The Role of GNAT within Project WOOP

Johann Blieberger
Bernd Burgstaller
Department of Automation
Technical University Vienna

November 13, 1995

Abstract

WOOP, as an acronym for *Worst Case Performance of Object-Oriented Programs,* is a research project at the Technical University of Vienna that is aimed at the determination of the timing-behavior of software for real-time systems[1]. Topics of research include the utilization of such high level language constructs as recursion and general loops so that they can be used safely without substantially reducing their computational capabilities. Beyond that, current research aims at merging the concept of full timing predictability with the paradigm of object-oriented programming in order to supply the developer with the ability to create fully predictable software at a very high level of abstraction. Due to its reliability and maintainability, Ada has been chosen as reference language for the entire project. A tool that incorporates the theoretical results is currently under development. Since its analytical tasks blend well with the analysis Gnat performs on programs, we decided to use Gnat in a rather uncommon way taking its source code to build our own tool on top of it.

1 Introduction

Back in 1993 one of the authors conducted an Ada programming course that the other author attended. Both were curious when they heard about the existence of Gnat, because their only way of programming in Ada by this time was to log into some ancient workstation running an even older compiler on it. It was Gnat's appearance that changed things radically. All of a sudden it was possible for everyone to run one's 'Personal Ada' on one's Personal Computer. Since then a long time has passed and 'Ada for the People' alias Gnat is close to validation, which is much more then most people had expected. There is strong evidence

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

that this year's programming course will be held for the second time now using Gnat, tasking and everything supported. The former Ada beginner has become a team member of project WOOP, and he is currently among those who build the Woop Pre-Processor (Wpp) on top of Gnat. By the time of its completion, this tool will be capable of providing timing constraints for object-oriented real-time programs containing discrete loops [Bli94] and even recursions [BL95].

By today's standards, this is a rather uncommon use of a compiler, but the public availability of the source code of Gnat could make this a common case, since we found it suitable for many different kinds of applications. This paper is meant as an example of how this can be accomplished. It should give an idea of the vast potential within Gnat that waits there to be explored and used.

The remaining sections of this paper are structured as follows: Section 2 gives an overview of the major research topics of project WOOP namely recursion, discrete loops and real-time objects. Section 3 elaborates on the concept of discrete loops. Section 4 lists the options one has to implement a tool like Wpp. It points out the reasons that finally led to the decision to build on top of Gnat. Section 5 delves into the implementational details of discrete loops. It contains examples of applications where our approach of (re)using the source code of Gnat could also be useful.

2 About WOOP

The following excerpt is taken from [Bli95].
Project Description. 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. We recognize two difficult tasks in estimating the timing behavior of a program:

- Determine the number of iterations of a certain loop
- Investigate the behavior of recursive procedures in time and space

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. Recursive procedures are forbidden in all well-known real-time languages, too. 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 & 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.

Recursive procedures and functions can also be used within real-time applications, as long as certain requirements are met (cf. [BL94]). We managed to prevent these requirements from being too restrictive such that not only simple mathematical recursions, like the Factorial Numbers and the Fibonacci Numbers, but also more complicated recursive procedures, like a recursive version of Mergesort and recursive tree traversal algorithms, can be handled.

Concerning *object oriented programming*, we require that each method of a real-time object/class has to be equipped by a **WCP**-*formula* which describes the **W**orst **C**ase **P**erformance behavior of the method. In general this formula does not only depend on the execution speed of the processor, but it also depends on *generic parameters* of the real-time object. One major advantage of WCP-formulas is that they greatly improve *reusability* of software.

Summing up, Project WOOP comprises several tasks:

1. Define syntax and semantics of WCP-formulas.

2. Develop language features to implement real-time programs. This includes *discrete loops* and adapting *recursive procedures* and *functions.*

3. Define syntax and semantics of a programming language, that contains all concepts elaborated in (1), (2) and (3).

4. Adapt an existing compiler (Gnat) for this programming language by adding all necessary features.

5. Design and implement a tool to estimate all simple statements of the programming language. Note that this task extends to analyzing the program as well as the corresponding object code.

