Methods for Handling Exceptions in Object-oriented Programming Languages

M.Sc. Thesis

by

Morten Mikael Christensen

Advisor: Søren Larsen

Department of Mathematics and Computer Science

Odense University, Denmark

January, 1995

2nd ed.

Contents

1. INTRODUCTION 7

1.1 THE BASIS FOR THIS THESIS 7
1.2 SO WHAT IS EXCEPTION HANDLING 7
1.3 ERROR-HANDLING WITHOUT EXCEPTION HANDLING 9

2. EXCEPTION HANDLING REQUIREMENTS 11

2.1 EXCEPTION HANDLING MODEL SPECIFICATION 11
2.2 EXCEPTION HANDLING DESIGN REQUIREMENTS 12
2.3 EXCEPTION HANDLING IMPLEMENTATION REQUIREMENTS 12
2.4 REQUIREMENTS DUE TO PROGRAMMING LANGUAGE 13

3. REQUIREMENTS OF PROGRAMMING LANGUAGE 15

3.1 OBJECT MODEL 15
3.2 LANGUAGE CONSTRUCTS 15
3.3 IDENTIFIERS AND SCOPES 16
3.4 MODULES 16

4. DESIGN OF EXCEPTION HANDLING MECHANISM 17

4.1 THE RAISE STATEMENT 17
4.2 THE EXCEPTION CONTROL CONSTRUCTIONS 17
4.3 EXCEPTIONS ARE OBJECTS 18
4.4 THE RAISES CONSTRUCTION 18

5. THE MEHL PROGRAMMING LANGUAGE 21

5.1 THE SYNTAX OF MEHL 21
5.2 RESERVED WORDS IN MEHL 23
5.3 EXAMPLES OF MEHL PROGRAMS 23
5.4 MEHL, THE DETAILS 24

6. METHODS FOR HANDLING EXCEPTIONS 27

6.1 FUNCTIONAL AREAS IN RUNTIME MECHANISM 27
6.2 COMPILER ISSUES 27
6.3 GENERAL CONSIDERATIONS 27
6.4 OVERVIEW OF ALTERNATIVE IMPLEMENTATION METHODS 28

7. TRANSFER OF CONTROL 31

7.1 TRANSFERRING CONTROL USING THE DYNAMIC REGISTRATION APPROACH 31
7.2 TRANSFERRING CONTROL USING THE STATIC TABLE APPROACH 34

8. OBJECT CLEANUP 39

8.1 REQUIREMENTS FOR DESTROYING LOCAL, GLOBAL AND SUB-OBJECTS 39
8.2 DESTROYING EXCEPTION OBJECTS 40
8.3 DETERMINING WHICH OBJECTS TO DESTROY WITH THE DYNAMIC REGISTRATION APPROACH 41
8.4 DETERMINING WHICH OBJECTS TO DESTROY WITH THE STATIC TABLE APPROACH 43
8.5 DESTROYING OBJECTS 45

9. EXCEPTION IDENTIFICATION 47

9.1 REQUIREMENTS FOR EXCEPTION IDENTIFICATION 47
9.2 METHODS FOR EXCEPTION IDENTIFICATION 47

10. STORAGE MANAGEMENT **51**

10.1 REQUIREMENTS OF THE EXCEPTION STORAGE MANAGER 51
10.2 DESIGNING THE EXCEPTION STORAGE MANAGER 51

11. EXCEPTION HANDLING AND COMPILER ISSUES **53**

11.1 TRANSLATING THE TRY-EXCEPT CONSTRUCTION 53
11.2 TRANSLATING THE TRY-FINALLY CONSTRUCTION 54
11.3 TRANSLATING RAISE STATEMENTS 55
11.4 OTHER ISSUES 56

12. THE EH METHOD USED IN THE MEHL COMPILER **57**

12.1 THE TARGET SYSTEM SPECIFICATION 57
12.2 THE METHOD IN SUMMARY 57
12.3 THE STATIC TABLES 58
12.4 OBJECT CLEANUP INFORMATION 61
12.5 RETRIEVING THE SIZE OF A FUNCTION'S LOCAL ENVIRONMENT 64
12.6 RUNTIME TYPE INFORMATION FOR EXCEPTION TYPES 64
12.7 ADDRESSING THE LOCAL ENVIRONMENT OF MEHL FUNCTIONS 65
12.8 STATIC TABLES EXAMPLE 66
12.9 CODE FOR EH CONSTRUCTIONS AND STATEMENTS 70

13. THE EH RUNTIME MECHANISM **73**

14. THE SYSTEM DESIGN OF THE MEHL COMPILER **93**

14.1 THE STRUCTURE OF THE COMPILER 93
14.2 COMPILATION OF MULTIPLE MODULES 93
14.3 THE SCANNER. 93
14.4 THE PARSER 94
14.5 THE MEHL COMPILERS INTERMEDIATE REPRESENTATION 101

15. THE CODE GENERATOR FOR THE MEHL COMPILER. **105**

15.1 OVERVIEW OF THE CODE GENERATOR 105
15.2 USE OF REGISTERS 106
15.3 TRANSLATION OF A FUNCTION / PROCEDURE 107
15.4 GEN_VARIABLE_INIT_OR_DONE 110
15.5 THE FUNCTION GEN_OBJECT_DECLS 111
15.6 GENERATING RTTI FOR EXCEPTIONTYPE'S 112
15.7 GENERATING CODE FOR THE EXCEPTIONAL CLASSES 112
15.8 GENERATING CODE FOR THE EXCEPTIONHANDLER CLASS 113
15.9 CODE GENERATED FOR INDIVIDUAL DAG NODES 114

16. SPECIFICATION OF TESTS & TEST SUMMARY **119**

16.1 TESTING THE MEHL COMPILER'S FRONT END. 119
16.2 TESTING THE MEHL COMPILER'S BACK END. 119
16.3 TESTING THE EH RUNTIME MECHANISM. 120
16.4 SUMMARY OF TEST RESULTS 120

17. CONCLUSION **123**

APPENDIX A INHERITANCE OVERVIEWS FOR MEHL IR **127**

APPENDIX B OBJECT MODELS FOR MEHL IR **129**

APPENDIX C THE STDIO UNIT **135**

OVERVIEW OF THE STDIO UNIT 135

APPENDIX D THE MEHL RUNTIME LIBRARY **137**

HEAP-MANAGEMENT 137

Other internal routines 137

APPENDIX E GENERATING CODE FROM DAGS 139

APPENDIX F THE 80286 PROCESSOR 143

The 80286 Registers 143
Common 80286 instructions 143
Memory Models 144

APPENDIX G THE TURBO ASSEMBLER LANGUAGE 145

Declaring and using procedures 145
Declaring and using objects 145

APPENDIX H TESTS AND TEST RESULTS 147

Testing type-checks in the MEHL compiler 147
Testing Exception Handling 147

APPENDIX I REFERENCES 159

Exception Handling related: 159
Compiler related: 159
Pascal & Assembler language: 159
General nature: 159

1. Introduction

This thesis will present different methods for implementing exception handling (EH) in modern object-oriented languages.- Exception handling which, as considered in this thesis, presents a conformable and highly attractive, alternative way of detecting and reacting to run-time errors in non-trivial applications.

Unfortunately, the support for EH is very complicated and potentially very costly in performance for the end-application, especially for some modern object-oriented programming languages such as C++[1].

This thesis will discuss existing methods for implementing EH, show their advantages and disadvantages and propose a new unified and very efficient method. As a major part of this thesis, a compiler is developed featuring an object-oriented programming language incorporating handling of exceptions using this method.

1.1 The basis for this thesis

For some years the author's programming language of choice has been C++, but only recently has EH been made available for this language. It is the inadequate methods used in many of these early C++ compiler's implementations of EH that have motivated this thesis.

The research of the subject started with an inquiry posted to Internet and some probing at the University Library, which eventually let to the retrieval of five recent articles discussing the handling of exceptions (See Appendix I for references). Apart from one article [Philbrow90] which includes an excellent survey of the general subject of exceptions in several languages, the articles are about handling exceptions in C++.

Unfortunately the articles discuss primarily the same general method for handling exceptions, called *the registration approach* (See chapter 6). The general method is portable but not very efficient.- Apart from being merely referred to, the alternative and much more efficient *static table approach*, is brought up only briefly by [Koening90] and there mostly as an existence proof for the method.

As a basis for this thesis the author has had no prior knowledge of methods for handling exceptions in (object-oriented) programming languages. Further the author has had only limited pre-experience with EH concepts in C++ and other programming languages. Regarding the compiler that has been developed for this thesis, the author has taken two university courses in compiler construction and has as such had some experience in constructing compilers before.- Although not to the extent that the experience included constructing compilers for object-oriented programming languages and the generation of Intel 80x86 assembler code output as this thesis' compiler does.

1.2 So what is exception handling

The general idea of EH is that when one part of a program runs into a problem, usually an error, which it cannot (or will not) cope with at the time and place, it *raises (or throws or signals [2])* an exception.- The raising of an exception transfers control to another part of the program, the *exception handler*, which will catch the exception, and react sensibly according to the problem at hand. Typically different kinds of exceptions can be thrown, and different exception handlers can exist, one for each type of exception and usually in multiple scopes.

---

[1] Exceptions are more complicated in C++, than in most other programming languages because of the notion of constructors/destructors in the C++ object model, which the exception mechanism must comply with.

[2] In this thesis these three terms will be used interchangeably, however in other exception terminologies their meaning may differ.

After being notified about the problem the exception handler decides what to do next. Asking the user for new or additional input followed by a retry of the operation originally at fault is one example of action that can be taken by an exception handler. To simply abort the program is another example.

Below can be seen an example of the basic syntax and use of exceptions in each of the modern programming languages C++ and Modula-3. The examples show an extract of a program that needs to open a file and then do some math calculations with its data. As it can be seen, the two examples are very much alike. Apart from the raises-specification, only the trivial syntax differences in the two programming languages set them apart.

C++ example:

```
void f()
{
 try_open_a_data_file();
 if (file_not_opened)
  throw FileErr(filename);
 do_some_math(file_data);
 ...
}

main()
{
 try {
  ...
  f();
  ...
 }
 catch(MathErr m) {
  // Do something with m.
 }
 catch(FileErr f) {
  // Do something with f.
 }
}
```

Modula-3 example:

```
PROCEDURE f()
 RAISES {FileErr, MathErr} =
BEGIN
 try_open_a_data_file();
 IF file_not_opened  THEN
  RAISE FileErr(filename);
 END;
 do_some_math(file_data);
 ...
END f;

BEGIN
 TRY
  ...
  f();
  ...
 EXCEPT
 | MathErr(m) =>
   ( * Do something with m *)
 | FileErr(f) =>
   (* Do something with f *)
 END;
END Module.
```

Two run-time errors have been considered in the examples above: The failure to open a file, and the event of some math error in the calculations. Both errors must be properly detected and handled. For example, if the file cannot be opened in the procedure *f*, the subsequent statements, including *do_some_math(...),* must not be executed, instead some action must be taken, e.g. writing an error on the user's screen and/or asking for another filename, etc.

The first failure is handled by the *throw/RAISE* statement in *f*. If the file cannot be opened, the procedure aborts and control is transferred directly to the statements belonging to the exception handler *catch(FileErr)* or *EXCEPT | FileErr =>* in main. The name of the offending file is supplied with the exception as a parameter, which enables the handler to output an informative error message (Not shown in the examples).

The next source of failure is located in the procedure *do_some_math*. Somewhere in this procedure or the procedures/functions it calls directly or indirectly, a throw *MathErr(...) / RAISE MathErr(...)* can be issued as a response to an error condition (Not shown in the examples). When this happens, control is passed directly to the nearest *MathErr* handler in the call chain, which providing the sample procedure *do_some_math* does not do EH itself, will be the handler supplied by main.

Both *FileErr* and *MathErr* are user-defined entities. In C++ exceptions can be of any type, including objects. In Modula-3 exceptions must be names declared in an EXCEPTION-construction much like the TYPE or VAR constructions, but with an optional single exception parameter that can be of any type. When the exception is handled, an instance of the exception type, containing the relevant user-defined information about the exception that has been supplied by the signaller, are provided to the handler through the handler's optional parameter.

The *MathError* is an example of a runtime failure that can potentially happen outside the programmer's written code. For example, the procedure *do_some_math* can be a library routine. If this is the case, the *do_some_math* procedure that identifies and understands the error, probably do not know exactly what to do when the fault arises: Abort program, output error message, retry operation ?- Issuing an exception is in this case, a perfectly good answer to the problem. The raising of an exception transfers the control to the first handler that knows what to do with the problem, along with all relevant information about the error. In the examples above, the *MathErr* handler in main is the nearest (and only) candidate. The handler decides what to do with the error, possibly inspecting any relevant error information (From *MathError* handler parameter *m*).

In Modula-3 it must be specified when exceptions are allowed to propagate out of a procedure or function. In Modula-3 the exceptions that may propagate are explicitly specified by a RAISES-construction in the function's header; procedure *f's* header in the example. In C++ all exceptions are allowed to propagate as a default, but propagation can be limited in a similar manner to Modula-3 by specifying an explicit throw list as part of the function declaration. In both languages, if an exception that is not permitted to propagate is terminating a function, the normal response for the run-time system is to stop the program and issue a suitable error message.

1.3 Error-handling without Exception handling

The type of errors addressed in the examples in the previous section can of course also be tackled without EH, but not as simple and conformable as with EH.- Without EH, a large number of checks of function results and/or inspections of global error variables is needed together with a massive number of conditional statements to propagate errors and to discriminate between execution of normal and error handling code.

When done properly, error handling without exceptions is often tedious to write, it may well take up more place then the real processing, and it may also destroy an otherwise pure design.

Finally it is worth noting that error handling in the traditional way, without exceptions, imposes an inevitable run-time overhead, however limited it may be, due to the conditional statements related to the explicit handling and propagation of errors.

2. Exception handling requirements

This chapter discusses various EH models and specifies the EH model used in this thesis. The requirements for the design and implementation of EH adapted from this model are discussed and presented, as are the additional requirements due to the object-oriented programming language of which it will be a part.

2.1 Exception handling model specification

Roughly, most EH models can be classified according to the following terms (See [Pilbrow90] and [Koenig90]):

- Are exceptions *synchronous* and/or *asynchronous*?
- Is the *Termination model* or the *Resumption model* used?
- Are exceptions handled in a *single level* or *multi-level* fashion?
- Can exceptions be *parameterized* and if so, in a fixed or user-defined manner?
- How are an exception and its handler matched?

Synchronous exceptions are thrown in the context of the running program, in response to a problem/error such as an array range check, a resource exhaustion or some other specified condition. Examples of asynchronous exceptions are keyboard- and other system interrupts. Only synchronous exceptions are addressed in this thesis!

The use of the resumption model versus the termination model decides whether or not the signaller of the exception continues to exist while the handler is executed, and can be resumed were it left off when the handler completes. - In the resumption model this action is supported, in the termination model, resuming is disallowed and execution is always transferred to the exception handler, and from there to the next valid instruction in the handler's scope. Note that the word *termination* only applies to the signaller of the exception, not to the handler and/or the program. The program is free to continue execution after the exception handler has finished!

It has been an issue of discussion among language designers whether explicit support for resumption in the EH model is truly desirable. The popular view seems to be that support for resumption is not advantageous (see [Koenig90] section 7, and [Philbrow90] section 3.1.2), and that ordinary function calls can be used to archive the same means. A view that is shared by the designers of the programming languages C++ and Modula-3. Specifically, the designer of C++, Bjarne Stroustrup, and his co-author Andrew Koenig states in [Koenig90] that "resumption provides ... a contorted, nonobvious, and error-prone form of coroutines".

As the EH in the programming languages C++ and Modula-3 have been the main inspiration for this thesis, and because resumption does not seem to be desirable, this thesis will only consider the termination model!

In the single-level EH model, an exception must be handled by the part of the program that raised it. In the multi-level model, an exception can propagate indefinitely back the call chain until a matching exception handler can be found. -For this thesis a multi-level capability has been targeted.

In a parameterized EH model, exceptions can be supplemented with additional information about the exception. This information is in the most general case taking the form of a parameter to the exception being thrown. The parameter can be used to discriminate between different variations of the exception, to bring vital information to the exception handler or can simply be a text message to be printed. Some parameterized exception models are fixed, in the sense that only strings as parameters are supported, in other models the type of information is user-defined.- This thesis will consider user-defined parameterized exceptions!

Some method must be used to match exceptions raised, with their corresponding exception handlers. This can be done by comparing types or by comparing values. - In most models exceptions are matched by their type. This is also the case in this thesis.

