2_1.Value := m mod n;
                if PSV2_1.Value > n then
                    raise WoopDefs.MONOTONIC_ERROR;
                end if;
            exception
                when CONSTRAINT_ERROR =>
                        PSV2_1.Out_Of_Range := True;
                  when others =>
                        raise;
            end;
        end if;
        if not (PSV1_1.Out_Of_Range or else PSV2_1.Out_Of_Range)
          and then (PSV1_1.Value = m and then PSV2_1.Value = n)
        then
            raise WoopDefs.MONOTONIC_ERROR;
        end if;

--  Sequence of Statements:
        h := m;
        m := n;
        n := h mod n;

        if (PSV1_1.Out_Of_Range or else PSV1_1.Value / = m) or else (
          PSV2_1.Out_Of_Range or else PSV2_1.Value / = n)
        then
            raise WoopDefs.SUCCESSOR_ERROR;
        end if;
    end loop;
end;
```

The discrete loop has been replaced by a while loop with the condition

$$\forall i_{1 \leq i \leq Dim(L)}\ identifier(L,\ i)\ in\ discrete\_subtype\_definition(L,\ i)$$

Transformation of multi-dimensional discrete loops works pretty much the same as transformation of their one-dimensional counterparts. Differences are due to the fact that we are now dealing with *aggregate variables.*

Step 1: Declaration of Types and Objects.

Type PSV_Type_i is used to hold the successive values for parameter a_i of the loop variable. For every distinct type among the parameters one PSV_Type_i has to be declared. Since in this example both dimensions of the loop variable are of the same type, we only need to declare one of them.

For every iteration function of every dimension of the loop one object of the corresponding PSV_Type has to be declared. Those objects are indexed as PSVi_j where i stands for the i^{th} dimension and j for the j^{th} iteration function of that dimension. Lines 8 & 9 declare the parameters of the loop variable. Dimension one as well as dimension two need a range constant for the upper bound of their discrete_subtype_definitions (lines 10 & 11). Range_constants are indexed through the dimension they belong to.

Step 2: Computation of the Possible Successive Values.

Computing the possible successive values and checking monotony can be done in one step with one-dimensional monotonic discrete loops. Either the new value is greater (smaller with reverse loops) than the loop variable or it is not. Monotony in the case of multi-dimensional discrete loops however is a little trickier to check. Definition 1.2.6 requires only one parameter of a possible successive value (p_i) to be greater (smaller if reverse loop) than the corresponding parameter of the loop variable. The remaining parameters may keep their value. For this reason monotony can only be checked in two passes:

- Pass one computes the possible successive values on a per-dimension basis (lines 14 - 26 & lines 27-39 in this example). Monotony of a loop variable (p_i) cannot be ensured by examining only single parameters. Therefore the goal of the first pass can only be to ensure that no parameter's new value is smaller (greater if reverse loop) then the current value of that parameter. This corresponds to the first part of Definition 1.2.6:

  $$\forall i_{1 \leq i \leq Dim(L_{(a_i)})} \; a_i \, \underline{\scriptstyle\mathrm{M}} \, b_i$$

  If this check fails exception monotonic_error is raised.

- Pass two deals with Part two of Definition 1.2.6:

  $$\exists j : a_j \, {\scriptstyle\mathrm{M}} \, b_j$$

  Lines 40 - 44 are devoted to this check. If all parameters of the loop variable have kept their value, then the second part of Definition 1.2.6 is violated and exception monotonic_error is raised accordingly. Note that this example contains only one possible successive value for the loop variable. Checks for additional possible successive values can be '**or**ed' to lines 40 & 41.

Step 3: Consistency Checks.

It must be ensured that the loop variable has been assigned a value during execution of the loop body that conforms to the iteration_function_specification. If the iteration_function_specification failed to predict the new value of the loop variable exception successor_error is raised (lines 49 - 53).

8.2 Transformation of Remainder Function Loops into Standard Ada

Discrete loops with remainder functions can (but need not) contain the declaration of a loop variable. The following sections elaborate on the transformation of case1, case2, and case3 remainder function loops. Since the transformation of the loop variable's part does not change with the different kinds of remainder function loops, it is only explained in the first section. Thereafter only transformations related to the remainder loop variable are explained and code-samples have been stripped from code related to the loop variable to increase clarity and to reduce space. The complete (unstripped) code samples can be found in Appendix B.

8.2.1 Case1 Remainder Function Loops

The following example has been taken from Section 1.1.2 on page 6.

Binary Tree Search: Source

```
1   discrete Node_Pointer := Root
2      new Node_Pointer := Node_Pointer.Left | Node_Pointer.Right
3    with H := Height
4      new H = H - 1 loop

5      – loop body:
6      – Here the node pointed at by node_pointer is processed
7      – and node_pointer is either set to the left or right
8      – successor.
9      – The loop is completed if node_pointer = null.

10  end loop;
```

Binary Tree Search: Transformation

```
declare
   type PSV_Type_1 is record
        out_of_range : Boolean := false;
        value : tree_pointer;
   end record;
   PSV1_1 : PSV_Type_1;
   PSV1_2 : PSV_Type_1;
   node_ptr : tree_pointer := root;
   h : natural := height;
   Calculated_h : natural;
begin
   loop
      if h = 0 then
         raise WoopDefs.LOOP_ERROR;
      end if;
      Calculated_h := h - 1;
      if not PSV1_1.Out_Of_Range then
         begin
            PSV1_1.Value := node_ptr.left;
         exception
            when CONSTRAINT_ERROR =>
               PSV1_1.Out_Of_Range := True;
            when others =>
               raise;
         end;
      end if;
      if not PSV1_2.Out_Of_Range then
         begin
            PSV1_2.Value := node_ptr.right;
         exception
            when CONSTRAINT_ERROR =>
               PSV1_2.Out_Of_Range := True;
            when others =>
               raise;
         end;
      end if;

--  Sequence of statements (suppressed for space considerations).

      if Calculated_h >= h then
         raise WoopDefs.MONOTONIC_ERROR;
      else
         h := Calculated_h;
      end if;
      if (PSV1_1.Out_Of_Range or else PSV1_1.Value / = node_ptr)
```

```
44            and then (PSV1_2.Out_Of_Range or else PSV1_2.Value / =
45            node_ptr)
46         then
47            raise WoopDefs.SUCCESSOR_ERROR;
48         end if;
49      end loop;
50   end;
```

Transformations related to the Loop Variable

The code associated with the loop variable of a remainder function loop differs in two important aspects from a monotonical discrete loop variable's code:

- It cannot raise exception monotonic_error (lines 17 - 26 & 27 - 36).
- No discrete_subtype_definition can occur with remainder function loops. This is the reason why remainder function loops have to be transformed into an ordinary loop without an iteration_scheme as opposed to a while-loop in case of monotonical discrete loops (confer line 12).

Apart from the above differences we code on the analogy of monotonical discrete loops: Lines 2 - 5 declare a PSV_Type used to hold possible successive values. Again we might need several types with multi-dimensional loops. Thereafter we declare objects of the PSV_Type(s) (lines 6 - 7). The loop variable is declared and initialized in line 8. After execution of the sequence_of_statements we have to ensure that one of the possible successive values contains the new value of the loop variable. Otherwise exception successor_error is raised (lines 43 - 47).

Transformations related to the Remainder Loop Variable

Line 9 declares and initializes the remainder loop variable. Its type has been derived by the algorithm depicted in Figure 22. Another object of that type is needed for the value computed by the remainder function. For case1-remainder function loops this object is named *Calculated_<rem_identifier>* (line 10).

Within the loop body it has to be ensured that the value of the remainder loop variable has not reached zero. Otherwise exception loop_error is raised (lines 13 - 15). Thereafter we compute the value of the remainder_function (line 16). After the sequence of statements it is checked whether this value is smaller than the current value of the remainder loop variable. If this is the case the remainder loop variable is assigned the value of the remainder function. Otherwise exception monotonic_error is raised (lines 38 - 42).

8.2.2 Case2- & Case3 Remainder Function Loops

This section's example contains a template for the traversal of BB[α]-trees. Note that its transformation has been stripped from code that relates to the loop variable. A summary on BB[α]-trees as well as a non-stripped transformation of this example can be found in Appendix B.

Template for Operations on BB[α]-trees: Source