3 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]

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 Figure1. 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**

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

Figure 1: A simple example of a discrete loop

would be reached. 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

```
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

than **N** would be reached. 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.

3.1 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_range

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

list_of_iteration_functions ::=
    iteration_function { | iteration_function}

iteration_function ::= expression
```

For a loop with a **discrete** iteration scheme, the loop parameter specification is the declaration of the *loop variable* with the given identifier. The initial value of the loop variable is given by initial_value. The optional keyword **reverse** defines the 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 range, and on the iteration functions.

Figure 4 shows an implementation of *Heapsort* using a discrete loop.

3.2 Discrete Loops with a Remainder Function

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, we have introduced the concept of the so-called *remainder loop variable.* 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[2]. The value of the remainder loop variable is computed during each iteration by the *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 is free to iterate over whatever he chooses except *limited* or *abstract* types, which is considered a major advantage over the traditional for-loop.

Figure 3 shows an example of a discrete loop with a remainder function. Its purpose is traversing a binary tree. The loop variable points to the current node, whereas the remainder loop variable describes the height of the remaining subtree.

Due to space considerations, the remaining sections of this paper will only deal with monotonical discrete loops.

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

```
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 3: Template for Binary Tree Traversal

4 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 that meant that we would have to build a compiler for a programming language that had been augmented by the concept of discrete loops. Two possibilities seemed to exist to get the job done:

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

To further ease the task of implementation we decided to build a precompiler at first. This precompiler would have to translate discrete loops to standard Ada. The resulting code could then be compiled by any Ada compiler. Thus our tool had 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 that this has to be done for the entire program, considering only those statements belonging to a discrete loop does by no means suffice!

In this situation it was the availability of Gnat that saved us a lot of work and enabled us to focus on the main problems. It prevented us from coding a complete front-end for the whole Ada programming language while providing every facility we 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. We found the source code of Gnat very well structured and fully documented. It left us with enough space to add where we needed and in the way we wanted.

5 What we actually did

In this section we discuss the implementation of Wpp within Gnat. We will focus on the important cornerstones in order to get the "big picture". Besides giving some insight on the way Gnat internally works, this should also convince the reader, that Gnat's source, despite its sheer size and complexity, can be understood and used creatively in any area that somehow depends on the analysis of Ada programs. Possible areas of application include software metrics, pretty printers, code transformers, and code optimizers.

5.1 The Abstract Syntax 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. Every single piece of the Ada programming language finds its reflection in the AST. It has got nodes for statements, expressions, declarations, tasks and so on. We chose to treat discrete loops as a special form of for-loops and augmented the node N_Loop_Parameter_Specification (a descendant of N_Iteration_Scheme which is a descendant of N_Loop_Statement) with nodes for the *initial_value* and the *list_of_iteration_functions*.

AST-access is governed by a high-level interface that checks for the validity of the required access (you cannot convince Gnat to provide you with a loop-body out of a case-statement, for example). Since, for obvious reasons, Gnat is very picky about this, we created our own high-level interface for WOOP-specific parts of the AST. A tool that checks for the consistency of such modifications is provided with Gnat.

5.2 Modifying Gnat's Scanner

We only had to add the keyword **discrete**, which was no big deal at all. Keywords are ordered alphabetically in the scanner, so we put it between **digits** and **do**. We also had to provide a string-representation for the new token. All in all it took not more than about ten lines of code!

5.3 Modifying the Parser

As it has been stated above, Gnat incorporates a recursive descent parser. It is divided into thirteen child units that properly reflect the structure of the corresponding chapters of the Ada Reference Manual. Loops belong to statements, therefore we were only concerned with the file representing chapter 5 of [RM95]. The first thing to do was to tell the parser to call our function when encountering a discrete loop:

```
-- Discrete_Statement

when Tok_Discrete =>
   Append_To (Statement_List, P_Discrete_Statement);
```

The code-fragment above is located in the part of the parser that deals with a *Sequence_of_Statements*. On encountering token **discrete**, it calls our function *P_Discrete_Statements* and appends the resulting tree to the current node. Further changes took place only within this function which freed us from the task of "messing around" too much with the parser. *P_Discrete_Statements* deals with the following topics:

1. Parse the identifier containing the loop variable.
2. Parse the expression containing the *initial_value*.
3. Parse the loop's *discrete_range* and check for the keyword **reverse**.
4. Parse the list of iteration functions.
5. Put together the results of (1) .. (4) in a (sub)tree and return it to Gnat.

5.4 Semantic Analysis of Discrete Loops

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 [DS95]. In the case of a loop statement, Gnat has to analyze the loop's *iteration_scheme* and the body of the loop. Since the body of a discrete loop does not semantically differ from any other loop, we leave the latter task to Gnat. This of course does not prevent us from performing special checks (compare 3.1) on the body after semantic analysis has been completed.

The sole purpose of the analysis of the *iteration_scheme* of a discrete loop is to determine the type of the loop variable and to verify that this type is a *discrete* type. Please note that such an *iteration_scheme* contains three entities that provide type-information:

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

The algorithm we chose for type resolution is depicted in pseudo-code in Figure 5.
It starts with calls to procedure *Analyze* to carry out the bottom-up pass for the loop's *init_id* and *iteration_scheme.* This leaves us with at least one possible type for both arguments. Strictly speaking, we get one type for non-overloaded entities and N ($N \in \mathcal{N}$, $N > 1$) 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* where 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 1 & 15) and *return_error* (lines 19 & 28): the first one complains put continues execution, whereas the latter one quits issuing an error-message. Note that our algorithm does in no way terminate before having passed line 18! 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.* Confer Figure 2 that this absolutely makes sense.

The remaining part of the algorithm (lines 20 - 37) distinguishes between two cases:

- We have found type 'Universal_Integer', which makes us consult the list of *iteration_functions* to find some specific integer-type (line 27).
- We have already found a specific type that has to fit the *iteration_functions.*

In order to avoid anonymous base types (we are generating source code, how should we refer to them) we have to compute their so-called *first named subtypes* (lines 21 & 33). What remains to be done is to resolve all three entities *(init_id, discrete_range & list_of_iteration_functions* with the type we have found. This corresponds to pass two (top-down) of Gnat's type resolution algorithm.

5.5 Estimating the Number of Iterations of a Discrete Loop

Simple cases, like the one depicted in Figure 4 are solved by Wpp itself. If things turn out to be more complex (e.g. complicated recurrence relations are involved), Wpp passes this task over to Mathematica. Since it was not until now that Mathematica has been ported to Linux, our interface to Math Link is still under implementation.

5.6 Transformation of Discrete Loops into Standard Ada

When transforming a discrete loop, two prerequisites have to be met under all circumstances:

1. For every discrete loop that has to be transformed, we have to preserve the semantics given in section 3.1.
2. The enclosing program's semantics must not be changed.

Figure 6 shows the resulting code that has been generated for the loop given in Figure 4. The body of the loop remains untouched (lines 27 - 36). The code before and afterwards corresponds to that 'extra work' it takes to make the loop behave in a *'discrete'* way. It can be regarded as three steps:

1. Declare the types and objects we need.
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 sections elaborate on the topics given above.

5.6.1 Step 1: Declaration of Types and Objects.

In order to meet prerequisite (1) stated above, we chose to declare things locally using Ada's *block_statement* (cf. line 1 - 11). Constant *UB* contains the upper bound for the number of iterations of that loop.

The loop variable is declared and initialized at line 10. Its type has been derived by the algorithm in Figure 5. The remaining declarations are devoted to holding the possible successive values while the loop body is executed. Note that for any declaration except the loop variable we have to ensure that it does not hide some other entity declared in an enclosing scope of the program.

The exceptions monotonic_error and successor_error are declared in a package called *DiscLoops*. Wpp checks whether those entities are visible and generates a *with_clause* if necessary.

5.6.2 Step 2: Computation of the Possible Successive Values.

This has to be done *'before'* each iteration (cf. lines 13 - 25)[3]. The reason for the exception handler (lines 19 - 24) 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. As long as at least one valid iteration function remains to predict the new value of the loop variable, the loop is correct, provided that this value is assigned to the loop variable in the next iteration.

Furthermore we have to ensure that the values each iteration function computes (lines 15-18) are monotonically increasing or decreasing. As stated in 3.1, this would otherwise raise the exception monotonic_error.

5.6.3 Step 3: Consistency Checks.

An iteration of a discrete loop can not 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 37 & 38). 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.

5.7 Code Generation

It is one of Gnat's built in abilities to dump the source code from the generated tree. Although this feature was only meant for debugging purposes, it perfectly suits our needs: 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 is depicted in Figure 6. For the loop body and at the end of the loop we return control to Gnat.

6 Conclusion

In this paper we have shown that the source code of Gnat can be (re)used in any area that somehow depends on the analysis of Ada programs. Possible areas of application include software metrics, pretty printers, code transformers, and code optimizers.

Concerning the implementation of Wpp, Gnat left us with nothing to wish for. We found its source code very well structured and fully documented. 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.

[3]The computation for the 2^{nd} iteration function has been left out for space considerations

Tools are provided that ensure the consistency of modifications of critical parts of the compiler. Various debugging facilities ease the task of testing.

References

[RM95] 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,* (to appear), 1994.

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

[DS95] R.Dewar and E.Schonberg, *The GNAT Project: A GNU-Ada9X Compiler,* cs.nyu.edu 1995.

DEPARTMENT OF AUTOMATION (183/1), TECHNICAL UNIVERSITY OF VIENNA, TREITL-STR. 1/4, A-1040 VIENNA

Email: **blieb@auto.tuwien.ac.at bburg@auto.tuwien.ac.at**