2.2 Exception handling design requirements

As mentioned above a synchronous, termination based, multi-level and general parameterized EH model has been adapted.- A choice of EH classification which corresponds to the models employed in a number of existing programming languages. For example C++, Modula-3, and Ada with the exception *(in the literal sense)* that Ada does not support the parameterized exception model.

Certain self-evident demands must be placed upon the the implementing of EH in any modern programming language:

- It is easy and safe to use.
- It presents no problems with BIG programs.
- It can be used with and across separately compiled modules.

Naturally the behaviour of an EH mechanism should always be determinable and not likely to introduce new possibly subtle forms of errors to the programmer. This implies that is must be possible to specify the exceptions allowed to propagate from a function, all of which must be handled by its caller.

EH must be usable also with large programs. An implication of this requirement is that some sort of exception grouping must be possible. The need for an exception grouping facility should come forth clearly when one considers a major programming project, where at some point hundreds or more different exceptions can potentially happen. Such a situation will be intolerable if one had to write an explicit exception handler for each possible exception at each such point. This problem will be much reduced if, on the other hand, exceptions can be arranged into different groups with the option of only having to write exception handlers on some group basis.

Having already addressed the subject of using EH in large applications, it is self-evident that the multi-module concept found in nearly all modern programming languages must be supported also. An example of such need of support is the situation of an exception being raised in one module and handled in another. In such a situation the compiler will certainly not be able to know, at compile-time, where a signaled exception will- or can be handled.

2.3 Exception handling implementation requirements

Furthermore, the following properties are demanded or deemed desirable in an implementation of the EH mechanism. All demands mentioned have a serious effect upon the implementation of the EH mechanism.

- Run-time overhead is minimal!
- Space overhead is small.
- It's possible to port.
- Support of mixed-language programming.

It is a basic requirement that EH should be implemented in a way that introduces only minimal run-time overhead! The inevitable run-time cost associated with exceptions can be divided between:

1. The normal program flow.
2. The cleanup and the transference of control from the signaller to the exception handler, which takes places when an exception is actually being raised.

To clarify matters, run-time costs will always exist but it is natural to assume that some of the costs can be moved from the normal program flow to the exception situation (or the other way).
If the view that exceptions should be used primarily for errors, and not for yet another way of normal execution control, is taken, the issue of minimal run-time overhead boils down to the case of normal program flow. This thesis will take that view.- Henceforth the use of exceptions will be considered restricted to error situations!

Only the normal program flow's run-time overhead will be tried minimised, at the likely cost of run-time overhead involved when raising an exception. In other words, during the normal execution of a program, none or at least very few exceptions should occur. For example, exceptions should not occur as a way of terminating loops, as an end of file notification, or as a response to a user-related action such as a key press or the like!

Note that in the above consideration of run-time cost associated with exceptions, it is assumed that the integration of an EH mechanism in the compiler will not seriously affect the quality of the generated, optimised, code (no hidden penalty) [3].

An implementation that does not impose a large space overhead is required. The support for EH must not mean that compiled programs increases dramatically in size.

An implementation which can be ported is demanded.- The implementation of the EH mechanism must not depend on the existence of certain operating system features, an uncommon CPU architecture or instruction set.

Finally it is desirable that the implementation of the EH mechanism can be used in situations where mixed-language programming is allowed. Many languages, most notably C++, support the notion of transferring control to and from a C or Assembler language function. For example this implies that program control can happen in the following way :

> Function *A* calls *B* which calls *C*, where *A* and *C* are functions written in the implementation language and *B* an external function written using another language.

If an exception is thrown in function *C* and no handlers for that exception exist in *C* the exception should be propagated back to its indirect caller, function *A*. Exceptions is considered only in the context of the implementation language. It is not demanded that external functions should be able to raise and/or handle exceptions by itself. Therefore the function *B* is left out in the search for an exception handler in the above described event of an exception being throw in function *C*.

2.4 Requirements due to programming language

The programming language in which EH is incorporated may force additional demands upon the design and implementation of EH. This thesis considers EH in object-oriented programming languages, which in some but not all cases introduce new significant problems to be addressed in an implementation.

The Modula-3 programming language is an example of an object-oriented language that apparently does not add to the complexity of the implementation of EH. The situation is different in the C++ programming language, where the EH mechanism is required to properly cleanup automatic[4] objects when propagating back to the exception handler. The cleanup involves explicitly calling the destructor for each successfully constructed object in each function frame it encounters when propagating an exception.

As described in chapter 3 an object-oriented programming language with the powerful C++ style object model that embodies the notion of *constructors* and *destructors* is targeted for this thesis. This means that as it is the case for C++, the EH mechanism considered in this thesis must be required to properly destruct previously

---

[3] This is not always true since effective optimization across exception regions is difficult to perform.

[4] Automatic objects are C/C++ terminology for objects of local duration, which are created on the stack when a block or function is entered, and deallocated when a program exits that block or function.

constructed automatic objects during exception propagation! The mechanism must ensure proper destruction of objects, including difficult cases such as partially constructed objects and array of objects etc.

3. Requirements of programming language

This chapter will present the basic requirements for the object-oriented programming language incorporating EH as specified in chapter 2. The requirements presented here form a specification of the capabilities of the compiler that will be developed for this thesis.

3.1 Object model

From the initial specification of the scope of this thesis, the C++ programming language object model and its influence on the EH mechanism has been targeted. Thus, a scaled down C++ style object model that embodies the notion of *constructors* and *destructors* for objects has been chosen for this thesis. More to the point, an object model with the following properties:

- Single inheritance.
- Polymorphism (virtual methods)[5].
- Data abstraction with private data members and public member methods, with constructors and destructors as special method cases.

The above properties, except for the constructor/destructor part borrowed from C++, represent the minimal requirements for a programming language to be object-oriented.

3.2 Language constructs

The following additional basic requirements for the programming language have been decided upon:

- Static, strongly typed, imperative.
- Syntax similar to Pascal (for simplicity).
- Integration of the most common of normal concepts in structured programming languages such as:
  - Global procedures/functions.
  - Assignments.
  - Conditional statements.
  - Loop statements.
  - Heap-management (An issue with EH, also we want our *pointers* to point to something).
  - Return statements (An issue with EH).

- Support of the following data types from which new types can be constructed.
  - Integer
  - String (Any language should be able to do a “Hello world”).

[5] Here polymorphism is referred to with respect to inheritance, as it is the case for virtual methods in C++ etc.

- Pointer (Without pointers, objects are not of great use).
- Array (Is an issue with EH and the destruction of automatics).
- Object

3.3 Identifiers and scopes

Identifiers can be types, variables, procedures, functions or methods. Variables can be global, local or of object members. Named types are always global. Procedures and functions are global, methods are in scope of the containing object and its descendants.

3.4 Modules

It must by verifiable that the EH mechanism facilitates separately compiled modules. Consequently a program written in the programming language may consist of several modules.- One main program module and several additional modules, which can be compiled separately. Identifiers can be imported and exported from/to other modules.

4. Design of exception handling mechanism

This chapter will present the design and syntax of the EH support as incorporated in the compiler used for this thesis. The design and syntax are borrowed from the Modula-3 programming language.

4.1 The RAISE statement

Exceptions can be thrown at any point using a RAISE statement with an exception as an optional parameter:

- `RAISE exception_type [.constructor(param 1, param 2, ...)]`

  or

- `RAISE`

Where *exception_type* is an object type descended from a standard object called *Exception* which is a supertype for all user-defined exceptions. The type can be supplemented with an explicit specification of the *exceptions_-type's* constructor argumented with actual parameters. If a RAISE statement is issued inside an exception handler and no exception type is specified, the current exception being handled is re-raised and propagated further back the call-chain until a new handler is found.

4.2 The exception control constructions

Two kinds of exception control constructions are possible - A try-except and a try-finally statement:

-
```
TRY
 guarded statements
EXCEPT
 | exception_type1(id) => statements
 | exception_type2(id) => statements

 or

 | exception_type1, exception_type2, ... => statements
END
```

-
```
TRY
 guarded statements
FINALLY
 final statements
END
```

In the try-except construction, if an exception is raised in one of the guarded statements, control will be transferred to the nearest matching exception handler. This construction corresponds to the try-catch statements found in the C++ programming language. If a single exception type with an identifier is specified, that identifier will act as a variable of the typed exception. If no identifier is specified, any number of exception types can be specified for the same action.

In the try-finally construction, the finally statements will always be executed eventually. If an exception is raised in one of the guarded statements, execution will continue immediately at the statements in the finally section, otherwise the finally statements will be executed after the last guarded statement. None of the finally statements do, however, handle the exception. After the last statement in the finally section has been executed, the exception is re-raised and must be taken care of by a proper exception handler. Note that the guarantee that the finally statements are always executed includes non-exceptional situations, such as a statement that jumps out of a guarded section of statements (as a guarded return statement etc.).

The try-finally construction is not supported in C++, which is unfortunate as the typical resource acquisition example shown below will demonstrate:

```
TRY
 OpenFile(...);
  ...
  IF some_file_error THEN RAISE FileError.Init(...); ENDIF;
  ...
FINALLY
 CloseFile(...);
END;
```

In the example above, it is guaranteed that the file will always be closed. In case a file error is identified, the try block will be immediately terminated, the file closed and an exception propagated to a matching handler (not present in the example). In case of no file error, the file will likewise be closed, after the execution of the last of the statements in the try section has finished.

Noteworthy, and in all fairness to C++, automatic objects can to some extent be used to manage allocation/-deallocation of resources by using constructors and destructors to perform these actions respectively. In some cases such use of automatic objects to handle resource management is within reason, but in other cases a more or less artificial object encapsulation is cumbersome and the try-finally construction is the only appropriate alternative.

4.3 Exceptions are objects

All user specified exceptions must be defined as objects derived from a standard *Exception* object. The requirement for all exception types to be objects derived from a common superclass provides a powerful, but simple syntax for specifying exceptions.

Since exceptions are objects they comply with the rule that any object *B* descended from *A is an A*. Therefore any exception handler for *A* will also handle any derived exceptions, including *B* (unless a handler for *B* has been explicitly specified). In this way the required exception grouping feature has been acquired without extending the language. If two distinct exception handlers are wanted, one for *A* and one for *B*, the derived object type furthest down the inheritance hierarchy, *B*, must be specified first.

Note that by specifying an exception handler for object type *Exception*, any exception regardless of type will be handled, as all exception types must be derive directly or indirectly from the object *Exception* (This is different from Modula-3 where an ELSE clause is used).

4.4 The RAISES construction

It is required from the EH mechanism that the user must be enable to specify the exact range of exceptions allowed to propagate out of a function. In the Modula-3 and C++ programming languages run-time checks for valid exceptions are activated through explicit declaration of the exceptions that may be propagated for each procedure or function.

The implementation language will address the problem in the same way, by extending the function header with an optional RAISES set of possible exceptions. Extraordinarily, an approach similar to C++ and contrary to Modula-3 has been elected which frees the user from unnecessary typing of exception specifications. With this approach a not present RAISES specification will mean that all exceptions are allowed!

Below the syntax for a RAISES specification is presented:

- RAISES {exception_type1, exception_type2, ...}

Where *exception_type1* & *exception_type2* are the (super-)type names of two of the exceptions that the procedure or function in question may propagate out. If an empty raises-set is specified it means that no exceptions can propagate.

5. The MEHL programming language

This chapter will present the complete language design of a programming language called **MEHL**[6], which can be translated by the compiler developed for this thesis. The presentation of the programming language is followed by clarifying examples and explanations.

5.1 The syntax of MEHL

The MEHL programming language borrows most of its basic syntax from *Borland Pascal With Objects 7.0*[7], except for the EH constructions that are taken from the Modula-3 programming language. Consequently the different elements in the MEHL compiler are inspired by three existing programming languages: The object model is inspired by C++, EH by Modula-3 and the basic syntax by Borland Pascal!

Below the decided syntax of MEHL is presented in Extended Backus-Naur form (EBNF).

```
module → program
        | unit

program → PROGRAM module_id ';' [uses]
           program_parts main

unit → UNIT module_id ';' [uses]
        INTERFACE program_parts
        IMPLEMENTATION [uses] program_parts [main]

uses → USES [ module_id { ',' module_id } ] ';'

main → statements '.'

program_parts → [TYPE { type_decl } ]
                 [VAR { var_decl } ]
                 { func_or_proc | method }

type_decl → type_id '=' type ';'

type → OBJECT [ '(' object_type_id ')' ]
       { var_decl | method }
       END
      | '^' type
      | ARRAY '[' const_expr ".." const_expr ']' OF type
      | type_id

var_decl → var_id ':' type ';'

func_or_proc → FUNCTION function_id fparams ':' type_id ';' [raises] body
             | PROCEDURE procedure_id fparams ';' [raises] body

fparams → ['(' [fparam { ';' fparam }] ')']

fparam → [VAR] param_id ':' type_id
```

6 MEHL - **M**y **E**xception **H**andling **L**anguage

7 Popular Object-Oriented Pascal compiler on the PC platform

```
method → FUNCTION [object_type_id '.'] function_id
          fparams ':' type_id ';' [raises] [VIRTUAL ';'] body
        | (PROCEDURE | CONSTRUCTOR | DESTRUCTOR)
          [object_type_id '.'] procedure_id fparams ';'
          [raises] [VIRTUAL ';'] body

statements → BEGIN { statement ';' } END

body → [VAR { var_decl } ] statements ';'
     | EXTERNAL ';'
     | FORWARD ';'

actual_parameters →  expr { ',' expr }

raises → RAISES '{' [ exception_type_id { ',' exception_type_id } ]'}'

statement → var_id_ref ":=" expr
          | var_id_ref '(' actual_parameters ')'
          | var_id_ref '.' method_id '(' actual_parameters ')'
          | NEW '(' var_id_ref [ ',' constructor_call ] ')'
          | DISPOSE '(' var_id_ref [ ',' destructor_call ] ')'
          | IF conditional_expr THEN { statement }
            [ ELSE { statement } ] ENDIF
          | WHILE conditional_expr DO { statement } END
          | REPEAT { statement } UNTIL conditional_expr
          | RETURN [ expr ]
          | TRY { statement } EXCEPT { exception_handler } END
          | TRY { statement } FINALLY { statement } END
          | RAISE [ exception_type_id [ '.' constructor_call] ]

constructor_call → constructor_id '(' actual_parameters ')'
destructor_call → destructor_id '(' actual_parameters ')'

call → var_id_ref '(' actual_parameters ')'
     | var_id_ref '.' method_id '(' actual_parameters ')'

var_id_ref → var_id { '^' | '[' expr { ',' expr } ']' }

exception_handler → '|' exception_spec "=>" statements ';'

exception_spec → exception_type_id '(' var_id ')'
               | exception_type_id { ',' exception_type_id }

const_expr → '(' const_expr ')'
           | NUMBER
           | ('+' | '-') const_expr
           | const_expr ('+' | '-' | '*' | '/') const_expr

conditional_expr → '(' conditional_expr ')'
                 | NOT condition_expr
                 | expr '<' expr
                 | expr "<=" expr
                 | expr '>' expr
                 | expr '>=" expr
                 | expr '=' expr
                 | expr "<>" expr
                 | expr

expr → '(' expr ')'
     | NUMBER
     | ('+' | '-') expr
     | expr ('+' | '-' | '*' | '/') expr
     | var_id_ref '(' actual_parameters ')'
     | var_id_ref '.' method_id '(' actual_parameters ')'
     | call
```

```
       | '@'[8] var_id
       | TEXT_STRING

xxx_id → ID
object_type → type
```

The EBNF grammar for the MEHL programming language is a complete but intelligible rewrite of the compiler's LALR(1) grammar.- The original grammar is much longer and much more complicated.

As it can be inspected, the MEHL programming language fulfils, syntax-wise, all the requirements of a programming language implementing the required exception mechanism as discussed in chapter 2-3.

5.2 Reserved words in MEHL

Below can be observed an overview of the reserved words in the MEHL programming language.

TRY, EXCEPT, FINALLY, RAISE, RAISES, NOT, PROGRAM, UNIT, TYPE, VAR, INTERFACE, IMPLEMENTATION, USES, END., OBJECT, PROCEDURE, FUNCTION, VIRTUAL, FORWARD, EXTERNAL, NEW, DISPOSE, BEGIN, END, DESTRUCTOR, CONSTRUCTOR, RETURN, IF, THEN, ELSE, ENDIF, WHILE, DO, REPEAT, UNTIL, ARRAY & OF.

5.3 Examples of MEHL programs

Below, are listed two examples of valid MEHL program : The compulsory "Hello Word" program, and a slightly larger, but hopefully equally easy to understand, example involving units, objects and exceptions.