```
discrete Node_Pointer := Root
   new Node_Pointer := Node_Pointer.Left | Node_Pointer.Right
 with R := N --  N = number of leaves of the tree
   new R <= Floor ((1-Alpha)*R) and R >= Floor (Alpha*R)
loop
--  loop body (suppressed)
end loop;
```

Template for Operations on BB[α]-trees: Transformation

```
declare
   r : natural := n;
   Previous_UB : natural := r;
   Previous_LB : natural := r;
   Calculated_UB : natural;
   Calculated_LB : natural;
begin
   loop
      if r = 0 then
         raise WoopDefs.LOOP_ERROR;
      end if;
      Calculated_UB := floor ((1 - alpha) * r);
      Calculated_LB := floor (alpha * r);
--  loop body (suppressed)
      if Previous_UB = r then
         if not (Calculated_UB < R
          and then Calculated_LB <= Previous_LB
          and then Calculated_LB <= Calculated_UB)
         then
            raise WoopDefs.MONOTONIC_ERROR;
         else
            r := Calculated_UB;
         end if;
      else
         if not (Previous_UB > Calculated_UB
          and then Calculated_UB >= r
          and then Calculated_LB <= Previous_LB
28              and then Calculated_LB <= R)
29             then
30                raise WoopDefs.MONOTONIC_ERROR;
31             end if;
32          end if;
33          Previous_UB := r;
34       end loop;
35    end;
```

Line 2 declares and initializes the remainder loop variable. Its type has been derived by the algorithm depicted in Figure 22. The remainder_function's upper_bound_expression provides an upper bound for the value of the remainder loop variable. Case3-loops provide an additional lower bound in their lower_bound_expression. In order to ensure monotony of the remainder loop variable during subsequent iterations of the loop, it is necessary to compare the *current* value of each bound to its *previous* value. Naturally this takes two objects per bound. They are named *Previous_X* and *Calculated_X* where *'X'* stands for 'UB' (upper bound) or 'LB' (lower bound). Those objects are declared to be of the same type as the remainder loop variable (lines 3 - 6).

Within the loop body it has to be ensured that the value of the remainder loop variable has not reached zero. Otherwise exception loop_error is raised (lines 9 - 11). Thereafter (lines 12 & 13) the *current* values for the bounds are calculated.

Execution of the loop body (body suppressed due to space considerations) leaves us with two possibilities:

- *The value of the remainder loop variable has* not *been altered in the loop body:* Here the current value of the upper bound has to be smaller than the value of the remainder loop variable (line 16). Since the value of the remainder loop variable and the *previous* value of the upper bound are the same in this case, it is implicitly checked that the *current* value of the upper bound is smaller than the *previous* value of the upper bound. Additional checks are introduced with case3-remainder function loops:
  - The *current* value of the lower bound must not be greater than the previous value of the lower bound (line 17).
  - The interval [*current* value of lower bound, *current* value of upper bound] must contain at least one element (line 18).

  If one of this checks fails, exception monotonic_error is raised. Otherwise the remainder loop variable is assigned the *current* value of the upper bound (line 22).

- *The value of the remainder loop variable has been altered in the loop body:* Here it has to be checked explicitly that the *current* value of the upper bound is smaller than the *previous* value of the upper bound (line 25). The *current* value of the the loop variable has to be equal or smaller than the *current* value of the upper bound (line 26). This ensures that the remainder loop variable cannot do worse in the loop body than predicted by the upper bound. Additional checks are introduced with case3-remainder function loops:
  - The *current* value of the lower bound must not be greater than the *previous* value of the lower bound (line 27).
  - The interval [*current* value of lower bound, *current* value of remainder loop variable] must contain at least one element (line 28).

  If one of the above checks fails exception monotonic_error is raised (line 30).

Line 33 finally assigns the *current* value of the loop variable to the *previous* upper bound. Note that right before this point the property

$$current\ value\ of\ upper\ bound \geq current\ value\ of\ remainder\ loop\ variable$$

holds in any case. If the second branch of the if-statement (lines 24 - 32) is taken, even the property

$$current\ value\ of\ upper\ bound > current\ value\ of\ remainder\ loop\ variable$$

holds. If we would take the *current* value of the upper bound as the next iteration's *previous* value of the upper bound, a 'Byzantine' remainder loop variable could exploit the interval

$$[current\ value\ of\ upper\ bound, current\ value\ of\ remainder\ loop\ variable]$$

to go backwards without notice.

The consequence of this assignment statement is that if we talk about the *previous* value of the upper bound, we actually mean the *previous* value of the remainder loop variable. Another consequence is that by that means we save one object that would be used to store the *previous* value of the remainder loop variable otherwise. The initialization of the bounds with the initial_value of the remainder loop variable (lines 3 & 4) conforms to this scheme.

Chapter 9
CODE GENERATION

It is one of GNAT's built-in abilities to dump the source code from the generated abstract syntax tree. Despite the fact that this feature was only meant for debugging purposes, we climb the band wagon and exploit it for our needs. The mode of operation is as follows: we let GNAT do all the work until it encounters a discrete loop in the tree. This is the point were we take over in order to generate what has been explained in Section 8. For the loop body and at the end of the loop we return control to GNAT.

Package Sprint contains the code responsible for tree dumps. Two procedures and one function are necessary to accommodate code-generation of discrete loops:

- *Generate_Name*: Used to generate distinct names that are not occupied by some other entity (see next section).
- *Check_Context_Clauses*: Called on encountering a node N_Compilation_Unit. Checks for the necessity of a context clause for package WoopDefs (see next section but one).
- *Write_Discrete_Loop*: Outputs the source code for discrete loops.

Since all three functions depend on entities declared in the body of package Sprint, they have to be declared as subunits instead of procedures within a child package of Sprint.

9.1 The Name Generation Algorithm

It has been stated in Section 8 that the entities that are generated by WPP must not hide entities of the same name. It is the purpose of the name generation algorithm to find a name that is not already occupied by some other entity.

My first guess was to take the desired name and append the current system time to it if the name is already visible. Trying this for about $N + 1$ times should result in a distinct name (provided that any Ada program that we can think of contains only N named entities). Since the other WOOP people did not like this method, I resorted to something more common: If the desired name is already visible, suffix "_wpp" is appended to it. If an entity of that name is also visible, an index gets appended. This index is replaced by index'SUCC until a distinct name is finally found.

Note that the names used in Section 8 are 'desired' names throughout.

9.2 Package WoopDefs

The code that we generate depends heavily on three exceptions: monotonic_error, successor_error and loop_error. They are declared in package WoopDefs. To have them at hand, this package is added to a compilation unit's context_clause if it is not already visible.

woopdefs.ads

```
package WoopDefs is

   MONOTONIC_ERROR : exception;
   SUCCESSOR_ERROR : exception;