```
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 4: An implementation of Heapsort using a discrete loop

```
Analyze (init_id); Analyze (discrete_range);

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

if Is_Overloaded (init_id) and Is_Overloaded (discrete_range) then
    T := Compute_the_Intersection_of_both_Typesets;
elsif Is_Overloaded (init_id) and not Is_Overloaded (discrete_range) then
    if Type(discrete_range) $\in$ Typeset(init_id) then
        T := Type(discrete_range);
    end if;
elsif not Is_Overloaded (init_id) and Is_Overloaded (discrete_range) then
    if Type(init_id) $\in$ Typeset(discrete_range) then
        T := Type(init_id);
    end if;
else
    if Type(init_id) $\not\sim$ Type(discrete_range) 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 $\neq$ Universal_Integer then
    T := First_Named_Subtype (Base_Type (T));
    Resolve_Init_Id (T);
    Resolve_Discrete_Range (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_Range (T);
    Resolve_List_Of_Iteration_Functions (T);
end if;
```

Figure 5: Pseudo-code type resolution of an *iteration_scheme*.

```
declare
   UB : constant integer := DiscLoops.Numerics.Floor (
    DiscLoops.Numerics.Log (2, n/(2*k))+1.0);
   type PSV_Type is record
         out_of_range : Boolean := false;
         value : integer;
   end record;
   PSV1 : PSV_Type;
   PSV2 : PSV_Type;
   h : integer := k;
begin
   while h in 1 .. n / 2 loop
      if not PSV1.Out_Of_Range then
         begin
            PSV1.Value := 2 * h;
            if PSV1.Value ≤ h then
               raise DiscLoops.MONOTONIC_ERROR;
            end if;
         exception
            when CONSTRAINT_ERROR =>
                  PSV1.Out_Of_Range := True;
            when others =>
                  raise;
         end;
      end if;

      – Sequence of Statements:
      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;

      if (PSV1.Out_Of_Range or PSV1.Value ≠ h) and then (
       PSV2.Out_Of_Range or PSV2.Value ≠ h) then
         raise DiscLoops.SUCCESSOR_ERROR;
      end if;
   end loop;
end;
```

Figure 6: Discrete Loop of Figure 4 transformed into Ada