Example 1

```
PROGRAM HelloWorld; # Program 1 example (comment)

USES stdio;

BEGIN
 WriteString('Hello World');
END.
```

Example 2

```
UNIT A; # Unit for example 2.

INTERFACE

TYPE
 MathFault = OBJECT(Exception) END;

 T = OBJECT
      a: INTEGER;

      CONSTRUCTOR init();
      BEGIN
       a:=42;
      END;

      FUNCTION test(i: INTEGER): INTEGER; FORWARD;
     END;
```

[8] The character @ is used as an address operator.

```
IMPLEMENTATION

FUNCTION T.test(i: INTEGER): INTEGER;
BEGIN
 IF i=0 THEN RAISE MathFault; ENDIF;
 RETURN i/a;
END;

END. # A


PROGRAM B; # Program for example 2

USES A, stdio;

VAR
 x: T;
 i: INTEGER;

BEGIN
 TRY
  i:=ReadInt();
  WriteInt(x.test(i));
 EXCEPT
 | MathFault => WriteString('Error in math operation');
 END;
END. # B
```

Firstly, note the use of the “#”-character to insert single line comments. It can be used as an alternative to the standard Pascal multi-line *(* *)* comments which are also supported.

The “Hello Word” example should present no problems so only the more complicated example 2 will be discussed. Example 2 inputs a number.- Then, if the number is different from zero, it outputs that number divided by 42, otherwise it outputs an error message. Example 2 is divided into 2 modules for the purpose of the example.- A unit module *A* and a program module *B*.

In a unit, only text between *INTERFACE* and *IMPLEMENTATION* is visible from the outside, so *MathErr* and *T*, which are used in *B*, must be defined in that section in the unit *A*. *MathErr* is declared as an object type originating from *Exception*. It is followed by a definition of an object type *T*, with a constructor named *init* and a test function named *test*. For the sake of this example, it is show that methods can be defined either inside (as the constructor *init* is) or outside the object type declaration (as the function *test* is).

The program *B* imports from two units. The standard *stdio* unit and the unit *A*. The *stdio* unit provides basic I/O support through the procedures *WriteInt*, *WriteString*, *ReadString* and the function *ReadInt*. See Appendix C for more details.

5.4 MEHL, the details

The following will not contain a comprehensive explanation of the MEHL programming language, but only a mentioning of those parts that set MEHL apart from ordinary Pascal, aside from the EH constructions which has already been covered in chapter 4.

5.4.1 Modules

A compilation can consist of one program module and a number of unit-modules . Each module can import from other units with the *uses* construction. A unit exports all global types, variables and functions/procedures that are

declared inside its interface section. Everything declared in a unit's implementation section is private to that module.

Standard Pascal does not have any module capability,- this feature has been taken directly from Borland Pascal. Compared to Modula-2, a MEHL unit is a combination of a definition and an implementation module. It should be noted that the combination of a definition and an implementation module is not a demand, but a convenience for the module's programmer which limits the number of files in a project. - With this kind of unit, it is common practice to strip out the implementation section when a programmer distributes a unit to others.

In order to eliminate the problem of in-which-sequence-to-compile a list of interdependent modules, the MEHL compiler is required to be able to compile a set of modules in one single compilation. Furthermore if a one-compilation model is adopted, swapping of symbol tables to secondary storage can be eliminated thus increasing the compiler performance and making the compiler less complicated!

Consequently the MEHL compiler is designed to support the one-compilation model for a program. In the MEHL compiler, when a program, consisting of *n* modules, is to be compiled the filenames of all *n* modules are given as parameters to the compiler. Then, the compiler will deduce the correct compilation sequence and compile all interfaces followed by a compile of existing, outdated implementations. The program-module will be compiled after any unit-modules, as it always will be the last in the dependency list.

5.4.2 Types

MEHL incorporates the build-in types *INTEGER*, *PCSTRING* and *EXCEPTION*, together with the constructions *POINTER*, *ARRAY* and *OBJECT* which can be used to define new types.

PCSTRING is a special pointer-to-null-terminated-string type which is used to facilitate a minimum of string support. Whenever the compiler comes across a text-string the type will be interpreted as a *PCSTRING*. Note that *PCSTRING* is not the same as, nor as powerful as, the type STRING, found in most Pascal compilers.

5.4.3 Objects

An object type can inherit from an ancestor by supplying the ancestors name in parentheses. If a variable or a method is declared in both an ancestor object type and the derived object type, the derived object type overrides its ancestor. Virtual methods must also be declared virtual in its derived object types.

Note that in MEHL any number of different constructors and destructors can be specified for an object type as long as they are named differently. The first constructor/destructor specified which has no arguments, act as a default constructor/destructor and will be called implicit by the compiler for all global, local or sub-objects of that type. Other constructors and destructors, including those with arguments, can only be used in an explicit way, as in the *NEW, DISPOSE* or *RAISE* statements.

The construction order is the reverse of the destruction order for objects (First created, last destroyed). Most complex is the construction/destruction order for sub-objects in objects. For a non-trivial object the order of construction and destruction is shown below:

1. Construct ancestor.
2. Construct first object member, second object member, ...
3. Construct first local object variable for constructor, second object variable, ...
4. Construct the object.

Figure 1 Construction order for an object

1. Destroy the object.
2. Destroy last local object variable for destructor, last but one object variable, ...
3. Destroy last object member, last but one object member,...
4. Destroy ancestor.

Figure 2 Destruction order for an object

Using C++ terminology methods are public and data members are private. Methods must be declared inside their object type, but the body of the methods can be defined later outside the object type. When defining a method outside its object type, the name of the method must be prefixed with the object type name and a punctuation.

5.4.4 Exceptions

Exceptions are objects of type *Exception* or type derived from *Exception*. As such exceptions are defined and manipulated just like ordinary objects.

5.4.5 Miscellaneous topics

A return statement equal to that of Modula-3 has been adapted. When a return statement is guarded by a try-finally construction the finally statements are executed before returning from the function.

Unlike standard Pascal, a procedure/function call must always be supplied with parentheses even when there are no parameters. - This improves the performance of the compiler, because the parser can distinguish between a variable and a function call.

The *NEW* and *DISPOSE* statements allocate memory accordingly to the size of the type of the item pointed to by its first argument. When allocating/disposing objects, a second optional argument can be supplied which specifies a constructor/destructor to be called. If none are supplied, the default constructor or destructor will be used.

It is the programmer’s responsibility to free all memory successfully allocated with the *NEW* statement with a corresponding *DISPOSE* statement. If an exception occurs during the execution of a *NEW* statement (i.e. in the constructor called by *NEW*), no corresponding *DISPOSE* statement should follow, as it is then the responsibility of the EH mechanism that no memory is actually allocated.

6. Methods for Handling Exceptions

This chapter will present an overview of the issues to be addressed when handling exceptions and the different methods that can be applied.

6.1 Functional areas in runtime mechanism

Using the same structural decomposition as [Cameron92] the EH runtime mechanism can be divided into four major functional areas:

- Transfer of control
- Object Cleanup
- Exception Identification
- Storage management

When an exception is raised the EH mechanism must transfer control from the signaller to the handler of the exception. The *transfer of control* area covers this fundamental aspect of the EH mechanism which is discussed in detail in chapter 7.

When propagating an exception all previously constructed objects, that are brought out of scope, must be destroyed. The *object cleanup* area covers this aspect of the EH mechanism which is discussed in detail in chapter 8.

When an exception is raised the type of the exception must be matched against different handlers in order to find the corresponding handler. Also, before propagating out of a function, the type of the exception must occasionally be checked against a raises-set specification for the function. The *exception identification* area covers these aspects of the EH mechanism which is discussed in detail in chapter 9.

When an exception is raised, storage must be allocated to hold the exception object. The *storage management* area covers this aspect of the EH mechanism which is discussed in detail in chapter 10.

6.2 Compiler issues

The compiler must generate code for the TRY-EXCEPT and TRY-FINALLY constructions and the RAISE statements in addition to static tables with the information necessary for the EH mechanism to work. Finally, on the whole, the compiler must ensure that the code generated does not conflict with EH. These compiler issues are discussed in chapter 11.

6.3 General considerations

The handling of exceptions must be done at runtime, since it is generally not possible to predict in advance which handler to transfer control to, identify which exception has been raised, where to perform object cleanup during the propagation of an exception and how much memory to pre-allocate for exception storage.

Usually the EH runtime mechanism is compiler-specific and a part of compiler's runtime library. However in some cases support for EH is provided through the operating system.

A number of modern operation systems support software-generated exceptions and their handling. On PC's EH is supported by the IBM OS/2 operating system and by the Microsoft Win32 application programming interface used in a number of their most recent operating systems including NT & Windows 95.

An implementation based on a consistent EH mechanism which is supported directly by the operating system is, unlike a compiler specific implementation, usable with all programming languages and machines. Still, support for EH is non-existing for most operating systems.- Often the term exception, if it is used, is associated only with hardware faults.

6.4 Overview of alternative implementation methods

The *transfer of control* and *object cleanup* areas in the EH mechanism are closely connected. They are both very important for how portable and efficient the mechanism is. It is the implementation of these areas that decides the overruling part of the cost of the EH mechanism. In comparison the *exception identification* and *storage management* areas are of isolated nature and only of limited importance with respect to how portable and efficient the EH mechanism is.

Basically, for methods, two different and well-known approaches to EH exist:

- The dynamic registration approach.
- The static table approach.

Due to lack of common and accurate terminology the naming of these approaches has been done by the author of this thesis. In [Cameron92] the approaches are referred to as portable and non-portable exception handling respectively, and in [Koenig90] as portable and efficient exception handling respectively. The dynamic registration approach is described in detail in [Cameron92]. No comprehensive description of the static table approach has been obtained, but a very general outline can be found in [Koenig90].

6.4.1 Overview of the dynamic registration approach

When the *dynamic registration approach* is used the fundamental component of the EH mechanism is a dynamic exception information structure that is continuously updated while the program is running in its normal state. In the dynamic structure, which is normally a stack of information-markers, it is registered when a scope which includes exception handler(s) is entered or left, whenever objects are constructed/destroyed, and when a function that includes a raises-set is entered or left.

When an exception is raised the EH mechanism backtracks over the dynamic structure using its previously registered information to locate the correct exception handler, all the while appropriately destroying objects and checking the legality of the propagation of the specific exception.

The *dynamic registration approach* is attractive in the aspect that it has little need for close co-operation between the EH mechanism and the target system and therefore is very portable. Significantly, EH using the *dynamic registration approach* can be implemented in a high-level language[9] as demonstrated by the *C* implementations discussed by [Cameron92] and [Koenig90].

The drawback for the *dynamic registration approach* is that it, due to the continuing updates of the exception information structure, introduces an ever-present overhead in the normal program flow! An overhead which is much aggravated by the C++ object model's requirement that objects must be cleaned up in the event of an exception.

---

[9] Providing support for capturing and setting the state of the program counter and stack register(s) exist as provided by C with the *setjmp* and *farjmp* functions respectivly.

On the other hand, but less important, the *dynamic registration approach* offers optimal performance for the actual handling of an exception.-Optimal because only the information registered in the dynamic exception structure needs to be examined which, in opposition to the static table approach, makes this approach independent of the distance between the raiser and the exception handler.

Where EH is a part of the operating system (OS) a *dynamic registration approach* is always used[10]. OS calls are used to continuously inform the EH mechanism about changing exception handlers and cleanup actions. Note that in comparison with a compiler specific implementation, the use of the OS will impose an additional overhead due to the operating system calls.

6.4.2 Overview of the static table approach

When an ideal *static table approach* is used the EH mechanism relies completely on static table(s) build by the compiler. The tables describe guarded regions of code and associated exception handler(s), raises-sets, and destructible object(s) and where they are constructed and destroyed.

When an exception is raised the EH mechanism uses the program counter (PC) to search the table for information about the region of code in which the exception has been signaled, including information about exception handlers, objects to be destroyed and raises-sets. When an exception needs to be propagated the mechanism “hijacks” the return address from the stack, unwinds the activation record from the stack and then restarts the process by using the hijacked address as a new PC value.

Characteristically the ideal *static table approach* imposes absolutely no run-time overhead in the normal program flow - all the work is done when an exception is actually raised!

The major shortcoming of the *static table approach* is that it needs close co-operation between the EH mechanism and the target system thus requiring a native, non-portable implementation. Also the *static table approach* requires of the target system’s architecture the possibility to capture the return address and manipulate the stack in the above described manner during unwinding.

Furthermore if exceptions are to be used in mixed-language programs using a method based on an ideal *static table approach*, a consistent well-defined call-frame structure is required of the processor architecture to support the stack-unwinding.

Raising an exception with the *static table approach* is relatively more expensive then with the *dynamic registration approach,* since the entire work is done when an exception is raised. Specifically, the cost of raising an exception is proportional to the number of subroutine calls between the exception handler and the raiser of the exception. However the increased cost of raising an exception is not a serious disadvantage if EH is used only to handle errors, which can be presumed to happen infrequently.

Note that the *static table approach,* although still technically possible, may not be a true option for a compiler for an operating system (OS) that integrates support for EH using a *dynamic registration approach*. An OS supported static table approach to EH, although complicated, should in theory be possible but to the knowledge of the author such an operating system has yet to be contrived.

[10] At least this is the case for the operating systems offering support for exception handling which is known to the author.

6.4.3 The approaches in comparison

A comparison of the advantages and disadvantages in the use of an ideal *dynamic registration approach* versus a *static table approach* for the implementation of EH is presented in figure 3.

	Dynamic registration	**Static table**
Normal overhead	Significant	None
Exception raise cost	Relatively inexpensive	Relatively expensive
Portable	Yes	No
Operating system support	In some OS's	No
Mixed language support	Yes	Depends on architecture

Figure 3 Dynamic registration approach versus ideal static table approach.

A compiler implementation of a EH mechanism with a minimum of normal run-time overhead that supports mixed language programming has been targeted for this thesis. Clearly the dynamic registration approach is not the right choice when a minimum of normal run-time overhead is required. Also clearly is it that there is a problem with the static table approach concerning mixed language programs on some architectures.- A unified method based on a modification to the static table approach which provides support for mixed language program on all architectures while still providing a minimum of run-time overhead is presented in section 7.2.2.2.

7. Transfer of control

Control must be transferred from the signaller to the corresponding handler when an exception is raised. Since it is generally not possible to predict in advance which handler to transfer control to the EH mechanism must have static and dynamic information available at runtime.- Information that provides an opportunity to traverse the call-chain and retrieve information about relevant handlers (in addition to information about raises-sets and where there may be objects to destroy).

7.1 Transferring control using the dynamic registration approach

When the dynamic registration approach is used a list of dynamically updated information markers is applied at runtime to find the handler for an exception, to specify when an exception must be validated against a raises-set and to located regions of objects which must be destroyed. Three kinds of information markers exist (from [Cameron92]) :

- Try markers defining regions for which specific exception handlers exist.
- Raises-set specification markers limiting propagation.
- Cleanup markers recording where cleanup action must be taken.

Stored in each try marker is, in addition to references to static information about the type(s) and address of each handler, a state flag, a reference to an active exception, and the value of the stack pointer and frame pointer on entry to the try block (plus other registers that may be of interest). The stack and frame pointer values are later put to use if control is transferred to the handler. The state flag and the reference to the current exception are used when an exception is raised. The flag can be marked as *available, busy_doing_cleanup* or *busy_in_handler*. It is initialised to *available*.

Contrary to try markers, raises-set markers and cleanup markers consist solely of references to static data such as valid exceptions and cleanup region(s) respectively.

The various information markers are inserted and removed from the list at runtime during normal program execution and using a stack discipline. A try marker is, after being initialised with reference(s) to static information and the current stack pointer & frame pointer values, appended to the list when a try block is entered. The try marker is removed when the try block is left.

Correspondingly a raises-set specification marker is appended to the list when a function with a non-default raises-set is entered and removed when the function is left. A cleanup marker is appended to the list when a function or block[11] with destructible objects is entered and removed when the function or block is left (A single cleanup marker in combination with a block-counter can be used to for multiple blocks of destructible objects).

The list of information markers is put to use when an exception is raised in which case the actions below take place (in large from [Cameron92]). An algorithm is presented in section 7.1.2.

1. The list is searched, starting with the most recently appended marker, for a try marker with a matching exception handler. If a handler can be found the associated try marker, known as the destination marker, is marked as *busy_doing_cleanup.* If no handler can be found cleanup action for the entire program must be undertaken and the program must be terminated with a suitable error message (i.e. “unhandled exception ...”). The cleanup can be done using an internal marker to mark the end of the list or by using a terminate flag. The terminate flag and the *busy_doing_cleanup* markings are used to make sure that no unhandled exception is raised while performing object cleanup (if so an error is reported).