   LOOP_ERROR : exception;

end WoopDefs;
```

sprint-check_context_clauses.adb

```
procedure Check_Context_Clauses (Node : Node_Id) is
begin
   if Needs_WoopDefs_Context_Clause (Node) then
      Write_Indent;
      Write_Str_With_Col_Check ("with WoopDefs;");
      if Woop.Woop_Debug then
         Write_Indent;
         Write_Str ("-- BBdb:  wpp includes package WoopDefs!");
      end if;
   end if;
end Check_Context_Clauses;
```

The flag Needs_WoopDefs_Context_Clause gets set during semantic analysis on encountering a discrete loop provided that this package is not already visible.

9.3 Write_Discrete_Loop

This is really a big one. It does the code generation for all kinds of discrete loops. It is called from within package Sprint which contains a huge case-statement over all kinds of nodes that GNAT supports. Fiddling with the case_statement_alternative for node N_Loop_Statement (node-kinds are sorted in alphabetical order within Sprint) makes it call procedure Write_Discrete_Loop in case of a discrete loop:

sprint.adb

```
...

when N_Loop_Statement =>
   Write_Indent;

--  BB: If 'Node' contains a discrete loop, we use our own
--  output-function:
   if Present (Iteration_Scheme (Node)) and then
    Is_Discrete_Loop (Iteration_Scheme (Node)) then
      Write_Discrete_Loop (Node);
   else
--  Print common Ada95 loop statement.
   end if;

...
```

The structure of procedure Write_Discrete_Loop is as follows:

1. Declarations of pointers to strings are used to hold the names of the entities that are generated by WPP (lines 5 - 9).
   - *Previous_*, Calculated_**: Entities used to hold the bounds of a remainder function loop (see Section 8.2).
   - *Implicit Type*: Used for constrained discrete_subtype_definitions e.g.: `discrete ... in integer range ...`. In this case we have to create a subtype with the specified constraint. The reason for this extra work is that Ada does not allow constraint subtype_indications in iteration_schemes (like `while s in integer range ...`).
   - *Range_UB, Range_LB*: Constants that help enforce single evaluation of the 'while'-iteration_scheme.
2. Procedures devoted to the generation of various parts of discrete loops. The responsibilities of the different procedures are spelled out here. Line numbers

refer to the transformation of the discrete loop of the BB[α]-tree example (Appendix B). If the line numbers of some procedure are contained in those of another one, this means that the procedure is called by the 'enclosing' one.

- *Write_Discloop_Declarations*: Depending on the 'Loop_Type', declarations related to the loop variable are generated for monotonical loops or loops with remainder functions (lines 2-8).
- *Write_Remainder_Declarations*: Generates additional declarations for loops with remainder functions (lines 9 - 13).
- *Write_DiscLoop_Step1_And_Step2*: Generates code for the calculation of the possible successive values and checks for monotonic_error in case of a monotonical discrete loop (lines 21-40).
- *Write_Monotonic_DiscLoop_Body*: Generates the body of a monotonical discrete loop. Returns control to GNAT for the generation of the sequence_of_statements.
- *Write_Remainder_DiscLoop_Body*: Generates the body of a remainder function loop (lines 15 - 63). Returns control to GNAT for the generation of the sequence_of_statements.
- *Write_DiscLoop_Step3*: Generate checks for exception successor_error (lines 58-62).

3. Body of procedure write_discrete_loop.

sprint-write_discrete_loop.adb

```
1    separate (Sprint)

2    procedure Write_Discrete_Loop (N : Node_Id) is
3        I : constant Node_Id := Iteration_Scheme (N);
4        L : constant Node_Id := Loop_Parameter_Specification (I);

5        Previous_UB : String_Ptr;
6        Previous_LB : String_Ptr;
7        Calculated_UB : String_Ptr;
8        Calculated_LB : String_Ptr;

9        Implicit_Type : array (1 .. Dimensions (L)) of String_Ptr;

10       Range_UB : array (1 .. Dimensions (L)) of String_Ptr;
```

```
    Range_LB : array (1 .. Dimensions (L)) of String_Ptr;

    procedure Write_DiscLoop_Declarations (LT : Loop_Type);

    procedure Write_DiscLoop_Remainder_Declarations;

    procedure Write_DiscLoop_Step1_And_Step2 (LT : Loop_Type);

    procedure Write_Monotonic_DiscLoop_Body;

    procedure Write_Remainder_DiscLoop_Body;

    procedure Write_DiscLoop_Step3;

--  Procedure bodies of above declarations
--  withheld due to space considerations!

begin --  Write_Discrete_Loop
    if Is_Monotonical_Loop (L) then
        Write_Indent_Str ("declare");
        Write_DiscLoop_Declarations (M_Loop);
        Write_Indent_Str ("begin");
        Write_Monotonic_DiscLoop_Body;
        Write_Indent_Str ("end;");
    else
        Write_Indent_Str ("declare");
        if Has_Loop_Variable (L) then
            Write_DiscLoop_Declarations (R_Loop);
        end if;
        Write_DiscLoop_Remainder_Declarations;
        Write_Indent_Str ("begin");
        Write_Remainder_DiscLoop_Body;
        Write_Indent_Str ("end;");
    end if;
end Write_Discrete_Loop;
```

The implementation of the body of procedure Write_Discrete_Loop is straightforward. It contains two branches, one for monotonical discrete loops, the other for remainder function loops. Both start with the output of keyword 'declare' in order to start the block_statement needed for declarations. Depending on the type of loop the declarations are generated. Keyword 'begin' marks the end of the declarative_part and the beginning of the handled_sequence_of_statements of the block_statement. The loop body is generated with a call to Write_Monotonic-_DiscLoop_Body or Write_Remainder_DiscLoop_Body. Keyword 'end' followed by a *semi-colon* ends code-generation of a discrete loop.

Chapter 10

CONCLUSIONS AND FURTHER WORK

The development of the WOOP preprocessor made it possible to use the up to this point solely theoretical concept of discrete loops in a real-world programming environment. Although in principle there was no cause to favor any particular programming language, it was the reliability and maintainability of *Ada95* that finally lead to the decision to build a preprocessor for that language. Another important decision was to implement WPP by extending the source code of GNAT, the Gnu Ada Translator. Although it took some time to become familiar with this huge piece of software, it payed off in any respect. Since GNAT itself is written entirely in Ada, its overall structure can be regarded as very modular. Coupling between modules is really loose and one can apply modifications locally and need not know all the details of the program as a whole.

Upper bounds for the number of iterations of discrete loops are calculated by means of Mathematica, a commercial computer algebra package. Parts of this preprocessor are therefore written in Mathematica's internal programming language.

In order to call Mathematica's kernel from within WPP, an Ada to Mathematica binding has been implemented.

Starting with the first infant versions of WPP numerous examples from various books on algorithms (e.g. [SE88]) have been successfully programmed with discrete loops.

The implementation of WPP also helped the definition of multi-dimensional discrete loops. Code generation provided insight on run-time checks and their possible avoidance.

Further Work

There is much room for further work with WPP, ideas for improvements on various parts of the system are given below.

- The math-subsystem should be extended to support the generation of upper bounds for symbolic values.

- Symbolic analysis should be performed on the loop body in order to obtain more accurate bounds for multi-dimensional monotonical discrete loops.
- The math-subsystem should be extended to support enumeration types and modular types.
- Run-time checks for exception successor_error and loop_error could be avoided by means of symbolic analysis of the loop body.
- Upper bounds for multi-dimensional discrete loops are computed on a per-dimension basis. Due to possible mutual dependencies between different dimension's loop parameters in the iteration_function_specification this should be done for all dimensions at once.

Appendix A
SYNTAX SUMMARY

This Annex summarizes the complete syntax of discrete loops. A description of the notation used can be found in [Ada95](1.1.4).

```
loop_statement ::=
    [loop_simple_name:]
        [iteration_scheme] **loop**
            sequence_of_statements
        **end loop** [loop_simple_name];

iteration_scheme ::= **while** condition
    | **for** for_loop_parameter_specification
    | **discrete** discrete_loop_parameter_specification

for_loop_parameter_specification ::=
    identifier **in** [**reverse**] discrete_subtype_definition

discrete_loop_parameter_specification ::=
    monotonical_discrete_loop_parameter_specification
    | discrete_loop_with_remainder_function_parameter_specification
    | multi_dimensional_monotonical_discrete_loop_parameter_specification
    |multi_dimensional_discrete_loop_with_remainder_function_parameter_spec

monotonical_discrete_loop_paramter_specification ::=
      identifier := initial_value **in** [**reverse**] discrete_subtype_definition
          **new** identifier := list_of_iteration_functions

discrete_loop_with_remainder_function_parameter_specification ::=
    [identifier := initial_value
        **new** identifier := list_of_iteration_functions]
        **with** rem_identifier := initial_value **new** remainder_function
```

remainder_function ::=
rem_identifier = expression |
rem_identifier <= upper_bound_expression
[**and** rem_identifier >= lower_bound_expression]

multi_dimensional_monotonical_discrete_loop_parameter_specification ::=
identifier_aggregate **:=** initial_value_aggregate **in** [**reverse**] range_aggregate
new identifier_aggregate **:=** iteration_function_specification

identifier_aggregate ::= **(** identifier { **,** identifier } **)**

initial_value_aggregate ::= **(** initial_value { **,** initial_value } **)**

range_aggregate ::= **(** [**reverse**] discrete_subtype_definition
{**,** [**reverse**] discrete_subtype_definition} **)**

iteration_function_specification ::= iteration_function_aggregate
{| iteration_function_aggregate}

iteration_function_aggregate ::= **(**list_of_iteration_functions
{**,**list_of_iteration_functions}**)**

multi_dimensional_discrete_loop_with_remainder_function_parameter_spec ::=
[identifier_aggregate **:=** initial_value_aggregate
new identifier_aggregate **:=** iteration_function_specification]
with rem_identifier **:=** initial_value **new** remainder_function

remainder_function ::=
rem_identifier = expression |
rem_identifier <= upper_bound_expression
[**and** rem_identifier >= lower_bound_expression]

identifier_aggregate ::= (identifier { , identifier })

initial_value_aggregate ::= (initial_value { , initial_value })

iteration_function_specification ::= iteration_function_aggregate
    {| iteration_function_aggregate}

iteration_function_aggregate ::= (list_of_iteration_functions
    {,list_of_iteration_functions})

list_of_iteration_functions ::=
    iteration_function { | iteration_function}

iteration_function ::= expression

Appendix B

EXAMPLES OF DISCRETE LOOPS AND THEIR TRANSFORMATIONS

B.1 Weight-Balanced Trees

Weight-balanced trees have been introduced in [NR73]. They are treated in detail in [ME84], the following summary is taken from [Bli94].

1. Let T be a binary tree with left subtree T_l and right subtree T_r. Then

   $$\rho(T) = |T_l|/|T| = 1 - |T_r|/|T|$$

   is called the root balance of T. Here $|T|$ denotes the number of leaves of tree T.

2. Tree T is of bounded balance α if for every subtree T' of T:

   $$\alpha \leq \rho(T') \leq 1 - \alpha$$

3. BB[α] is the set of all trees of bounded balance α.

If the parameter α satisfies $1/4 < \alpha \leq 1 - \sqrt{2}/2$, the operations *Access, Insert, Delete, Min,* and *DeleteMin* take time O(Log N) in BB[α] trees. Here N is the number of leaves in the BB[α]-tree. Some of the above operations can move the root balance of some nodes on the path of search outside the permissible range $[\alpha, 1-\alpha]$. This can be "repaired" by *single* and *double rotations* (for details see [ME84]).

BB[α]-trees are binary trees with bounded height. In fact it is proved in [ME84] that

$$height(T) \leq \frac{logN - 1}{-log(1-\alpha)} + 1,$$

where N is the number of leaves in the BB[α]-tree T.

The following template utilizes a discrete loop for the traversal of BB[α]-trees. The remainder function operates on the number of leaves of the tree.

Template for Operations on BB[α]-trees: Source

```
discrete Node_Pointer := Root
    new Node_Pointer := Node_Pointer.Left | Node_Pointer.Right
 with R := N --  N = number of leaves of the tree
    new R <= Floor ((1-Alpha)*R) and R >= Floor (Alpha*R)
loop
--  loop body (suppressed)
end loop;
```

Template for Operations on BB[α]-trees: Transformation

```
declare
    type PSV_Type_1 is record
          out_of_range : Boolean := false;
          value : tree_pointer;
    end record;
    PSV1_1 : PSV_Type_1;
    PSV1_2 : PSV_Type_1;
    node_pointer : tree_pointer := root;
    r : natural := n;
    Previous_UB : natural := r;
    Previous_LB : natural := r;
    Calculated_UB : natural;
    Calculated_LB : natural;
begin
    loop
        if r = 0 then
            raise WoopDefs.LOOP_ERROR;
        end if;
        Calculated_UB := floor ((1 - alpha) * r);
        Calculated_LB := floor (alpha * r);
        if not PSV1_1.Out_Of_Range then
            begin
                PSV1_1.Value := node_pointer.left;
            exception
                when CONSTRAINT_ERROR =>
                    PSV1_1.Out_Of_Range := True;
                when others =>
                    raise;
            end;
        end if;
        if not PSV1_2.Out_Of_Range then
            begin
                PSV1_2.Value := node_pointer.right;
            exception
```

```
                when CONSTRAINT_ERROR =>
                    PSV1_2.Out_Of_Range := True;
                when others =>
                    raise;
            end;
        end if;
--  loop body (suppressed)
        if Previous_UB = r then
            if not (Calculated_UB < r and then Calculated_LB <=
              Previous_LB and then Calculated_LB <= Calculated_UB) then
                raise WoopDefs.MONOTONIC_ERROR;
            else
                r := Calculated_UB;
            end if;
        else
            if not (Previous_UB > Calculated_UB
              and then Calculated_UB >= r
              and then Calculated_LB <= Previous_LB and then Calculated_LB <= R)
            then
                raise WoopDefs.MONOTONIC_ERROR;
            end if;
        end if;
        Previous_UB := r;
        if (PSV1_1.Out_Of_Range or else PSV1_1.Value / = node_pointer)
         and then (PSV1_2.Out_Of_Range or else PSV1_2.Value / = node_pointer)
        then
            raise WoopDefs.SUCCESSOR_ERROR;
        end if;
    end loop;
end;
```

B.2 Mergesort

The algorithm performs a bottom-up (non-recursive) merge sort. A detailed description of Mergesort can be found in [SE88].

Mergesort: Source