2. Each marker in the part of the list between the last appended marker and the destination marker is popped off the marker list and examined. Destructive actions are performed whenever a cleanup marker is found. The

[11] Objects can be declared for blocks of code for C/C++ languages, for an entire function for Pascal languages.

exception is validated against a raises-set whenever a specification marker is retrieved. If a try marker with a *busy_in_handler* marking is found it means that an exception handler is being terminated by an exception and that the associated exception object must be destroyed.

3. The destination try marker is no longer marked as *busy_doing_cleanup,* instead it is marked as *busy_in_handle*r. Before transferring control, the new values for the stack pointer and frame pointer is retrieved from the try marker. Control is then transferred to the exception handler in question.

In [Cameron92] all markers are not popped one at the time as it has been specified above. Instead they are removed by setting the list's head pointer to the destination try marker after cleanup. Presumably that strategy means that, without safeguards, there is a risk that objects may be destroyed more than one time if an exception is propagated out of a destructor invoked for cleanup.

7.1.1 Possible problems with strongly recursive functions

In non-recursive programs the number of entries in the list of information markers is normally not a problem. However the list may grow excessively for strongly recursive functions for which a try-, specification- and/or cleanup marker must be appended upon entry to the function. There seems to be no easy solution to this problem. Rather it must be anticipated by the implementation.

Interestingly this problem does not exist with the static table approach, leaving out the overhead in unwinding a potential large number of singular frames for recursive functions.

7.1.2 Algorithm for transfer of control

The overall algorithm for the transfer of control using the dynamic registration approach is presented in figure 4. The algorithm uses a linked list of markers that can be iterated upon and used like a stack. Markers are pushed on and popped from the head of the list and iterations are performed from the head towards the tail of the list. Generally the algorithm is based on the method discussed in [Cameron92]. The algorithm presupposes a standard (CISC) architecture with a stack and a frame pointer.

```
PROGRAM TransferOfControl_Algorithm;

TYPE
 STATE = (AVAILABLE, BUSY_DOING_CLEANUP, BUSY_IN_HANDLER);

 RAISE_MARKER = RECORD
                  state : STATE;
                  exception: EXCEPTION;
                  SP : POINTER TO STACK;
                  FP : POINTER TO STACK; (* If exist *)
                  ... (* Ref to Exception handlers info *)
                  prev : POINTER TO MARKER;
                END;

 CLEANUP_MARKER = RECORD
                    ... (* Ref to cleanup region(s) info *)
                    prev : POINTER TO MARKER;
                  END;

 SPECIFICATION_MARKER = RECORD
                          ... (* Ref to raises-set info *)
                          prev : POINTER TO MARKER;
                        END;

 MARKER = RAISE_MARKER OR CLEANUP_MARKER OR SPECIFICATION_MARKER;
```

```
VAR
 head : POINTER TO MARKER; (* Head of list of markers *)

PROCEDURE Raise(exception: EXCEPTION)

VAR
  match : POINTER TO TRY_MARKER;
  eh_addr : POINTER TO CODE;

BEGIN
 match := FindTryMarkerForMatchingExceptionHandler(exception);
 IF match THEN (* Exception handler found ? *)
   (* Enable detection of attempts to exit destructor invoked
      for cleanup by an exception: *)
   match^.state:=BUSY_DOING_CLEANUP;

   (* Pop cleanup markers from head to match and perform
      cleanup actions for objects *)
   Cleanup(match);

   (* Enable detection of exit of previous exception handler *)
   match^.state:=BUSY_IN_HANDLER;

   (* Store reference to current exception object so that it
      can be cleaned up later if its exception handler is
      terminated *)
   match^.exception:=exception;

   (* Perform context switch and jump to exception handler *)
   eh_addr:=GetPcAddrForMatchingHandler(match);
   SP:=match^.SP;
   FP:=match^.FP;
   JumpTo(eh_addr);
 ELSE (* No exception handler found *)
   (* Pop all cleanup markers and perform cleanup actions *)
   Cleanup(0);
   Error("Unhandled exception");
 ENDIF;
END; (* Raise *)

PROCEDURE Cleanup(end: POINTER TO MARKER);

VAR
 marker: POINTER TO MARKER;

BEGIN (* Always start from head of list *)
 IF head THEN (* list not empty ? *)
  WHILE head<>end DO BEGIN
   marker:=head;
   head:=head^.prev; (* pop marker *)
   IF marker points to a CLEANUP_MARKER THEN
     PerformCleanActionsFor(marker); (* Destroy objects *)
   ELSE IF marker points to a SPECIFICATION_MARKER THEN
     (* Check and inforce raises-set *)
     IF NOT PropagationAllowed(marker, exception) THEN
       Error("Propagtion of exception not allowed");
     ENDIF;
   ELSE IF marker points to a TRY_MARKER THEN
     IF marker^.state = BUSY_IN_HANDLER THEN
       (* Terminat exception handler for a previous exception *)
       DestroyExceptionObject(marker^.exception);
     ELSE IF marker^.state = BUSY_DOING_CLEANUP THEN
       (* Attempt to exit cleanup by an exception - Error *)
       Error("Illegal exception in destructor invoked for cleanup *)
     ENDIF;
   ENDIF;
  END; (* WHILE *)
 ENDIF;
END; (* Cleanup *)
```

Figure 4 Transfer of control algorithm for dynamic registration approach

7.2 Transferring control using the static table approach

With the static table approach a program counter (PC) value in conjunction with pre-build static table(s) is used to find the handler for an exception, to specify when the exception must be validated against a raises-set and to locate objects which must be destroyed. When no handler can be found for the exception it propagates to the calling function by “hijacking” the return PC value from the stack. The stack is then unwounded and the hijacked PC value is used for future transfer of control.

Using the static tables a mapping from a PC counter value into information about exception handlers, raise-sets and destructible objects can be made.

One organisation of the information in the static tables is to have one main static table of PC ranges for the functions in a program and to have the main table refer to additional specific tables for those functions that require it. As standard information for each PC range counts at least an associated pointer that, for non-trivial functions (with respect to destructible objects, raises-sets and exception handlers), will point to an additional table of information for the individual function.

Because the main program table can be rather large the lookup must be efficient. By having the compiler emit equally sized PC-range entries for functions in the order of their definition, binary search is achievable rather easy and without an initialisation cost, provided the program as whole is statically linked and that adequate linker support exists. These advantages are unmatched by other more efficient representations of the program table.

When employing a sorted program table an optimisation may be applied in which the end address for each function is not actually included in the table entry. Instead it can be deduced from the start address of the function in the subsequent entry.

To ensure that a correct end address can be obtained from the subsequent entry, each module must include a dummy entry as its last entry in the main program table. Without such a dummy entry a gap in the address range between two functions in two compiler created modules would be mistaken as belonging to code for the last function in the module before the gap.

The individual function information tables specify guarded[12] regions of code and their exception handlers, cleanup regions and raises-sets. Guarded regions for exception handlers can be specified by internal PC ranges or by a list of PC-values for guarded points such as call-sites and raise-points.

7.2.1 Propagating an exception

To support the retrieval of the return address and the unwinding of the stack needed when propagating an exception the following information must be available for each function:

- The location of the return address in the function’s activation record.
- Unwind information about the callers stack pointer and frame pointer[13].

The issue of the location of the return address and the unwinding of the stack as well as how much and what information must be provided depends on the system’s architecture and what code is generated to set up activation records.

[12] Guarded regions of code are contained by TRY-EXCEPT & TRY-FINALLY constructions.
[13] Providing a frame pointer is used in the system architecture.

7.2.2 Unwinding and retrieving return address on CISC architectures

Provided a uniform call frame scheme is used for each and every function generated by the compiler, the return address can be obtained via the permanent entry in the activation record.

On some CISC architectures a frame pointer (FP) is used to access entries in the function's activation record. When a frame pointer is used the return address is easily obtained at a fixed offset from FP (See figure 5), otherwise a more elaborate scheme must be used in which the return address is obtained at an inconstant offset from SP for different PC values (See section 7.2.2.1).

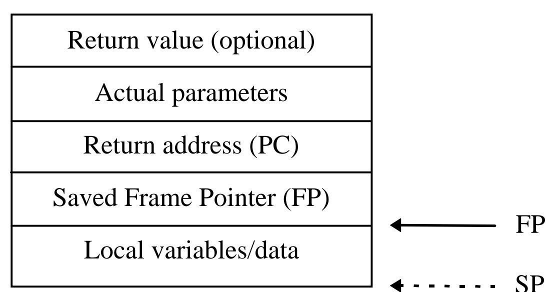


Figure 5 Typical activation record

The amount of data above the saved FP entry that must be unwounded from the stack, depends on the actual parameters and on whether a stack-based return value is used. Hence to assist the unwinding static data must be generated for each function, containing a size entry designating the number of bytes to unwind.

It would be obvious to have the static size entry for each function be equal to the size of the formal parameters + the size of the return value entry (if any). However such a scheme will at times result in initial arguments not being properly unwound for function calls which have function calls as arguments. Also the scheme can't be used for languages such a C where functions with a variable number of arguments are allowed.

A general solution to the unwind problem is to have the size entry equal to the size of the local variables/data. When a frame pointer is employed the size entry is used as follows: The static size value for the calling function is used to specify the value of the calling function's stack pointer with respect to its frame pointer. Consequently, after having retrieved the callee's return address, and through that a key to the static information for the calling function, and after having restored FP, SP can be set to FP +/- the size of the calling function's local variables/data.

7.2.2.1 Complications where a frame pointer is not used.

A frame pointer scheme may be either unavailable in general or be optimised away for functions with no arguments and no local environment. Where a frame pointer is missing, additional static information about the location of the return address on the stack and the number of bytes to unwind must be available.

Sometimes the number of bytes to unwind depend on the raise point or point of propagation (call points). In these cases the static information must be available for each such point. This may mean a significant increase in storage costs of the static table(s).

7.2.2.2 *Problems with mixed language support*

The crux of the matter in the static table approach is the hijacking of the return address + the stack unwinding. The problem is that these actions are not possible to perform in a mixed-language environment without a strict machine architecture for call-frames (also called self-describing stacks).

That CISC architectures does not have a strict definition of call-frames is normal. It is the case for widespread CISC architectures such as the Intel 80x86 and the Motorola 68000. To simply promote a standard call frame is not enough. On the Intel 80286+ a standard call frame can be established by the specialised ENTER and LEAVE instructions. However their use is not compulsory which means that the call frames can't be considered as strictly defined.

Provided the architecture allows you the basic stack operations needed and if the right static information about the activation record is available, it is always possible for functions generated by the compiler to propagate by hijacking the return address and unwinding the stack.

However the propagation may fail where a call chain is a mixture of function calls to functions generated by the compiler and of calls to external functions (possibly written in C or Assembler). The external functions have no knowledge of exceptions, and are in general not able to propagate exceptions since they conform to no requirements about call frames and have no associated static information for the EH mechanism.

The failure of propagation occurs when a function A with an exception handler calls an external function B, which in turn calls a function C that raises, but does not handle, an exception meant to be handled by the function A. The EH mechanism will lose tracks trying to propagate from the external function B to function A, leaving out the special case that a frame pointer is used and the call frame for B is the same as the one universally used by the compiler. To solve the problem compromises must be adopted. A purely static table approach with no overhead is no longer possible.

7.2.2.3 *The unified method*

The problem for the static table approach with mixed-language programming on architectures without strict call frames can be rectified by applying the following solution which will be called the unified method:

- Each function called which is not compiled using the compiler must be declared as an *external* function (*external* meaning external to the language and not only to the containing module).
- Each external function called from a compiler generated function is called through a small code stub. Before the actual call the stub pushes, and after the call it pops, a location marker on a location information stack provided by the EH mechanism and used for only that purpose.

A location marker contains the PC location at/after the call of the external function and the stack and frame pointer values at the time of the call. If during propagation an unknown PC-address is found, that is the exception has propagated into an external function, a location-marker is popped of the location marker stack and the system's stack pointer and frame pointer is restored. At that point the search for a handler for the exception can continue from the stored PC-address value.

The unified method can be optimised if it is made possible for the user to specify raise-sets for external functions, or if it at least is possible to specify that for specific external functions no exception can propagate. Where no exception can propagate from an external function, the code stub and the use of a location-maker are no longer necessary.

7.2.3 Unwinding and retrieving return address on RISC architectures

If a register window scheme is used as in the SPARC RISC Architecture then static information to facilitate propagation is normally unnecessary. The return address can easily be retrieved from the designated register I7 in the register window. Just as easily the unwinding can be done by adjusting the current window pointer (CWP) back one level by issuing a RESTORE instruction, and utilising the callers stack/frame-pointer SP/FP, in the register window. Note that although static information is generally not required, a modest amount of static information may be necessary if deviations of the normal window scheme are allowed, such as a special optimisation for leaf procedures[14].

The overlapping register window scheme used in the SPARC RISC Architecture is an example of an architecture with strictly specified call-frames. The strict call-frames implies that given any function which raises an exception it is always possible to walk the chain of calls in search for an exception handler. Consequently potential problems in mixed-language scenarios are not an issue here.

7.2.4 Algorithm for transfer of control.

The basic algorithm for the transfer of control using the unified method is presented in figure 6. The algorithm presupposes a standard (CISC) architecture with a stack and a frame pointer and with a stack that grows downwards (SP decreases when data is pushed).

```
PROGRAM TransferOfControl_Algorithm;

TYPE
 PROGRAM_TABLE_ENTRY = RECORD
                        begin_range: POINTER TO CODE;
                        info_ptr:POINTER TO FUNCTION_TABLE;
                       END;

 LOCATION_MARKER = RECORD
                    call_point: POINTER TO CODE;
                    saved_fp: POINTER TO STACK;
                   END;

PROCEDURE Raise(exception: EXCEPTION);

VAR
 pc, eh_addr : POINTER TO CODE;
 fp: POINTER TO STACK;
 sp: POINTER TO STACK;
 ptp: POINTER TO PROGRAM_TABLE_ENTRY;
 lmp: POINTER TO LOCATION_MARKER;
 ftp: POINTER; (* TO FUNCTION_TABLE *);
 ehp: POINTER; (* TO EXCEPTION_HANDLER_INFO *);

BEGIN
 pc := GetCallPointAddr();
 fp := GetCallersFramePointer();

 REPEAT
  ptp:=FindFunction(pc); (* Find function entry in main table *)
  IF ptp THEN (* Function found in main program table *)
    ftp:=ptp.info_ptr;
    (* Use FP value + size of local data to get SP value *)
    sp:=fp - SizeOfLocalData(ftp); (* + if stack grows upwards *)

    (* Look for an exception handler for the current frame *)
    ehp:=FindHandler(exception, ftp, pc);
```

[14] A leaf proedure is a leaf in a call graph (i.e. it does not call other procedures).

```
    (* Destroy exception objects for previous invocations of
       Raise found on the EH mechanism's stack which has pc
       and fp belonging to the current function frame but with
       ehp pointing to an exception handler which will be out of
       scope *)
    TerminatePreviousExceptions(ehp, fp, pc);

    IF (NOT ehp) THEN (* Exception handler found ? *)
      (* Exception handler not found - Cleanup objects *)
      PerformCleanupActions(ftp, pc, sp, fp);
      (* Check for end of program *)
      IF TerminationAddressReached(pc) THEN
        Error("Unhandled exception");
      ENDIF;
      (* Check and inforce raises-set *)
      IF NOT PropagationAllowed(exception, ftp) THEN
        Error("Propagtion of exception not allowed");
      ENDIF;
      pc:=HijackReturnPCAndUnwind(fp); (* Propagate exception *)
    ENDIF;
  ELSE  (* External function *)
    lmp:=PopLocationMarker();
    IF lmp THEN (* Location marker popped ? *)
      pc:=lmp^.call_point; (* Continue propagation from *)
      fp:=imp^.saved_fp;   (* saved location *)
    ELSE (* No location marker found - Error *)
      Error("Exception propagated to unknown address");
    ENDIF;
  ENDIF;
 UNTIL ehp;

 (* Perform context switch and jump to exception handler *)
 eh_addr:= GetPcAddrForMatchingHandler(ehp);
 SP:=sp;
 FP:=fp;
 JumpTo(eh_addr);
END; (* RAISE *)
```

Figure 6 Transfer of control algorithm for static table approach using the unified method

8. Object Cleanup

All aspects of the compiler including the EH mechanism must ensure that all objects that have been successfully constructed are properly destroyed when they are out of scope. For the EH mechanism the different cases in which destructors must be invoked are listed below:

- Local object variables, when propagating out of a function.
- Global object variables, when no handler can be located.
- Sub-objects[15], when an object has been only partially constructed.
- An exception object, when the handling of the exception is finished.