```
N : constant Integer := ??; --  Number of elements to be sorted.
subtype Index is Integer range 1 .. N;
type Gen_Sort_Array is array (Index range <>) of Integer;
subtype Sort_Array is Gen_Sort_Array (Index);
Target : Sort_Array;

procedure Merge_Sort (From, To : Index) is
    M : constant Integer := (From+To)/2 + 1;
    subtype Aux_Array is Gen_Sort_Array (M..To);
    Aux : Aux_Array;
begin
    if From = To then
        return;
    end if;
    Merge_Sort (From, M-1);
    Merge_Sort (M, To);
    Aux := Target (M .. To);
    discrete (P,Q,R) := (M-1, Aux'Last, To)
     in reverse (From-1 .. M-1, Aux'First .. Aux'Last, From .. To)
     new (P,Q,R) := (P-1, Q, R-1) | (P, Q-1, R-1)
    loop
        if P < From or else Target (P) < Aux (Q) then
            Target (R) := Aux (Q);
            Q := Q-1;
        else
            Target (R) := Target (P);
            P := P-1;
        end if;
        R := R-1;
    end loop;
end Merge_Sort;
```

Mergesort: Transformation

```
procedure merge_sort (from, to : index) is
    m : constant integer := (from + to) / 2 + 1;
    subtype aux_array is gen_sort_array (m .. to);
    aux : aux_array;
begin
    if from = to then return; end if;
    merge_sort (from, m - 1);
    merge_sort (m, to);
    aux := target (m .. to);
    declare
        type PSV_Type_1 is record
            out_of_range : Boolean := false;
            value : integer;
        end record;
        PSV1_1 : PSV_Type_1;
        PSV1_2 : PSV_Type_1;
        PSV2_1 : PSV_Type_1;
        PSV2_2 : PSV_Type_1;
        PSV3_1 : PSV_Type_1;
        PSV3_2 : PSV_Type_1;
        p : integer := m - 1;
        q : integer := aux'last;
        r : integer := to;
        Range_LB_1 : constant integer := from - 1;
        Range_UB_1 : constant integer := m - 1;
        Range_LB_2 : constant integer := aux'first;
        Range_UB_2 : constant integer := aux'last;
        Range_LB_3 : constant integer := from;
        Range_UB_3 : constant integer := to;
    begin
        while (p in Range_LB_1 .. Range_UB_1) and then (q in
         Range_LB_2 .. Range_UB_2) and then (r in Range_LB_3 ..
         Range_UB_3)
        loop
            if not PSV1_1.Out_Of_Range then
                begin
                    PSV1_1.Value := p - 1;
                    if PSV1_1.Value > p then
                        raise WoopDefs.MONOTONIC_ERROR;
                    end if;
                exception
                    when CONSTRAINT_ERROR =>
                        PSV1_1.Out_Of_Range := True;
                    when others =>
                        raise;
                end;
            end if;
            if not PSV1_2.Out_Of_Range then
                begin
                    PSV1_2.Value := p;
                    if PSV1_2.Value > p then
                        raise WoopDefs.MONOTONIC_ERROR;
                    end if;
                exception
                    when CONSTRAINT_ERROR =>
                        PSV1_2.Out_Of_Range := True;
                    when others =>
                        raise;
                end;
            end if;
            if not PSV2_1.Out_Of_Range then
                begin
                    PSV2_1.Value := q;
                    if PSV2_1.Value > q then
                        raise WoopDefs.MONOTONIC_ERROR;
                    end if;
                exception
                    when CONSTRAINT_ERROR =>
                        PSV2_1.Out_Of_Range := True;
            when others =>
                raise;
        end;
    end if;
    if not PSV2_2.Out_Of_Range then
        begin
            PSV2_2.Value := q - 1;
            if PSV2_2.Value > q then
                raise WoopDefs.MONOTONIC_ERROR;
            end if;
        exception
            when CONSTRAINT_ERROR =>
                PSV2_2.Out_Of_Range := True;
            when others =>
                raise;
        end;
    end if;
    if not PSV3_1.Out_Of_Range then
        begin
            PSV3_1.Value := r - 1;
            if PSV3_1.Value > r then
                raise WoopDefs.MONOTONIC_ERROR;
            end if;
        exception
            when CONSTRAINT_ERROR =>
                PSV3_1.Out_Of_Range := True;
            when others =>
                raise;
        end;
    end if;
     if not PSV3_2.Out_Of_Range then
         begin
             PSV3_2.Value := r - 1;
             if PSV3_2.Value > r then
                 raise WoopDefs.MONOTONIC_ERROR;
             end if;
         exception
             when CONSTRAINT_ERROR =>
                 PSV3_2.Out_Of_Range := True;
             when others =>
                 raise;
         end;
     end if;
     if ( not (PSV1_1.Out_Of_Range or else PSV2_1.Out_Of_Range
       or else PSV3_1.Out_Of_Range) and then (PSV1_1.Value = p
       and then PSV2_1.Value = q and then PSV3_1.Value = r))
       or else ( not (PSV1_2.Out_Of_Range or else PSV2_2
       .Out_Of_Range or else PSV3_2.Out_Of_Range) and then (
                PSV1_2.Value = p and then PSV2_2.Value = q and then
                PSV3_2.Value = r))
              then
                  raise WoopDefs.MONOTONIC_ERROR;
              end if;
--  Sequence of Statements:
              if p < from or else target (p) < aux (q) then
                  target (r) := aux (q);
                  q := q - 1;
              else
                  target (r) := target (p);
                  p := p - 1;
              end if;
              r := r - 1;
              if ((PSV1_1.Out_Of_Range or else PSV1_1.Value / = p)
                or else (PSV2_1.Out_Of_Range or else PSV2_1.Value / = q)
                or else (PSV3_1.Out_Of_Range or else PSV3_1.Value / = r))
                and then((PSV1_2.Out_Of_Range or else PSV1_2.Value / = p)
                or else (PSV2_2.Out_Of_Range or else PSV2_2.Value / = q)
                or else (PSV3_2.Out_Of_Range or else PSV3_2.Value / = r))
              then
                  raise WoopDefs.SUCCESSOR_ERROR;
              end if;
          end loop;
      end;
end merge_sort;
```

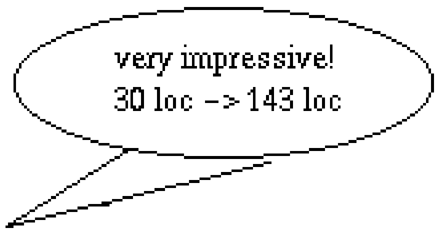

Figure 27: Project leader commenting code size

Appendix C
THE ADA TO MATHEMATICA BINDING

The MathLink functions built into Mathematica and included in the MathLink library implement MathLink communication over various transport systems. The Unix version supports communication via pipes or TCP. MathLink can be used to exchange data between Mathematica and external programs, between a Mathematica kernel and a front-end, or between two kernels. The MathLink library functions provide an interface between the elements of Mathematica expressions and external data types. MathLink as well as Mathematica are implemented in the C programming language. This binding is a thin binding which means that it provides wrap-around Ada-functions that closely resemble they C-ish counterparts. It strongly depends on package Interfaces.C in order to import functions from C and to convert data(types) between Ada and C.

C.1 Basic Pieces of MathLink Programming

- Package MathLink (the Ada to Mathematica Binding).
- MathLink library function for opening a link.
- MathLink library functions for putting data to or getting data from link.
- MathLink library functions for checking the type of incoming data elements.
- MathLink library function for closing a link.

Figure 28 contains a sample program that uses all of those elements.

C.1.1 Package MathLink

This package must be visible from all program units that use functions from the MathLink library. It contains all data types and functions that are necessary for MathLink programming. This package has been implemented according to the declarations given in the C file 'mathlink.h' that comes with any Mathematica distribution containing MathLink. It makes use of pragma Import in order to make C-entities accessible from Ada. The code of package MathLink is given at the end of this Appendix.