The following sections will present various specialised methods that enable the EH mechanism to properly destroy objects in an efficient way. Note that for local, global and sub-objects it is possible to translate the object cleanup problem into implicit TRY-FINALLY constructions (nested, one for each destructible object). Unfortunately this approach is not very efficient, which is especially important for objects because they are universally used.

8.1 Requirements for destroying local, global and sub-objects

The main problem when destroying either local, global or sub-objects is to decide which destructors for objects to call given an exception point and a list of destructible objects in the current context (function, main procedure or constructor/destructor respectively).

The example program extract in figure 7 illustrates the main issues of the destruction of objects. The example only covers local objects and sub-objects, but it corresponds equally to global objects.

In the example an exception can be raised at seven points, numbered after execution order. Note that objects are destroyed in reverse order of the construction order.

```
PROGRAM cleanup_example;

TYPE
  A = OBJECT
       CONSTRUCTOR init(); FORWARD;
       DESTRUCTOR done(); FORWARD;
      END;

  B = OBJECT
       CONSTRUCTOR init(); FORWARD;
       DESTRUCTOR done(); FORWARD;
      END;

  C = OBJECT (B)
       CONSTRUCTOR init(); FORWARD;
       DESTRUCTOR done(); FORWARD;
      END;

VAR
  I : INTEGER;

CONSTRUCTOR A.init();
BEGIN
 IF I=1 THEN RAISE Exception; ENDIF;
END;
```

[15] A sub-object is an ancestor object or a member object

```
CONSTRUCTOR B.init();
BEGIN
 IF I=2 THEN RAISE Exception; ENDIF;
END;

CONSTRUCTOR C.init();
BEGIN
 IF I=3 THEN RAISE Exception; ENDIF;
END;

DESTRUCTOR A.done();
BEGIN
 IF I=7 THEN RAISE Exception; ENDIF;
END;

DESTRUCTOR B.done();
BEGIN
 IF I=6 THEN RAISE Exception; ENDIF;
END;

DESTRUCTOR C.done();
BEGIN
 IF I=5 THEN RAISE Exception; ENDIF;
END;

PROCEDURE P();

VAR
 a: A;
 c: C;

BEGIN
 IF I=4 THEN RAISE Exception; ENDIF;
END;
```

Figure 7 Cleanup example

The relationships between raise points and objects to be destroyed are shown in figure 8 (for an explanation of the relative destructor count refer to section 8.3).

Raise point	Destructor counter value relative to start of procedure P.	Objects to destroy
1	0	
2	1	a
3	2	a, object B part of c
4	3	a, c
5	2	a, object B part of c
6	1	a
7	0	

Figure 8 Objects to be destroyed by EH mechanism at different raise-points.

8.2 Destroying exception objects

Exception objects are not include in the table discussed in the previous section since they can be explicitly destroyed at the end of each exception handler or by the EH mechanism if the exception handler is itself being terminated by an exception.

Apart from that, if an exception is raised during the construction or destruction of an exception object, the sub-objects within the exception object can be destroyed in the same way as ordinary objects.

The destruction of an exception object is trivial if an exception handler is terminated in a normal way. When an exception handler is terminated by an exception it can be difficult to ensure cleanup of the relevant exception object because is must be detected when (and exactly when) the scope that contains the exception handler is exited.

How it can be detected when a previous exception handler is exited with the dynamic registration approach is covered in section 7.1. With the static table approach it can be done during propagation by continuously comparing function scopes of the active exception and handlers for previous exceptions. When a match of the dynamic scope's of the functions has been detected the previous exception object must be destroyed if :

- No handler can be found for the active exception in the function, or
- A handler for the active exception, with a nest level <= nest level for previous exception's handler, has been found in the function.

Note that to discover a match in the scope's of two functions it is not adequate to compare the PC-address's of the functions (because of recursive functions etc.). Rather the values of the frame pointers (or initial stack pointers) for the functions must be compared by the EH mechanism, they should be equal if the function instances are equal.

8.3 Determining which objects to destroy with the dynamic registration approach

The most simple, but least efficient, method based on dynamic registration which can be used to find out which objects to destroy, is to let all objects which must be destroyed form a linked list [Koenig90]. The list is only used in the event of an unhandled exception. Each object is inserted to and removed from the linked list by the constructors and destructors respectively.

The disadvantage of the linked list scheme is that it takes up too much memory and is too costly in runtime overhead. In the rest of this section a space saving and more efficient method will be discussed which merely uses a global counter to count the objects which must be destroyed.

An efficient method is to use a global *destructor counter* to play the key role in determining which objects have been constructed and not yet destroyed = which objects are to be destroyed by the EH mechanism in the event of an unhandled exception.

The destructor counter is increased by each constructor and decreased by each destructor. Example pseudo code for a constructor/destructor pair can be observed for the object C in figure 9 and figure 10. Recall from figure 7 that C has a single sub-object B.

```
CONSTRUCTOR C.init();
BEGIN
  B.init(); (* Construct sub-object B *)
  ... user code ..
  destructor_counter:=destructor_counter+1;
END;
```

Figure 9 Modifications of destructor counter in C's constructor

```
DESTRUCTOR C.done();
BEGIN
  destructor_counter:=destructor_counter-1;
  ... user code ...
  B.done(); (* Destroy sub-object B *)
END;
```

Figure 10 Modifications of destructor counter in C's destructor :

For the general destructor counter strategy to work the compiler must ensure that a constructor/destructor pair exists for every object for which some destructible action must be taken (that is if the object has an ancestor with a destructor or member objects with destructors). If the constructor or destructor is missing the compiler must supply one.

The original value of the destructor counter is saved upon entry for each function which is not a constructor or destructor and which may require cleanup. The original value makes it possible to calculate the *relative destructor counter* value for the current context.

The *destruct count value* is a statically computable value for a destructible type that is equal to the total number of objects to be destroyed for the type itself to be fully constructed. The destruct count can be computed to: n * (1 + the sum of the destruct count values for member objects (if any) + the destruct count value for an ancestor (if any)). Where n is 1 for objects and equal to the number of elements for array of objects. For simple object types this value is one, for object C shown in the previous examples this value is 2. The EH mechanism must have access to the destruct count value for each destructible type, preferable by having the value calculated at compile time for each type in question and having it saved in a static table.

Given the relative destructor counter value at the time of the exception and the various destruct count values of the objects which are candidates for destruction, a list of objects which must be destroyed can always be determined. A simple algorithm for determining the objects to be destroyed is presented in figure 11.

```
PROCEDURE destroy_objects()

BEGIN
 dc := destructor counter - original destructor counter
 (* dc := relative destructor counter *)
 o := first declared destructible type
 WHILE dc>0 DO
 BEGIN
   IF dc >= o’s destruct count THEN
     Include o in list of destructible types to be destroyed.
     dc:=dc-o’s destruct count.
     o:=next declared destructible type.
   ELSE
     (* o is an array of destructible objects, of which only
        a part must be destroyed. *)
     Include the first {dc / destruct count of object element}
     elements in the list of destructible types to be destroyed.
     dc:=dc MOD destruct count of object element;
   ENDIF;
 END;
 Destroy all listed types one at the time, in reverse order.
END;
```

Figure 11 Determining which objects to destroy using a destructor counter

8.3.1 Deallocating memory for a partial constructed heap object

The problem of deallocating memory for a partial constructed heap object is for the EH mechanism to detect when it has to do with a constructor for a heap object (It is assumed that the calls to the heap management routines are normally performed by the object’s constructor and destructor respectively).

To enable detection of the termination of a constructor for a partial constructed heap object a cleanup region for each construction point of a heap object can be created and managed. - In the same way that multiple cleanup regions can be used to represent multiple blocks of variables. ([Cameron92]).

8.4 Determining which objects to destroy with the static table approach

When a static table approach is used the EH mechanism must use PC counter values to determine which objects to destroy in case of an unhandled exception. To enable this the compiler must output information in static tables that maps PC counter values into the information necessary to perform cleanup decisions.

8.4.1 Local and global objects

Typically the code generated for each function is divided into consecutive regions describing construction and destruction states of objects. Similar regions exist for global object variables which are constructed and destroyed just like local objects in specialised initialisation and deinitialisation functions. The order of the regions in which objects are constructed and destroyed is symmetric.

The construction and destruction of objects are done implicitly be the compiler. For the procedure P in figure 7 the implicit calls of constructors and destructors are shown in figure 12. The corresponding static table information for P, used to provide a mapping from program states to cleanup action, is shown in figure 13.

```
PROCEDURE P();

VAR
 a: A;
 c: C;

BEGIN
 a.Init(); # Region 1
 c.Init(); # Region 2
 IF I=4 THEN RAISE Exception; ENDIF; #Region 3
 c.Done(); # Region 4
 a.Done(); # Region 5
END;
```

Figure 12 Procedure P with implicit calls of constructors and destructors shown

Region addr range	Description	Cleanup action
range 1	a being constructed	
range 2	c being constructed	destroy a
range 3	user code	destroy a,c
range 4	c being destroyed	destroy a
range 5	a being destroyed	

Figure 13 Mapping from PC ranges to cleanup actions for P.

For a PC range in which a constructor is being constructed or destroyed only the PC-location (at/after) the actual call to the constructor/destructor is of interest. Consequently it is not necessary for the static information to record the PC range for each object constructed or destroyed, rather the PC value of the call-point for each constructor or destructor can be stored.

8.4.2 Partially constructed objects

In the same way that PC ranges can be used to determine what objects to destroy, PC ranges can be used to figure out what sub-objects to destroy in a partially constructed object by mapping PC ranges in constructors and destructors into cleanup information. For the object C from the example in figure 7 only one sub-object (the

ancestor) is constructed. In figure 14 - figure 15 the implicit calls of constructors and destructors are shown for object C. In figure 16- figure 17 the corresponding static table information is presented.

```
CONSTRUCTOR C.init();
BEGIN
 B.init();          # Region 1
 ... user code ..   # Region 2
END;
```

Figure 14 Constructor for object C with implicit calls of constructors shown

```
DESTRUCTOR C.done();
BEGIN
 ... user code ...   # Region 1
 B.done();           # Region 2
END;
```

Figure 15 Destructor for object C with implicit calls of destructors shown

Regions addr range	Cleanup action
1	None
2	Destroy sub-object B

Figure 16 Mapping from PC ranges to cleanup actions for C.Init()

Regions addr range	Cleanup action
1	Destroy sub-object B
2	None

Figure 17 Mapping from PC ranges to cleanup actions for C.Done()

8.4.3 Arrays of objects

When an unhandled exception occurs during construction of an array of objects the PC value is not sufficient to determine how many elements that have been constructed[16].

Generally the array is constructed and destroyed iterativly by two small runtime library routines that apply the constructor or destructor respectively on each element of the array. In case of an unhandled exception involving an array the PC value will point into one of these runtime routines. Knowing that the exception occurred in one of these helper routines the EH mechanism must be able to retrieve information about the elements that was constructed from the local environment of the helper routine. To enable this the internal workings, as well as the address spans of these array helper routines in the runtime library, must be known to the EH mechanism.

8.4.4 Deallocating memory for a partial constructed heap object

Generally an exception in the construction of a heap object can be detected by having the compiler append a *New* region for each *New* statement issued in the static information for the contained procedure. When the EH mechanism propagates it checks whether the PC-value belongs to a *New* region. - If it does then the memory is deallocated.

[16] Unless the array is constructed in a non-iterative fashion which is unlikely for all but the smallest arrays.

Providing it is possible to find our whether or not an address belongs to memory that was allocated by *New* an alternative is possible that avoids the addition of *New* regions at the cost of 1 bit per non-trivial procedure/function. (Again, it is assumed that the calls to the heap management routines are normally performed by the object's constructor and destructor respectively).

In the alternative approach the memory for an object can be deallocated directly if an unhandled exception occurs in a constructor for an object which is not implicitly constructed and whose address is allocated by *New*. Note that this alternative means that the static information for each non-trivial procedure must be specified whether a procedure is a constructor or not (using one bit of information).

A check for an object not being implicitly constructed is necessary to avoid having the same memory (with an optional offset) being deallocated more than one time. The check can be performed by examining the PC at the point where the constructor for the object was called. If the PC is a part of one of the object construction regions laid down for each implicit constructed object then the test has failed and the memory should not be deallocated.

8.5 Destroying objects

Having determined which objects to destroy, static information is needed in order to enable the actual destruction. Regardless of which approach is used the EH mechanism must have access to an ordered list of static information for each variable which is a destructible object or which is an array of destructible objects. The information needed for each type is in general:

- The address of the object/array of objects.
- The address of the default destructor to be applied.
- Is it an ordinary object or an array of objects ?
- The *destruct count* for the object/array of objects (for registration approach only).

If the type is a destructible array the following additional information is needed:

- The element size of the array.
- The element size.

The address of an object/array of objects can be either relative with respect to the beginning of the stack frame for locally scoped variables, relative to start of containing object for sub-objects or absolute for global variables.

For the example in figure 7 the static information is as indicated in figure 18.

Variables	**Var addr**	**Dtr addr**	**Array ? , Array size**	**Destruct count (optional)**
a	@a	@A.done()	Not array	1
c	@c	@C.done()	Not array	2

Figure 18 Static information for procedure P.

In addition to the static information for variables the EH runtime mechanism must also have access to type specific information about sub-objects and their relative addresses in case a partial constructed object must be destroyed. For ancestor objects the address of the object is always the same as that of the descending type, for member types the relative address is equal to the size of an optional ancestor + the size of any previous member objects. The address for each sub-object can be precalculated and stored in the type information table or computed at runtime. An example of a compact table of cleanup specific type information is shown in figure 19.

Types	**Ancestor type**	**Member types**	**Size**
A	No ancestor	None	SIZEOF(A)
B	No ancestor	None	SIZEOF(B)
C	B	None	SIZEOF(C)

Figure 19 Runtime type information for cleanup for program object_cleanup

The cleanup oriented type information must either be associated with each object and therefore made accessible through the object instances (in the same a manner as discussed for exception objects in section 9.2.1) or the type information must be placed together with the static information for variables.

9. Exception Identification

Runtime exception identification enables the EH mechanism to compare different types of exceptions. The requirements of and methods for an exception identification facility are presented in the following two sections.

9.1 Requirements for exception identification

Given a one or more exception types it must be possible to check whether an active exception has a matching type. Such checks are repeatedly necessary when locating an exception handler for an exception and when validating an exception against a raises-set specification for a function.

When exceptions are objects the exception identification amounts to the runtime type equivalence problem of deciding whether an object type from a handler or a raises-set is a supertype for the object type of the active exception.

9.2 Methods for Exception identification

The type equivalents problem is similar to the pre-ISO/ANSI C++ problem of performing runtime type inquiries, where statically compiled type information must be used (See [Stroustrup91]).

Static type information describes the object and its ancestors. Each time an object is constructed the corresponding static type information most be associated with the object instance in order to make it possible to make type inquiries later.

9.2.1 Associating type information with objects

Below four simple methods of associating type information with an object is summarised:

1. Including type information in each instance of an object, either indirectly or (if the size of the type information is very small) directly.
2. Provide a virtual method for each object type that returns the type information.
3. Embedding a data pointer to type information in the object type's virtual method table.
4. Insert type information just before the virtual method table (VMT) for each object type.

Method 1 is very portable. It is the only solution if no virtual method table is available. However if a virtual method table is available this method should be discarded because of the unnecessary space overhead involved.

Method 2 is also very portable. It can be used if a virtual method table is available. It is however not very efficient for a native implementation.

Method 3 requires that the size of a data pointer is less then or equal to the size of a code pointer. In comparison to method 4 it uses a little more space and the type information is slightly more time-consuming to inspect.

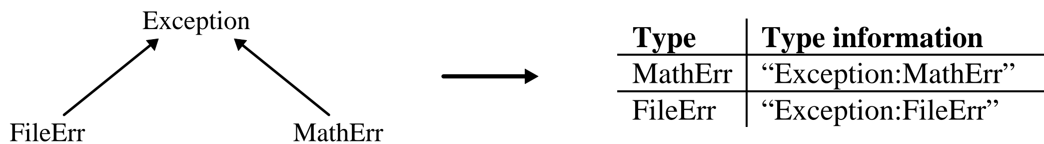
For a native implementation, where a VMT is available for all exception types and referred to using a VMT pointer located at a fixed offset in each object, method 4 is the best solution. The space overhead is of minimum size, one piece of type information pr. exception object type and no extra pointer. Also, as for method 3, there is no runtime overhead for the constructor since the object instance's VMT pointer is initialised anyway. Note that

because the size of the VMT may vary the type information can only be located before and not after the virtual method table.

9.2.2 Type information description

As mentioned above the type information associated with each exception object instance must include the type of the exception object and the type of all its ancestors.

The simplest and most portable way of representing the full types is by using an string representation as suggested by [Stroustrup91], [Koenig90] and [Lajoie94]. An example is shown in figure 20. Using this representation it is trivial to decide whether two types are the same or to see if one class is a superclass of another.



Type	Type information
MathErr	"Exception:MathErr"
FileErr	"Exception:FileErr"

Figure 20 String representation of type information

Note that in the string type representation presented in figure 20 it is assumed that the types are of global linkage (exportable). If the exception type is non-exportable there may exist exception types in other modules which may accidentally have the same name. This imposes a problem since an exception may legally propagate out of the scope in which the type is declared[17]. To correctly handle the case where two identically named exception types may not be of the same type, a module name or equivalent information can be added to the string representation of the type.

The drawback of using a string representation of type information is that it yields poor performance when comparing types and that the type information may take up a lot of space.

If it can be made certain that only one instance of the type information is present at all times, that is if all exception objects of a specific type across all modules refer to the same instance of type information, the following method inspired by [Cameron92] can be applied.

The type information is divided into two fields: A ancestor type information pointer and an identification flag. The ancestor type information pointer field points to type information for the type's immediate ancestor. Thus a full set of type information for a group of objects will usually be organised as a tree. The *id* field in the type information is used as a scratchpad field and must be initialised to zero. An example of such an organisation is presented in figure 21.

[17] Surprisingly this problem is not mentioned in any of the references.- It would be interesting to check existing C++ compilers against this possible bug.

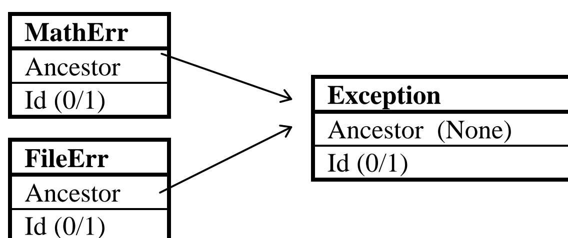


Figure 21 Ancestor/id type information

Because only one instance of the type information exists for each type of exception, two exceptions are of the exact same type if the addresses of their corresponding type information are equal. Similarly an exception object A is a superclass for an exception object B if somewhere in the linked list pointer to by the ancestor field in the type information for B, the type information for object A is pointed to.

In most cases where type information is used in the EH mechanism, one object B (the active exception object) must be checked against a wide range of other objects from handlers and raises-sets. Therefore, rather then performing the time-consuming task of checking the address of type information against the addresses for each exception object in question, a more efficient approach is used involving the *id* flag.

The algorithm using the *id* field is as follows (from [Cameron92]):

1. Mark all *id* fields in the exception object type B's type information list.
2. For each object match candidate, check the corresponding type information's *id* field. - If it is marked then the object is a superclass of the exception object type B.
3. Clear all *id* fields in the exception object type B's type information list.

Note that all *id* fields must be cleared whenever user-code is executed, to avoid having exceptions being thrown while some exception objects are still marked.

10. Storage management

The EH storage manager controls the allocation and deallocation of memory for exception objects. The requirements and design of such a manager are presented in the following sections.

10.1 Requirements of the exception storage manager

Dynamically storage must be set aside to hold the exception object when an exception is raised. The storage must be retained until the exception has been handled and it must be located globally since the exception object must be accessible throughout the propagation and handling of the exception.

If there was only need for temporary storage for one exception object at all times, storage could be allocated statically according to the size of the largest exception object being handled. Unfortunately this is not the case because nested exceptions are allowed.

In fact since nested exceptions are allowed, any number of exception objects can be active and accessible at the same time. This is demonstrated in the example in figure 22 where both the *MathErr* and *FileErr* exception objects must be accessible for the function *do_something* :

```
TRY
 ...
EXCEPT
 | MathErr(a) => ...
                 TRY
                  ...
                 EXCEPT
                  | FileErr(b) => do_something(a,b);
                 END;
                 ...
 ...
END;
```

Figure 22 Nested exception example

Storage for the exception objects is mostly but not always allocated/deallocated in a first allocated, last deallocated fashion. However when an exception is raised while in an exception handler the storage for the last but one exception object must be deallocated before the storage for the last exception object. Such a situation is shown in figure 23 where the storage for the first exception object MathErr must be deallocated when the second exception FileErr propagates out of the exception handler for MathErr:

```
TRY
 ...
EXCEPT
 | MathErr(a) => ... RAISE FileErr;
END;
```

Figure 23 Exiting an exception handler with an exception

10.2 Designing the exception storage manager

An ordinary heap management scheme such as a free list can be used to facilitate the storage management for the EH mechanism so one option is to simply use the general purpose free storage manager. This is however not advantageous because of the unnecessary overhead involved, since heap exhaustion is one perfectly reasonable candidate for the issue of an exception in which situation the use of the general storage is simply not possible,

and because it, as [Koenig90] states, is undesirable to make the EH mechanism exposed to exhaustion or corruption of the free store.

In [Cameron92] it is outlined how a separate scheme with less overhead (in space and time) than the general heap management approach can be applied. With this scheme the deallocation of storage for exception objects can be divided into two deallocation operations:

- An ordinary *deallocation operation* which frees the storage allocated for the topmost exception object.
- A *combine operation* which coalesces the storage for the topmost (last) exception object with the storage for the exception object immediately beneath. The actual deallocation of storage is postponed until the deallocation of the topmost exception object.

The deallocate/combine scheme presented above can be implemented using a stack of arbitrary sized storage blocks, with each storage block having a size entry.

11. Exception Handling and compiler issues

For a compiler several translation schemes exist for the TRY-EXCEPT and TRY-FINALLY constructions. They will be discussed in the following two sections. A discussion of the translation of raise statements follows in addition to a discussion of issues of general interest for a compiler having support for EH.

11.1 Translating the TRY-EXCEPT construction

figure 24 shows two translation schemes for the try-except construction. The scheme used in translation 1 is the easiest to implement, both for the compiler and for the EH mechanism. Nevertheless it imposes a small but unnecessary overhead in the normal execution (due to the jump instruction). The scheme used in translation 2 has no such overhead but is on the other hand the hardest to implement and it takes up more space for nested try-except constructions when a static table approach is used.

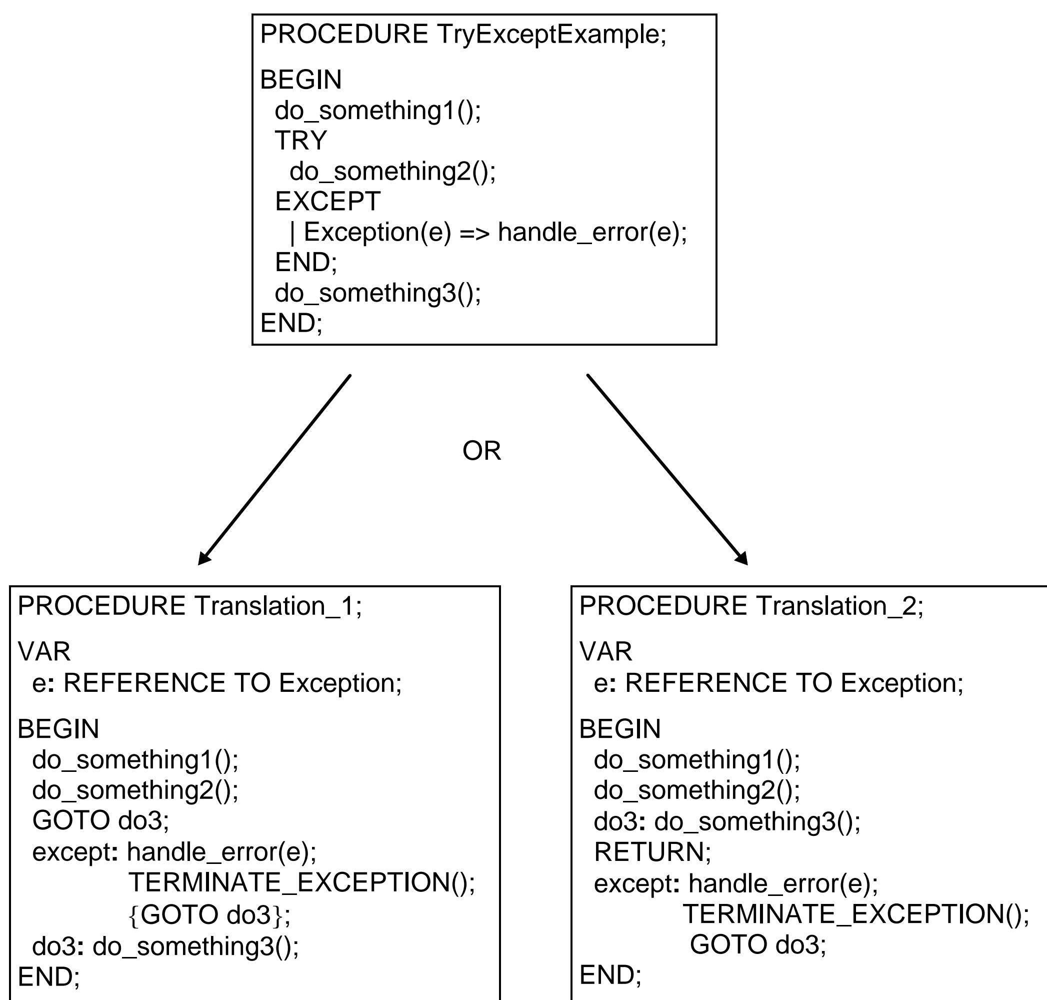


Figure 24 Translation schemes for TRY-EXCEPT construction

More to the point, with scheme 2 the static tables will be larger when a try-except construction is contained in the guard section of another try-except construction. -Larger, because two address ranges must be supplied for all but the outmost try-except construction, one for the try-except statements and one for the handler statements.

The translation examples in figure 24 shows how an exception handler's argument (**e** in the examples), can be mapped into a local *reference* variable. The optional argument must be useable just like a regular local variable (local because nested exceptions are possible) of the relevant exception object type. Since the address and exact type of the exception object are not known at compile time, the exception object address must be accessed through a hidden pointer, a reference.

The compiler must reserve space for the exception reference variable in the local environment for the function in which the try-except construction is placed. Different exception handler arguments for handlers of equal nest level can share the same space, since they are never needed at the same time.

Preferably the exception reference variable is initialised at runtime by the EH mechanism upon entry to the handler. To do that the EH mechanism must either know or be able to deduce the offset of the exception variable reference within the activation record.

The nest level for the handler can be used to deduce the relative offset for an exception reference variable, provided space for exception reference variables is allocated in order of the nest level at the top of the local data in the activation record. With the static table approach, rather then storing the offset of the exception reference explicitly, it is advantageous to deduce the offset from the nest level because the nest level is also used when performing cleanup for exception objects (See section 8.2).

The TERMINATE_EXCEPTION function shown represents the internal cleanup actions that must be performed by the EH mechanism, such as destroying the exception object.

11.2 Translating the TRY-FINALLY construction

Figure 25 shows two translation schemes for the try-finally construction. In scheme 1 code is appended to the finally statements to check whether the finally block was entered as a result of an exception, if so the exception is re-raised. The check is performed by a simple comparison. In translation scheme 2 the generated code for the statements in the finally section is duplicated, eliminating the runtime overhead of the test.

Both translation schemes have advantages and disadvantages. Which scheme is the more appropriate depends on the circumstances. Where the code for the finally statements is small in size, a duplication of the code is not a problem and scheme 2 can be chosen. Otherwise scheme 1 can be chosen.

The problem of deciding when a finally block was entered due to an exception can be solved by inspecting a local counter (called *finally_rnl* as in *finally_raise_nest_level* in figure 25). The counter is normally equal to zero, but in the event of an exception the counter is initialised to the nest level of the finally block before jumping into the finally handler.

To ensure that an inner finally block does not cause that an outer finally block is exited before time, the counter must be used rather then a simple flag[18]. The test at the end of the finally statements must ensure that the counter is equal to the nest level of the finally block (known at compile time) before re-raising the exception.

[18] Evidently not having considered this problem, a flag is used for this purpose in [Boland94].

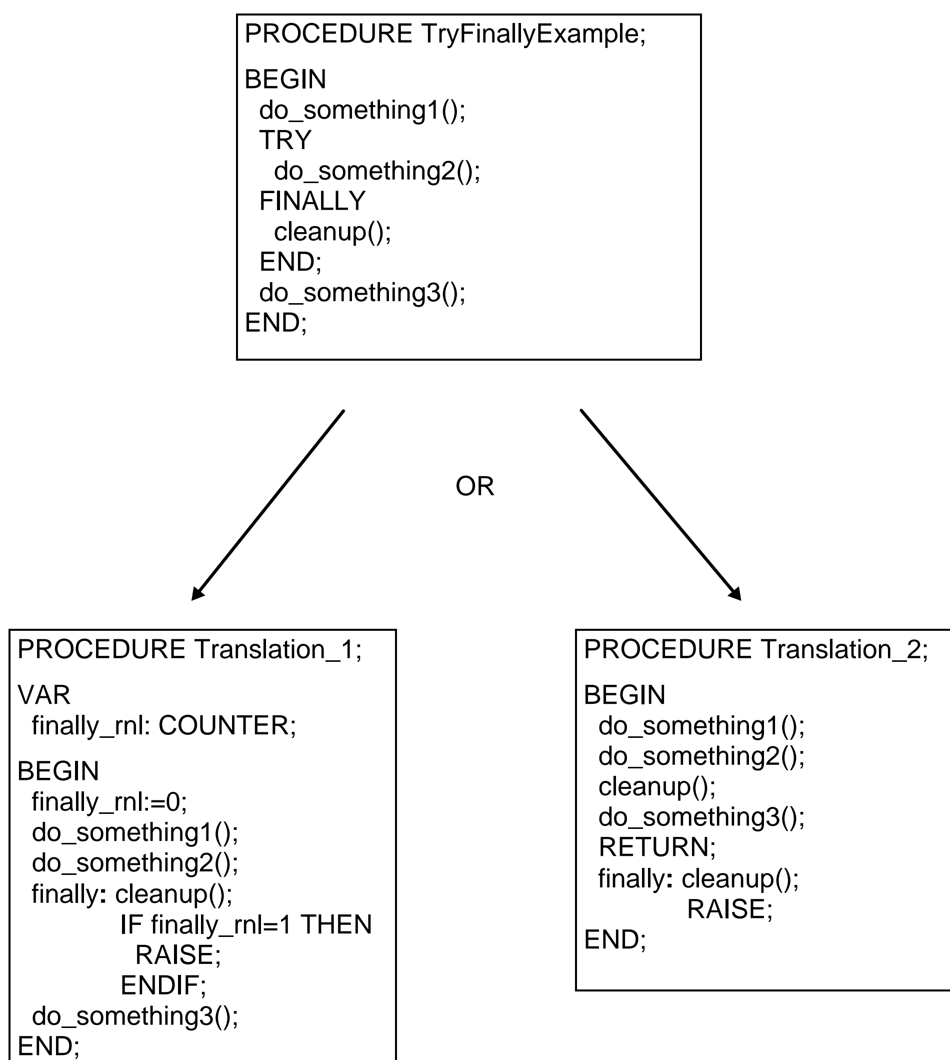


Figure 25 Translation schemes for TRY-FINALLY

Variations to the schemes for generating code for a try-finally construction are possible. Such variations include:

- In the event of an exception, place a jump or a break-point at the end of the finally statements, thereby avoiding the test code in scheme 1. This debugger inspired approach is only recommended (if recommended at all) for simple architectures without a cache.
- Generating an internal function for the statements in the finally block thereby avoiding the duplication of code in scheme 2. Not recommended because of the unnecessary large runtime overhead (function call + return).

11.3 Translating RAISE statements

The generation of code for a RAISE statement (or a RE-RAISE statement) is very similar to the generation of code for function calls. In essence a RAISE statement can be consider as a call to a function *RAISE* in the runtime library with the exception object as an argument.

Preferably, to save code space, the actual construction of the exception object is performed by the RAISE function in the runtime library. Thus the code generated by the compiler must provide the RAISE function with the address of the constructor for the exception object type as well as any arguments the constructor may have.

11.4 Other issues

When using register variables and when optimising code the compiler must be aware that any guarded calls (including those from RAISE statements) may provide a path from the try block to the exception handler [Cameron92]. Thus TRY constructions may be a major obstacle when generating optimised code.

Specifically, when performing common subexpression elimination the compiler must be aware that exceptions may cause assignments not to be executed if the right side of the assignment includes a function call. Thus elimination of unnecessary assignments may not always be possible.