```
with MathLink; use MathLink;
with Gnat.IO; use Gnat.IO;
with System; use System;
with Interfaces.C; use Interfaces.C;
with Interfaces.C.Pointers;
with Interfaces.C.Strings; use Interfaces.C.Strings;

procedure Ada_Addinteger is

  package C renames Interfaces.C;

  Env : MLEnvironment;
  Lp : MLINK;
  Err : C.Int; I1, I2 : Integer; Res : aliased C.Int :=0;
  Argv : aliased Chars_Ptr_Array (0 .. 2) := (New_String ("ada_addinteger"),
                                              New_String ("-linkname"),
                                              New_String ("math -mathlink"));
begin
  Env := MLInitialize;
  Lp := MLOpen (Argv'Length, Argv'Access);
  Put_Line ("Demo for the Ada - Mathematica Binding:");
  Put ("First Integer:  "); Get (I1);
  Put ("Second Integer:  "); Get (I2); New_Line;
  Err := MLPutFunction (Lp, "Evaluate_Packet", 1);
  err := MLPutFunction (Lp, "Plus", 2);
  Err := MLPutInteger (Lp, C.Int (I1));
  Err := MLPutInteger (Lp, C.Int (I2));
  Err := MlEndPacket (Lp);
  Err := MLNextPacket (Lp);
  while (Err / = RETURNPKT) loop
     Err := MLNewPacket (Lp);
     Err := MLNextPacket (Lp);
  end loop;
  Err := MLGetInteger (Lp, Res'Access);
  Put ("Sum:  "); Put (Integer (Res)); New_Line;
  MLClose (Lp);
end Ada_Addinteger;
```

Figure 28: MathLink sample program (adding two integers)

C.1.2 Link Variable Declarations

Every time a program opens a link, the connection function will return an object of type MLINK. An MLINK variable is a pointer to the link data structure that is created to manage communication over a MathLink connection. Every time such a connection is accessed, the link has to be identified by passing the MLINK object as the first argument to a MathLink function.

C.1.3 Opening a Link

MLOpen is a general function for opening a MathLink connection. Its command line arguments may look familiar to C programmers. The sample program of Figure 28 shows how this can be resembled with Ada (lines 12-14). External programs that launch Mathematica can do so by passing the command-line arguments -linkname '*mathcommand*' to MLOpen, where *mathcommand* is the appropriate command string for starting a Mathematica kernel on the system (for Unix versions this is usually 'math -mathlink'.

C.1.4 Put and Get Functions

The MathLink library has a large number of functions for writing data to or reading it from a link. The following table shows a basic set of those functions.

MLPutInteger(link,inum)	MLGetInteger(link,inum'access)
MLPutReal(link,rnum)	MLGetReal(link,rnum'access)
MLPutString(link,char_array)	MLGetString(link,access chars_ptr)
MLPutSympol(link,char_array)	MLGetSymbol(link,access chars_ptr)
MLPutFunction(link,char_array,count)	MLGetFunction(link, access chars_ptr, count'access)

MLDisown functions should be used together with some of the MLGet functions in order to manage memory properly. After a program has finished looking at a character string returned by MLGetString, MLGetSymbol or MLGetFunction, it should call MLDisownString or MLDisownSymbol with the string as the second argument.

C.1.5 Moving from One Expression to the Next

When writing to a link, function MLEndPacket should be used any time after a complete expression has been put on the link. When reading from a link, MLNewPacket can be called in the middle of an expression to discard the remainder of that expression and go to the next one. MLNextPacket can be used at the beginning of each incoming expression to determine what kind of packet it is.

C.1.6 Closing a MathLink Connection

MLClose(link) closes a link. A program must close all links it has opened before terminating.

MathLink.ads

```ada
with System; use System;
with Interfaces.C; use Interfaces.C;
with Interfaces.C.Strings; use Interfaces.C.Strings;

package MathLink is

   pragma Linker_Options ("-static"); --  Due to Wolfram's missing Libelf!
   pragma Linker_Options ("-lMLelf");
   pragma Linker_Options ("-lm");
   pragma Linker_Options ("-DSTANDALONEMLINK");
--  Due to the commands specified in GNAT's Makefile the above pragmas don't
--  work from within WPP.

    package C renames Interfaces.C;

    type MLINK is new System.Address;
    type MLEnvironment is new System.Address;

    NO_MLINK : constant MLINK := MLINK (Null_Address);

--  Six types of expressions are supported by Mathematica:
    MLTKSYM : constant C.Char := 'Y'; -- symbol leaf node
    MLTKSTR : constant C.Char := 'S'; -- string leaf node
    MLTKINT : constant C.Char := 'I'; -- integer leaf node
    MLTKREAL : constant C.Char := 'R'; -- real leaf node
    MLTKFUNC : constant C.Char := 'F'; -- non-leaf node
    MLTKPCTEND : constant C.Char := ']'; -- at end of top level expression
    MLTKERROR : constant C.Char := C.Nul; -- (0) bad token

--  The following constants represent different packet type codes which
--  functions like MLNextPacket might return.
```

```
    ILLEGALPKT : constant := 0;
    CALLPKT : constant := 7;
    EVALUATEPKT : constant := 13;
    RETURNPKT : constant := 3;
    INPUTNAMEPKT : constant := 8;
    ENTERTEXTPKT : constant := 14;
    ENTEREXPRPKT : constant := 15;
    OUTPUTNAMEPKT : constant := 9;
    RETURNTEXTPKT : constant := 4;
    RETURNEXPRPKT : constant := 16;
    DISPLAYPKT : constant := 11;
    DISPLAYENDPKT : constant := 12;
    MESSAGEPKT : constant := 5;
    TEXTPKT : constant := 2;
    INPUTPKT : constant := 1;
    INPUTSTRPKT : constant := 21;
    MENUPKT : constant := 6;
    SYNTAXPKT : constant := 10;
    SUSPENDPKT : constant := 17;
    RESUMEPKT : constant := 18;
    BEGINDLGPKT : constant := 19;
    ENDDLGPKT : constant := 20;
    FIRSTUSERPKT : constant := 128;
    LASTUSERPKT : constant := 255;

--  MathLink errors:
--  When some problem is detected within MathLink, routines
--  will return a simple indication of failure and store
--  an error code internally. (For routines that have nothing
--  else useful to return, success is indicated by returning
--  non-zero and failure by returning 0.)  MLerror() returns
--  the current error code;  MLErrorMessage returns an English
--  language description of the error.
--  The error MLEDEAD is irrecoverable.
--  For the others, MLClearError() will reset the error code to MLEOK.