12. The EH method used in the MEHL compiler

This chapter presents the details of the unified method for handling exceptions that is used in the native implementation of EH incorporated into the MEHL compiler. As the implementation is native it depends on the processor and operating system used for the target system. Consequently this chapter begins by stating the processor and operating system language as well as other specification issues of importance including the target assembler language that will be outputted by the compiler. The end system specification is followed by a presentation of the different aspects of the method implemented.

12.1 The target system specification

For the processor and operating system the following decisions have been made:

- INTEL 80286+ processor. The 80286 processor is neither the most primitive, nor the most advanced in the INTEL 80x86 family. Compared to other CISC processors, it is a processor of average capabilities, which is why it has been chosen. A short introduction to the processor is provided in Appendix F.

- IBM DOS operating system. DOS programs are staticly linked and there is no build in system support for handing exceptions (the whole work is appropriately left to the compiler).

For the MEHL compiler the following symbolic assembler language for has been targeted as output:

- Borland Turbo Assembler 3.0+ assembler format. Borland's Turbo Assembler is a state of the art macro assembler for 80x86 based systems. The Borland Turbo Assembler has been chosen because it is very powerful.- For instance it has unique support for object-oriented programming in assembler, very befitting for the MEHL compiler. A short introduction to the Turbo Assembler language is provided in Appendix G.

Naturally, the decision to use the INTEL 80286 processor, the IBM DOS operating system and the Borland Turbo Assembler, has been influenced by personal experience and preference. Anyhow this choice, as any other choice, of processor and operating system has a major influence on the implementation of the EH mechanism. Most importantly, the processor has no strictly defined call frames and the operating system has no build-in support for EH.

In addition to the processor and operating system targeted the following decisions about the 80286+ code generated by the MEHL compiler are important for the EH mechanism:

- Small 16 bit INTEL memory model. For this memory model words are 16 bits, pointers are 16 bit, code is placed in one segment and data, stack and heap are placed in another segment.
- Standard 80286 function frame using frame register BP in conjunction with the ENTER and LEAVE instructions. The standard function frame is extended to cover all functions produced by the MEHL compiler, even if they have no local environment in which case the function frame could (and normally would) be dropped (saving the overhead of the ENTER and LEAVE instructions). The standard function frame has been decided upon in order to simplify the EH mechanism.

12.2 The method in summary

In summary an unified method based on a compact static table approach to EH modified to handle mixed language programs has been adopted for the MEHL compiler. Although the method is of a general nature the compiler's variant is a native implementation tuned specifically for the target 80286+ architecture.

The static tables, which is the integral part of the EH support, can take up a significant amount of space. Influenced by the memory limitations in the target architecture an objective of this implementation has been to limit the size of the tables, hopefully proving false that a high static storage cost must be paid for an EH mechanism based on the static table approach.

EH is supported in MEHL programs through a combination of:

- The code and static tables generated by the compiler for individual MEHL programs. The code consists of user code for raise statements and exception handlers. The static tables provide the information necessary for the handling of exceptions by the EH runtime mechanism.
- The EH runtime mechanism which provides the common runtime support for all MEHL programs. The EH mechanism performs the various EH related actions at runtime, most notably transfer of control to exception handlers and the cleanup of relevant objects.

12.3 The static tables

An organisation of the static tables as described in section 7.1 has been chosen. It consists of one program table covering all functions in an entire program and a number of additional function tables, one for each of the functions that need the information.

To enable static binding of the main program table, to counter segment size limitations in the 80x86 architecture and to make the EH mechanism less sensitive to corruption of global data, the compiler generated tables must reside in a unique segment. Thus, in addition to the standard physical DATA and CODE segments, containing data & stack and code respectively, a physical segment called EXCEPT containing the tables for the EH is used.

The EXCEPT segment consists of two logical parts. One logical segment called EXCEPT@1 for the function tables and another logical segment called EXCEPT@2 for the main program table. The linker will automatically see to it that all parts of the EXCEPT segment from all the modules in a program are joined in one large physical segment. All the EXCEPT@1 and EXCEPT@2 logical parts are joined separately, in the same order as their containing module's CODE segments. Consequently a complete and fully sorted, statically linked main program table will be created by the linker. Thereby eliminating the need for initialisation by the EH mechanism.

12.3.1 The main program table

figure 26 shows the organization of the main program table. Each entry for the functions 1-N in a module M consists of a start address and a pointer to an optional function table. The value of the function table pointer field is equal to -1 when no function table exists[19]. To specify the end of a module a special entry with a function table pointer field value equal to -2 is used. All code and data references, of the types CODEPTR and DATAPTR respectively, are 16 bit in the implementation.

[19] The value zero (0) can not be used here because it's a valid address.

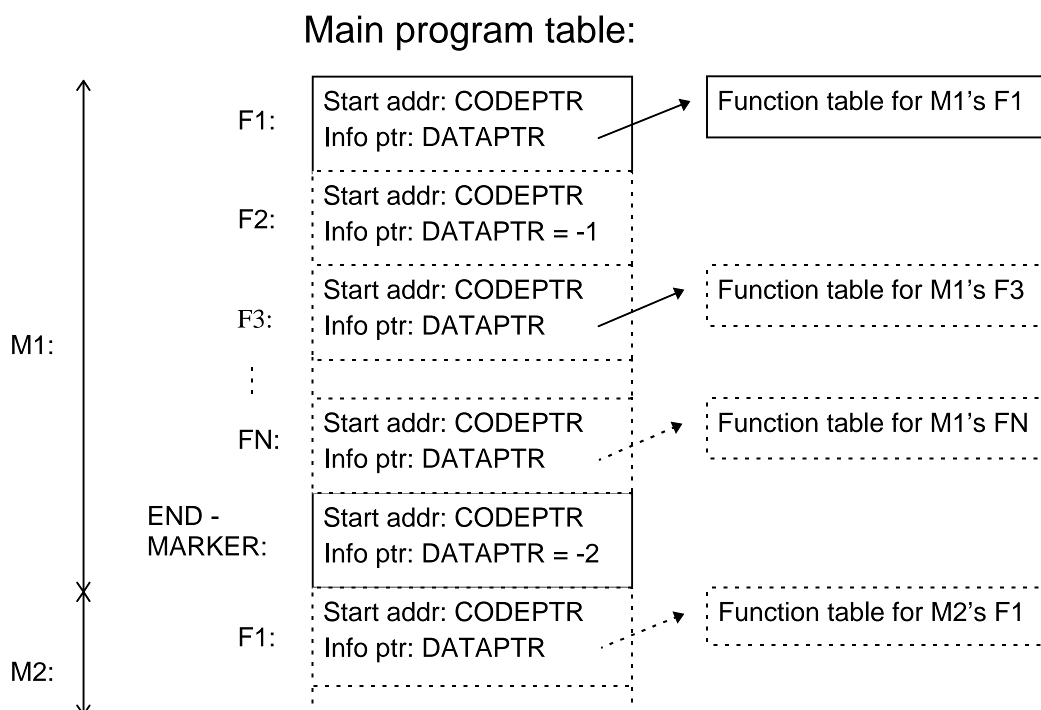


Figure 26 Organization of main program table

12.3.2 The function tables

figure 27 shows the organization of the function tables. The layout of the function tables has been optimized specifically for size, especially for normal functions with or without destructible objects. Referring to figure 27 each function table consists of the following information:

1 One byte of general information describing the function and any additional information that will follow. The format of the general information byte is presented in figure 28. The general information specifies which kind of function it is (ordinary function/procedure, method, constructor, destructor), whether is has destructible types (0, 1..6, or >6 - implying that the actual number of types will be specified later), whether it has exception handlers and finally it includes a (partial) raises-set specification (all exceptions can propagate, no exceptions can propagate, exceptions in a later specified raises-set can propagate).

2 An optional ordered list of the locations in the code where constructors and destructors for destructible types are called. If the number of destructible types is too large for the general information byte, a word containing the number of types initiates the list.

3 An optional ordered list of information for try constructions. The information for each construction begins with the guarded code region(s) covered by the try. After the region(s) a byte containing the *except nest level* is specified. The *except nest level* is followed by a byte which specifies the type of the try construction (finally or except) and by additional information for the specific construction.

   For a finally construction the specific information is the address of the handler and a byte containing the *finally nest level*. For an except construction the specific information consists of a list of information about the exception handlers. The information for each exception handler consists of the address of the handler and a list of the exception types covered by the handler.

4 An optional list of the exception types which are allowed to propagate.

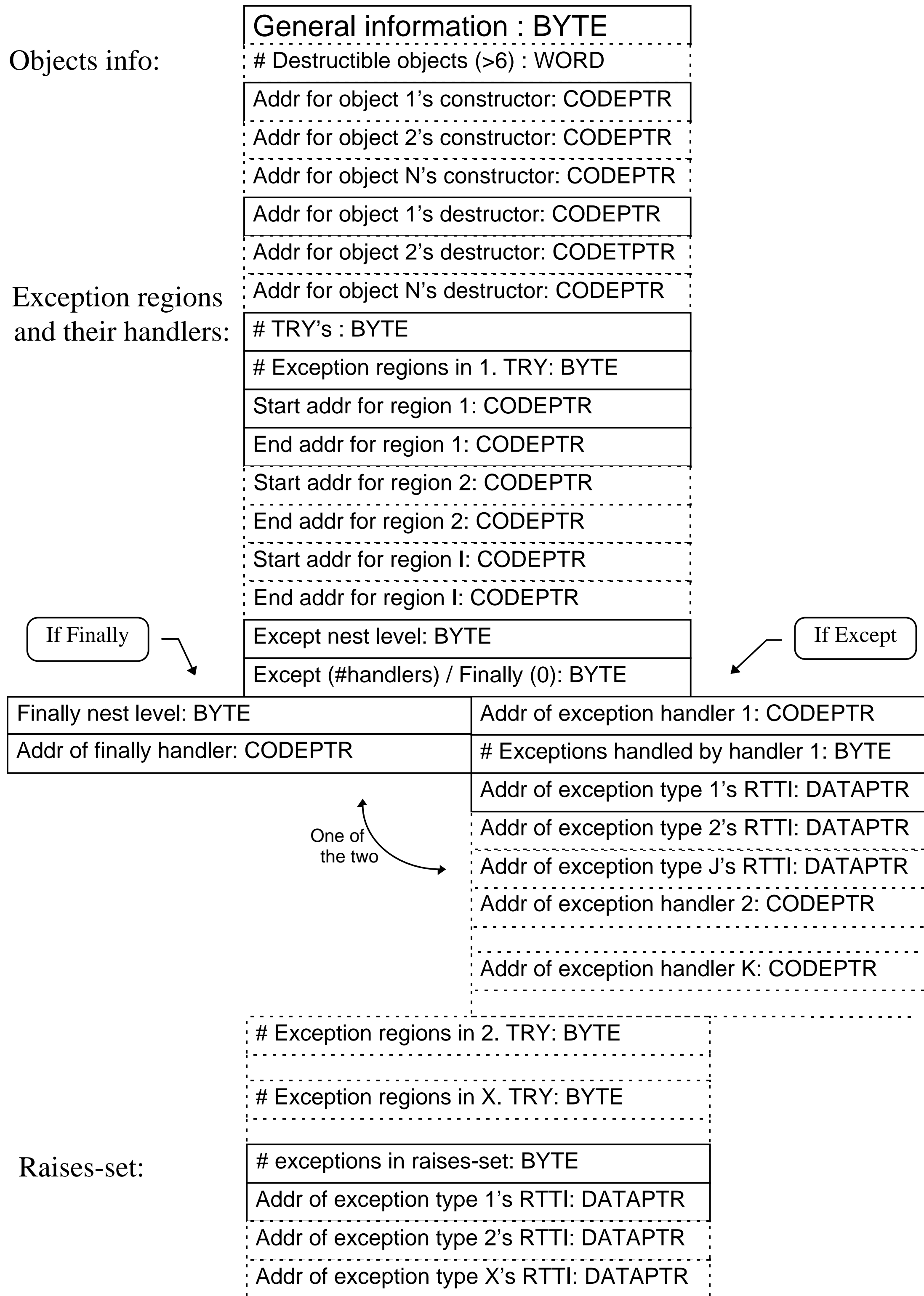


Figure 27 Organization of function tables

General information byte:

Bits: | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

0 → Function
1 → Method
2 → Constructor
3 → Destructor

0 → No objects
1..6 → #objects
7→ >=7

0 → No try's
1 → Has try's

0 → Raises any
1 → Raises none
2 → Raises set specified
3 → *Reserved*

Figure 28 Format of a function table's general information

To make it possible for the EH mechanism to determine which types to destroy in case of a unhandled exception in a constructor the construction address list must agree with the destruction address list. Each field with a construction address must have a corresponding destruction address field. Where a destructor but no constructor exists a null field must be inserted.

Destructible sub-types and global destructible types not located in the main module is a special case because the construction and destruction are done by two different functions.- For sub-types, the construction and destruction are handled by the containing object's constructor and destructor respectively. For global types, construction and destruction are handled by the initialization and deinitialization functions respectively.

A simple, but non-optimal, scheme that represents those destructible types in a similar way to local destructible types has been adopted. For a constructor or an initialization function the destruction address located in the corresponding destructor or deinitialization function is specified. For a destructor or a deinitialization function zero fields are inserted for each missing construction address.

12.4 Object cleanup information

Importantly, the pc-locations in the code where constructors or destructors are called are specified in the function tables, but there are no specific cleanup information about the destructible objects such as their addresses and the addresses of destructors etc.

The pc-locations of construction and destruction of objects are necessary to determine the objects to destroy. Similarly specific cleanup information is necessary when performing actual destruction of objects. However, unlike the pc-locations, explicit cleanup information stored in the function table is dispensable because it is already present at the pc-locations of destruction in the code.

The pc-locations for destruction could be called directly by the EH mechanism, but this would require the translated code to be modified at runtime in order to ensure that execution is returned to the mechanism after the destruction. Clearly, in general modifying code at runtime is not recommended, if indeed possible, for a lot of reasons most notably problems with shared code and with instruction cache's.- But what about inspecting a known code sequence for a destruction of an object and retrieving the necessary information for a corresponding destruction by the EH mechanism ? Apart from reasons of aesthetics, only the increased dependency between the compiler and the EH mechanism and the slight decrease in performance may be an issue here[20].

Contrary to what might be expected a cleanup scheme based on code inquiries is relatively easy to implement.- Easy because the code sequences for which inquiries must be possible are predetermined by the compiler, because they are small in size and because only a limited number of variations of code sequences are possible.

---

[20] Providing it is possible to perform reads of program code in the target artitecture as it is the case for the Intel 80x86.

For the code generated by the MEHL compiler the information necessary for the destruction of a typical local object can be retrieved by examining a code sequence of only three instructions.. figure 29 presents a listing of the code sequences generated by the MEHL compiler for the procedure *P* from figure 12 with the two local destructible objects *a* and *c* of type *A* and *C* respectively. The important information about the BP relative location of the objects and the PC relative address of the destructor is highlighted. The PC locations that must be saved in the function information table and used for the code inquiries are specified by the two *LABEL* statements. Note that the initial push instruction, which signifies that the object is not a heap object, can always be disregarded as it yields no cleanup information.

```
; Destruction of object c:
6A 01           push 1
LABEL $_P_DTR1  CODEPTR
8D 5E FE        lea  bx,[@@c]
53              push bx
E8 FEC3         call +@Table__C_ |   _DONE

; Destruction of object a:
6A 01           push 1
LABEL $_P_DTR2  CODEPTR
8D 5E FD        lea  bx,[@@a]
53              push bx
E8 FDEE         call +@Table__A_ |   _DONE
```

Figure 29 Listing of code sequences for destruction of objects in procedure P

Clearly, the retrieval of cleanup information from the code sequences shown in figure 29 is trivial. It is however a slight complication that the BP relative offset of the object can actually be either 8 or 16 bit in size. The size of the offset can be determined by inspecting bit 7 of the ModR/M byte in the 3-4 byte long LEA instruction. If the bit is set then the BP displacement is 16 bit otherwise it is 8 bit in size. The ModR/M byte is located right after the LEA instruction's upcode (8D). In the example the ModR/M byte is equal to 5E implying that the displacement which is FE is 8 bit in size.

Note that in addition to the specific information for a code sequence the EH mechanism must be able to retrieve the address after the call of the destructor (the return address for the destructor call). The address is used when checking for an exception in a destructor. When an exception has occurred in a destructor the address is equal to the PC value retrieved when unwinding the stack.