    MLEUNKNOWN : constant := -1; --  unknown error
    MLEOK : constant := 0; --  everything ok so far
    MLEDEAD : constant := 1; --  link died, unrecoverable error
    MLEGBAD : constant := 2; --  inconsistent data was read
    MLEGSEQ : constant := 3; --  MLGet? out of sequence
    MLEPBTK : constant := 4; --  MLPutNext() was passed a bad token
    MLEPSEQ : constant := 5; --  MLPut? out of sequence
    MLEPBIG : constant := 6; --  MLPutData given too much data
    MLEOVFL : constant := 7; --  machine integer overflow
    MLEMEM : constant := 8; --  out of memory
    MLEACCEPT : constant := 9; --  failure to accept socket connection
    MLECONNECT : constant := 10; --  a deferred connection is still
```

```
--                                          unconnected
   MLECLOSED : constant := 11; --  the other side closed the link, you
--                                          may still get undelivered data
   MLEPUTENDPACKET : constant := 21; --  unexpected call of MLEndPacket,
--                                          currently atoms aren't counted on
--                                          the way out so this error is raised
--                                          only when MLEndPacket is called in
--                                          the midst of an atom.
   MLENEXTPACKET : constant := 22; --  MLNextPacket called while the
--                                          current packet has unread data
   MLEUNKNOWNPACKET : constant := 23; --  MLNextPacket read in an unknown
--                                          packet head
   MLEGETENDPACKET : constant := 24; --  unexpected end-of-packet token
   MLEABORT : constant := 25; --  a put or get was aborted before
--                                          affecting the link
   MLEINIT : constant := 32; --  the MathLink environment was not
--                                          initialized
   MLEARGV : constant := 33; --  insufficient arguments to open link
   MLEPROTOCOL : constant := 34; --  protocol unavailable
   MLEMODE : constant := 35; --  mode unavailable
   MLELAUNCH : constant := 36; --  launch unsupported
   MLELAUNCHAGAIN : constant := 37; --  cannot launch the program again from
--                                          the same file
   MLELAUNCHSPACE : constant := 38; --  insufficient space to launch program
   MLENOPARENT : constant := 39; --  found no parent to connect to
   MLENAMETAKEN : constant := 40; --  the linkname was already in use
   MLENOLISTEN : constant := 41; --  the linkname was not found listening
   MLEBADNAME : constant := 42; --  the linkname was missing or not in
--                                          the proper form
   MLEBADHOST : constant := 43; --  the location was unreachable or not
--                                          in the proper form


   function MLClearError (Lp : MLINK) return C.Int;
   pragma Import (C, MLClearError, "MLClearError");

   procedure MLClose (Lp : MLINK);
   pragma Import (C, MLClose, "MLClose");

   function MLConnect (Lp : MLINK) return C.Int;
   pragma Import (C, MLConnect, "MLConnect");

   procedure MLDeinitialize (Env : MLEnvironment);
   pragma Import (C, MLDeinitialize, "MLDeinitialize");

   procedure MLDisownString (Lp : MLINK; Ptr : chars_ptr);
   pragma Import (C, MLDisownString, "MLDisownString");
```

```
procedure MLDisownSymbol (Lp : MLINK; Ptr : chars_ptr);
pragma Import (C, MLDisownSymbol, "MLDisownSymbol");

function MLEndPacket (Lp : MLINK) return C.Int;
pragma Import (C, MLEndPacket, "MLEndPacket");

function MLError (Lp : MLINK) return C.Int;
pragma Import (C, MLError, "MLError");

function MLErrorMessage (Lp : MLINK) return chars_ptr;
pragma Import (C, MLErrorMessage, "MLErrorMessage");

function MLFlush (Lp : MLINK) return C.Int;
pragma Import (C, MLFlush, "MLFlush");

function MLGetArgCount (Lp : MLINK; L : access C.Long) return C.Int;
pragma Import (C, MLGetArgCount, "MLGetArgCount");

function MLGetDouble (Lp : MLINK; F : access C.Double) return C.Int;
pragma Import (C, MLGetDouble, "MLGetDouble");

function MLGetInteger (Lp : MLINK; I : access C.Int) return C.Int;
pragma Import (C, MLGetInteger, "MLGetInteger");

function MLGetNext (Lp : MLINK) return C.Char;
pragma Import (C, MLGetNext, "MLGetNext");

function MLGetReal (Lp : MLINK; R : access C.Double) return C.Int;
pragma Import (C, MLGetReal, "MLGetReal");

function MLGetString (Lp : MLINK; Ptr : access chars_ptr) return C.Int;
pragma Import (C, MLGetString, "MLGetString");

function MLGetSymbol (Lp : MLINK; Ptr : access chars_ptr) return C.Int;
pragma Import (C, MLGetSymbol, "MLGetSymbol");

function MLInitialize (NULL_PTR : System.Address := System.Null_Address)
return MLEnvironment;
pragma Import (C, MLInitialize, "MLInitialize");

function MLNewPacket (Lp : MLINK) return C.Int;
pragma Import (C, MLNewPacket, "MLNewPacket");

function MLNextPacket (Lp : MLINK) return C.Int;
pragma Import (C, MLNextPacket, "MLNextPacket");

function MLOpen (Argc : C.Int; Argv : access chars_ptr_array) return MLINK;
pragma Import (C, MLOpen, "MLOpen");
```

```
   function MLPutFunction (Lp : MLINK; Str : char_array; Count : C.Long)
    return C.Int;
   pragma Import (C, MLPutFunction, "MLPutFunction");

   function MLPutInteger (Lp : MLINK; I : C.Int) return C.Int;
   pragma Import (C, MLPutInteger, "MLPutInteger");

   function MLPutString (LP : MLINK; Str : char_array) return C.Int;
   pragma Import (C, MLPutString, "MLPutString");

   function MLPutSymbol (Lp : MLINK; Str : char_array) return C.Int;
   pragma Import (C, MLPutSymbol, "MLPutSymbol");

   function MLReady (Lp : MLINK) return C.Int;
   pragma Import (C, MLReady, "MLReady");

end MathLink;
```

Appendix D
BONUS PROOFS

Proof I

Theorem 4.2 of [Bli94] gives an upper bound for an iteration function $f(x) = \lceil \alpha x + \beta \rceil,\ \alpha > 1,\ \beta \geq 0$. The corresponding lower bound is

$$\left\lfloor \log_\alpha \frac{N(\alpha-1)+\beta+1}{\alpha+\beta} + 1 \right\rfloor .$$

Proof. We clearly have

$$f(x) = \lceil \alpha x + \beta \rceil \leq \alpha x + \beta + 1$$

which leads to the iteration sequence

$$\begin{array}{rcl}
k_1 & = & 1 \\
k_2 & = & f(1) = \alpha + \beta + 1 \\
k_3 & = & f(\alpha+\beta+1) = \alpha^2 + \alpha(\beta+1) + \beta + 1 \\
\ldots & & \\
k_\nu & = & \alpha^{\nu-1} + \alpha^{\nu-2}(\beta+1) + \ldots + \beta + 1 = \\
 & = & \alpha^{\nu-1} + (\beta+1)(\alpha^{\nu-2} + \alpha^{\nu-3} + \ldots + \alpha + 1) \\
 & = & \alpha^{\nu-1} + (\beta+1)\frac{(\alpha^{\nu-2}+\alpha^{\nu-3}+\ldots+\alpha+1)(\alpha-1)}{(\alpha-1)} \\
 & = & \alpha^{\nu-1} + (\beta+1)\frac{\alpha^{\nu-1}-1}{\alpha-1}
\end{array}$$

Thus

$$\begin{array}{rcl}
k_\nu & \leq & \alpha^{\nu-1} + (\beta+1)\frac{\alpha^{\nu-1}-1}{\alpha-1} = \\
 & & = \alpha^{\nu-1}\frac{\alpha-1+\beta+1}{\alpha-1} - \frac{\beta+1}{\alpha-1} = \\
 & & = \alpha^{\nu-1}\frac{\alpha+\beta}{\alpha-1} - \frac{\beta+1}{\alpha-1}
\end{array}$$

To estimate the length of the iteration sequence k_ν we must have

$$\begin{array}{l}
\alpha^{\nu-1}\frac{\alpha+\beta}{\alpha-1} - \frac{\beta+1}{\alpha-1} > N \\
\alpha^{\nu-1} > \frac{N(\alpha-1)}{\alpha+\beta} + \frac{\beta+1}{\alpha+\beta} = \frac{N(\alpha-1)+\beta+1}{\alpha+\beta} \\
\alpha^\nu > \left(\frac{N(\alpha-1)+\beta+1}{\alpha+\beta}\right)\alpha
\end{array}$$

Taking logarithms we get

$$\nu > \log_\alpha \frac{N(\alpha-1)+\beta+1}{\alpha+\beta} + 1 \ \square$$

Proof II

Theorem 4.3 of [Bli94] gives an upper bound for an iteration function $f(x) = \lceil \alpha x^\gamma + \beta \rceil,\ \alpha > 1,\ \beta \geq 0,\ \gamma > 1$. The corresponding lower bound is

$$\left\lfloor \log_\gamma((\gamma-1)\log_{\alpha+\beta+1} N + 1) + 1 \right\rfloor .$$

Proof. We clearly have

$$f(x) = \lceil \alpha x^\gamma + \beta \rceil \leq \alpha x^\gamma + \beta + 1.$$

Thus $k_\nu \leq l_\nu$ where

$$\begin{aligned} &l_1 = 1 \\ &l_{\nu+1} = \alpha l_\nu^\gamma + \beta + 1 = \alpha l_\nu^\gamma \left(1 + \tfrac{\beta+1}{\alpha l_\nu^\gamma}\right) \end{aligned}$$

Taking logarithms and setting $m_\nu = \log\ l_\nu$ we obtain

$$\begin{aligned} m_1 &= 0 \\ m_{\nu+1} &= \gamma m_\nu + \log\ \alpha + \log\left(1 + \tfrac{\beta+1}{\alpha l_\nu^\gamma}\right) \end{aligned}$$

Since $l_\nu^\gamma > 0$ we have $m_\nu < n_\nu$ where

$$\begin{aligned} n_1 &= 0 \\ n_{\nu+1} &= \gamma n_\nu + \log\ \alpha + \log\left(\tfrac{\alpha+\beta+1}{\alpha}\right) \\ &= \gamma n_\nu + \log\left(\alpha \tfrac{\alpha+\beta+1}{\alpha}\right) \\ &= \gamma n_\nu + \log\ (\alpha+\beta+1) \end{aligned}$$

Setting $\log\ (\alpha+\beta+1) = d$ we get

$$\begin{aligned} n_1 &= 0 \\ n_2 &= d \\ n_3 &= \gamma d + d = d(\gamma+1) \\ n_4 &= \gamma^2 d + \gamma d + d = d(\gamma^2+\gamma+1) \\ \dots & \\ n_n &= d(\gamma^{n-2} + \ldots + \gamma + 1) = d\tfrac{\gamma^{n-1}-1}{\gamma-1} \end{aligned}$$

Therefore

$$n_\nu = \frac{\gamma^{\nu-1}-1}{\gamma-1} \log\ (\alpha+\beta+1)$$

To estimate the length of the iteration sequence we must have

$$k_\nu \ \leq \ (\alpha+\beta+1)^{\frac{\gamma^{\nu-1}-1}{\gamma-1}} > N$$

Taking logarithms we obtain

$$\begin{array}{lcl} \frac{\gamma^{\nu-1}-1}{\gamma-1} & > & \log_{\alpha+\beta+1} N \\ \gamma^{\nu-1} & > & (\gamma-1)\log_{\alpha+\beta+1} N + 1 \\ \gamma^{\nu} & > & ((\gamma-1)\log_{\alpha+\beta+1} N + 1)\gamma \end{array}$$

Taking logarithms once more we get

$$\nu > \log_{\gamma}((\gamma-1)\log_{\alpha+\beta+1} N + 1) + 1 \quad \square$$

Proof III

For an iteration function $f(x) = x + \beta, \beta > 0$ with initial value $k_1 = K$ the length of the corresponding loop sequence is bounded above by

$$\left\lfloor \frac{N-K}{\beta} + 1 \right\rfloor .$$

Proof:

$$k_1 = K$$

$$k_2 = f(K) = K + \beta$$

Thus

$$k_\nu = K + (\nu - 1)\beta$$

To estimate the length of the corresponding remainder loop sequence we must have

$$K + (\nu - 1)\beta > N$$

which is equivalent to

$$\nu > \frac{N-K}{\beta} + 1$$

which holds $\forall K \in \mathbf{Z}, \forall N \in \mathbf{Z}, N \geq K.\square$

It also works for monotonical discrete loops with keyword **reverse** and iteration-functions *f(x) = x - β, β > 0* .

Bibliography

[Ada95] ISO/IEC 8652, *Reference manual for the Ada programming language,* 1995.

[Bli94] J. Blieberger, *Discrete loops and worst case performance,* Computer Languages, 20 (1994), no. 3, 193-212.

[Bli95] J. Blieberger, *Project WOOP - Worst Case Performance of Object-Oriented Real-Time Programs,* A position paper on Project WOOP, (1995).

[BL94] J. Blieberger and R. Lieger, *Worst-case space and time complexity of recursive procedures,* Real-Time Systems 11, pp. 115 - 144, 1996.

[BL95] J. Blieberger and R. Lieger, *Real-time recursive procedures,* Proceedings of the 7th EUROMICRO Workshop on Real-Time Systems, Odense, Denmark, June 1995. IEEE Press.

[BB95] J. Blieberger and B. Burgstaller, *The Role of GNAT within Project WOOP,* Ada-Europe'95, Frankfurt, Germany, 1995

[BLB96] J. Blieberger, R. Lieger, and Bernd Burgstaller, *Augmenting Ada 95 with Additional Real-Time Features,* Ada-Europe'96, Montreux, Switzerland, June 1996

[GNAT] E. Schonberg and B. Banner, *The GNAT Project: A GNU-Ada9X Compiler,* In *Proceedings of Tri-Ada'94,* Baltimore, Maryland, 1994.

[GPL91] Free Software Foundation, *GNU General Public License,* 1991

[LIB94] Robert B.K. Dewar, *The GNAT Model of Compilation,* In *Proceedings of Tri-Ada'94,* Baltimore, Maryland, 1994.

[ME84] Kurt Mehlhorn, *Sorting and Searching,* Data Structures and Algorithms, vol. 1, Springer-Verlag, Berlin, 1984

[NR73] I. Nievergelt and E. Reingold, *Binary search trees of bounded balance,* SIAM Journal of Computing 2 (1973), no. 1, 33-43.

[PE] Roger Penrose, *The Emperor's New Mind,* Vintage

[SE88] Robert Sedgewick, *Algorithms,* Addison-Wesley, Reading, MA, second ed., 1988

[WO92] Stephen Wolfram, *Mathematica,* Addison-Wesley, second ed., 1992

[WR1] Wolfram Research, *MathLink Reference Guide,* http://mathsource.wri.com, second ed., 1993

[WR2] Wolfram Research, *A MathLink Tutorial,* http://mathsource.wri.com

[WR3] Roman E. Maeder, *Programming in Mathematica,* Reprint from The Mathematica Conference, June, 1992 Boston, MA, http://mathsource.wri.com

[WR4] Wolfram Research, *Major New Features in Mathematica Version 2.2,* Technical Report http://mathsource.wri.com

[WR5] Alexei V. Bocharov, *Solving equations symbolically with Mathematica,* Reprint from The Mathematica Conference, June, 1992 Boston, MA, http://mathsource.wri.com