In the example the return address for the destructor call is located at a positive offset of seven bytes from the start of the code sequence (not including the initial *push 1* instruction which is always present for local, global and sub-objects). If not for the purpose of the retrieval of cleanup information the destruction address saved in the function table would contain the return address for the destructor call (and still could if an extra pointer per object where allocated) instead of the start address of the significant part of the destroy action. In fact this is the case for the address references to the construction of an object.

Apart from local objects code inquiries for global objects, member objects and ancestor objects as well as arrays of objects must be possible. An overview of the distinct parts of the 7 code sequences possible is shown in figure 30.

The code sequences may vary a little where either an 8 or 16 bit operand is possible for an instruction. This is the case for the LEA instruction in the code for a local variable or member variable, and for the last two immediate PUSH instructions in the code for arrays. For the LEA instruction the 8/16 bit operand question is solved by examining bit 7 of the ModR/M byte as previously discussed. For immediate PUSH instructions the size of the operand depends on the upcode which is 6A for a 8 bit operand and 68 for a 16 bit operand.

```
; Local variable a of type A
8D 5E FD        lea  bx,[@@a]
53              push bx
E8 FDEE         call +@Table__A_ |   _DONE
```

```
; Member variable a of type A
8D 5C 04        lea  bx,[(_A_ PTR si).__a]
53              push bx
E8 FEAE         call +@Table__A_ |   _DONE

; Global variable a of type A
68 0002r        push offset _a
E8 FEE0         call +@Table__A_ |   _DONE

; Ancestor of type A
56              push si
E8 FEEE         call +@Table__A_ |   _DONE

; Local variable a of type Array[4] of A
8D 5E F6        lea  bx,[@@a]
53              push bx
68 00C9r        push offset +@Table__A_ | _DONE
6A 02           push size _A_
6A 04           push 4
E8 0000e        call ARRAY_DTR_APPLY

; Member variable a of type Array[4] of A
8D 5C 02        lea bx,[(_B_ ptr si).__a]
53              push bx
68 00C9r        push offset +@Table__A_ | _DONE
6A 02           push size _A_
6A 04           push 4
E8 0000e        call ARRAY_DTR_APPLY

; Global variable a of type Array[4] of A
68 0185r        push offset _a
68 00C9r        push offset +@Table__A_ | _DONE
6A 02           push size _A_
6A 04           push 4
E8 0000e        call ARRAY_DTR_APPLY
```

Figure 30 Listing of typical code sequences for cleanup actions.

The PC-locations pointed to in the function tables can be the start of any of the 7 code sequences in figure 30. Using the PC value the EH mechanism must be able to retrieve the specific information and to locate the destructor's return address for any of these code sequences.

Note that the code sequences for local and member objects are nearly identical. The only difference is the base register for the relative displacement which is BP for local objects and another index register (SI), representing the address of the containing object, for member objects. The LEA instruction's ModR/M byte specifies which base register is used for the instruction. If the bits 0-2 of the ModR/M byte is equal to 110b then register BP is used as the base register, otherwise another register or register combination is used.

Note that the code inspection technique used to retrieve cleanup information saves at least two WORDs of table storage space for each destructible object (and even more for arrays). For typical functions with destructible objects but no try constructions and trivial raises set specifications the static table is thereby reduced to : 2 WORD's for the program table + 1 BYTE for general information + 2 WORD's describing the construction and destruction address respectivly for each destructible object. The space saved for destructible objects corresponds to a 30-50% decrease in storage cost for the static EH tables (30% for one object, up to 50% for a large number of objects)!

12.4.1 Cleanup of global objects

Each MEHL program contains startup code for initialization and deinitialization. Construction and destruction of global objects are performed by the initialization and deinitialization code respectivly for each module. The

initialization and deinitialization of modules are, similar to construction and destruction of objects, symmetric. figure 31 shows the startup code generated for a MEHL program A which uses two modules B and C:

```
      STARTUPCODE ; System startup code.
INIT: CALL @C     ; Init and execute startup for module C.
      CALL @B     ; Init and execute startup for module B.
      CALL @A     ; Init, execute & deinit program A.
      CALL @FIN_B ; Deinit module B.
      CALL @FIN_C ; Deinit module C.
EXIT: EXITCODE 0  ; System exit code.
```

Figure 31 Startup code for program A using modules B and C

When terminating a program due to an unhandled exception the EH mechanism must destroy all previously constructed global objects with destructors for all modules. This is done by calling the deinitialization code for each of the modules which have been successfully initialized.

As each call instruction takes up 3 bytes and as the difference between EXIT and INIT can be statically computed, the deinitialization routines which must be called can easily be found out from the PC-value retrieved when propagating the unhandled exception (in the range INIT-EXIT).

12.5 Retrieving the size of a function's local environment

Significantly, the size the local data for a function, which is necessary when unwinding, is not specified in the function table or in the main program table. The size is not specified because the information can be retrieved in a trivial way by inspecting the corresponding argument in the ENTER instruction at the head of the function. The standard prolog and epilog code related to the function frame for a function with two WORD's (4 bytes) of local data is shown in figure 32. As it is evident from the example the size of the local data can be retrieved from the ENTER operand word at offset 1 from the start of the function.

```
SampleFunction PROC
               LOCAL a:WORD, b:WORD
C8 0004 00     ENTERW  0004h,0
               ...
C9             LEAVEW
C3             RET
SampleFunction ENDP
```

Figure 32 Listing of function frame related prolog code and epilog code

Importantly the simple code inspection technique used to retrieve the size of local data saves one WORD of table storage space for each compiler generated function. For simple functions with no additional function tables this corresponds to a 33% decrease in storage cost for the static EH tables!

12.6 Runtime type information for exception types

Figure 33 presents the format of the runtime type information (RTTI) for the exception types. Besides the name of exception that is include in the RTTI, the location and format of the RTTI for exceptions are identical to that shown in figure 21.

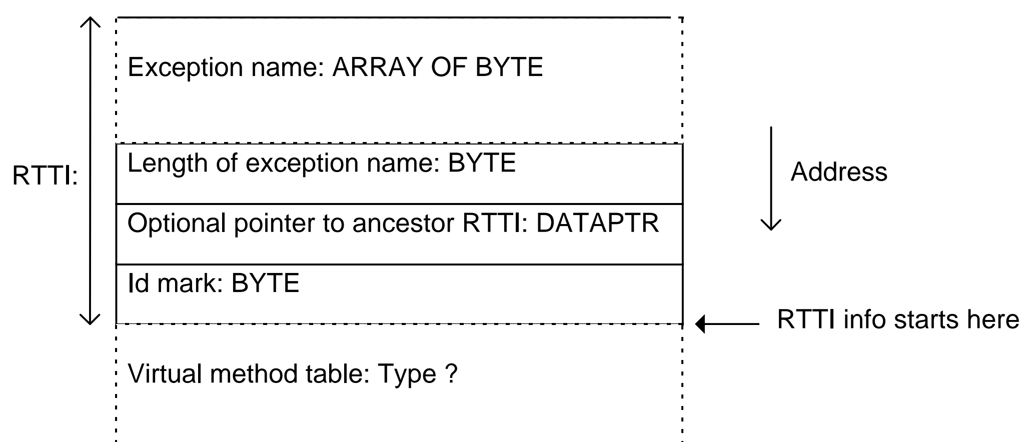


Figure 33 Format of runtime type information

The name of the exception type is included only to provide an exception name that can be included in error messages. The exception type name is not used for exception identification, for which the [Cameron92] method is used. Note that because the exact type (and size) of the virtual method table (VMT) varies the RTTI information is structured in order of decreasing address. Therefore the RTTI information is specified by its end address, which is equal to the address of the VMT.

12.7 Addressing the local environment of MEHL functions

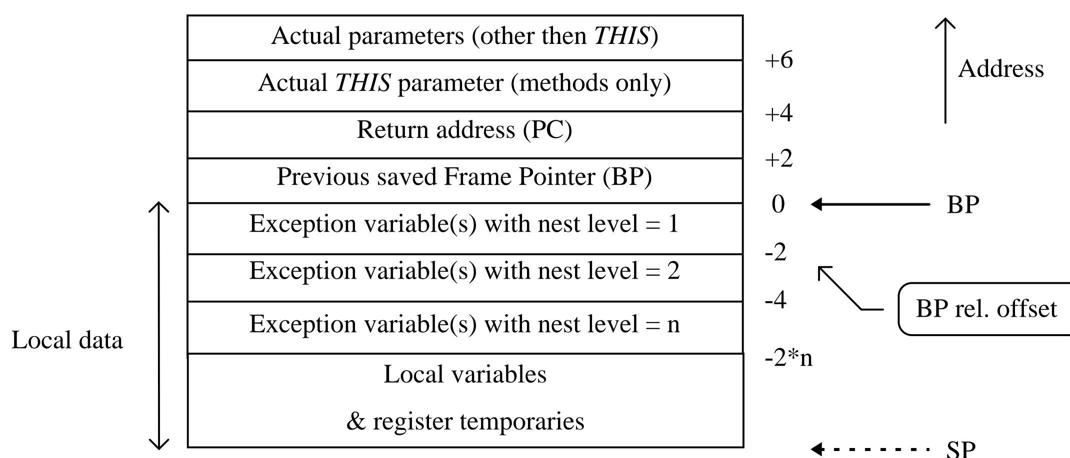


Figure 34 Layout of activation record for MEHL compiler

The format of the MEHL compiler's activation record is shown in figure 34. For entries that can be retrieved at a fixed offset from the frame pointer BP the relative offset is stated. The *THIS* parameter is only present for methods and the specified exception variables *1..n* are present only for functions (or procedures or methods) having TRY-EXCEPT constructions with exception handlers of nested depth *n*.

The format of the activation record for the MEHL compiler has been carefully designed so that all the information of interest for the EH runtime mechanism is available at fixed offsets from BP (Eliminating the need for the function table to describe the various interesting relative offsets).

The information of interest for the EH mechanism is:

- The caller's BP and the return address that can be found at the respective fixed offsets 0 and 2 from BP. As already discussed these values are fundamental for unwinding.
- The address of the current object (called *THIS*) for a method that can be found at fixed offset 4 from BP. Internally the MEHL compiler uses an index register (SI) to access object data for all methods. Before transferring control to a handler located in a method the *THIS* value found in the activation record is assigned to the register (Eliminating code for reloads at the start of each handler). Also, this address is also used when destroying sub-objects.
- The exception variables located at fixed offsets -2*(1..*n*) from BP, where *n* is the nest level of the exception handler. Before transferring control to an exception handler the address of the exception object is assigned to the exception variable for the handler.

12.8 Static tables example

In this section the static table tables and exception related code for an example program is presented. An extract of the MEHL source code for the example is shown in figure 35. A condensed extract of the code generated by the MEHL compiler is shown in figure 36.

```
PROGRAM table_example;

TYPE
  A = OBJECT
       CONSTRUCTOR init(); FORWARD;
       DESTRUCTOR done(); FORWARD;
      END;

  B = OBJECT
       CONSTRUCTOR init(); FORWARD;
       DESTRUCTOR done(); FORWARD;
      END;

  C = OBJECT (B)
       CONSTRUCTOR init(); FORWARD;
       DESTRUCTOR done(); FORWARD;
      END;

  ExceptionA = OBJECT (Exception) END;

  ExceptionB = OBJECT (Exception) END;

PROCEDURE P() RAISES {ExceptionA, ExceptionB};

VAR
  a: A;
  c: C;

BEGIN
...
END;
```

```
BEGIN
 TRY
   TRY
     P();
   EXCEPT
     | ExceptionA(e) => ...
     | ExceptionB(f) => ...
   END;
 FINALLY
  ...
 END;
END.
```

Figure 35 Source code extract of table_example program

```
; tbl_ex.asm

SEGMENT _DATA WORD PUBLIC 'DATA'
STRUC _EXCEPTIONA_ GLOBAL _EXCEPTION_ METHOD {
_INIT : WORD = _INIT@_EXCEPTIONA
VIRTUAL _DONE : WORD = _DONE@_EXCEPTIONA
}
ENDS _EXCEPTIONA_
DB 'EXCEPTIONA',10 ; RTTI information
DW @TableAddr__EXCEPTION_
DB 0
TBLINST

STRUC _EXCEPTIONB_ GLOBAL _EXCEPTION_ METHOD {
_INIT : WORD = _INIT@_EXCEPTIONB
VIRTUAL _DONE : WORD = _DONE@_EXCEPTIONB
}
ENDS _EXCEPTIONB_
DB 'EXCEPTIONB',10 ; RTTI information
DW @TableAddr__EXCEPTION_
DB 0
TBLINST

EXTRN FINALLY_RAISE_NEST_LEVEL:WORD
ENDS _DATA

GROUP DGROUP _DATA, _STACK
SEGMENT _TEXT WORD PUBLIC 'CODE'
ASSUME CS:_TEXT, DS:DGROUP, SS:DGROUP

EXTRN TERMINATE_EXCEPTION:PROC
EXTRN CONTINUE_RAISE_FROM_FINALLY:PROC

      STARTUPCODE
INIT: call @STDIO
      call @TABLE_EXAMPLE
      call @FIN_STDIO
EXIT: EXITCODE

PROC  _INIT@_C
      ARG @@THIS:WORD
      cmp [@@THIS],0
      jne @@OVR
      push size _C_
      call NEW
      or ax,ax
      jz @@FIN
      mov [@@THIS],ax
LABEL $_INIT@_C_CTR1 CODEPTR
@@OVR:push [@@THIS]
      call +@Table__B_ | _INIT
      ...
```

```
@@FIN:ret
ENDP  _INIT@_C

PROC  _DONE@_C
      ARG @@STATUS:WORD,@@THIS:WORD
      mov si,[@@THIS]
      ...
@@FIN:push 1
LABEL $_DONE@_C_DTR1 CODEPTR
      push si
      call +@Table__B_ | _DONE
      cmp [@@STATUS],0
      jne @@FRET
      push [@@THIS]
      call DISPOSE
@@FRET:ret
ENDP  _DONE@_C

PROC  _P
      LOCAL @@a:_A_, @@c:_C_
      lea bx,[@@a]
      push bx
      call +@Table__A_ | _INIT
LABEL $_P_CTR1 CODEPTR
      lea bx,[@@c]
      push bx
      call +@Table__C_ | _INIT
LABEL $_P_CTR2 CODEPTR
      ...
@@FIN:push 1
LABEL $_P_DTR1 CODEPTR
      lea bx,[@@c]
      push bx
      call +@Table__C_ | _DONE
      push 1
LABEL $_P_DTR2 CODEPTR
      lea bx,[@@a]
      push bx
      call +@Table__A_ | _DONE
      ret
ENDP  _P

PROC  @TABLE_EXAMPLE
      LOCAL @@_f:WORD:0,@@_e:WORD
LABEL $@TABLE_EXAMPLE_SR1 CODEPTR
LABEL $@TABLE_EXAMPLE_SR3 CODEPTR
      call _P
LABEL $@TABLE_EXAMPLE_ER3 CODEPTR
LABEL $@TABLE_EXAMPLE_ER1 CODEPTR
@@L1: ...
      cmp [FINALLY_RAISE_NEST_LEVEL],1
      jb @@FIN
      call CONTINUE_RAISE_FROM_FINALLY
@@FIN:ret

LABEL $@TABLE_EXAMPLE_SR2 CODEPTR
_$L0: ...
      push offset @@L1
      jmp TERMINATE_EXCEPTION
_$L1: ...
      push offset @@L1
      jmp TERMINATE_EXCEPTION
LABEL $@TABLE_EXAMPLE_ER2 CODEPTR
ENDP  @TABLE_EXAMPLE

LABEL END_LAST_PROC CODEPTR
ENDS _TEXT

SEGMENT _EXCEPT@1 PARA PUBLIC 'EXCEPT'
LABEL $$_INIT@_A DATAPTR
```

```
      DB 10000000B

LABEL $$_INIT@_B DATAPTR
      DB 10000000B

LABEL $$_INIT@_C DATAPTR
      DB 10001000B
      DW $_INIT@_C_CTR1
      DW $_DONE@_C_DTR1

LABEL $$_DONE@_C DATAPTR
      DB 11001000B
      DW 0
      DW $_DONE@_C_DTR1

LABEL $$_INIT@_EXCEPTIONA DATAPTR
      DB 10001000B
      DW $_INIT@_EXCEPTIONA_CTR1
      DW $_DONE@_EXCEPTIONA_DTR1

LABEL $$_DONE@_EXCEPTIONA DATAPTR
      DB 11001000B
      DW 0
      DW $_DONE@_EXCEPTIONA_DTR1

LABEL $$_INIT@_EXCEPTIONB DATAPTR
      DB 10001000B
      DW $_INIT@_EXCEPTIONB_CTR1
      DW $_DONE@_EXCEPTIONB_DTR1

LABEL $$_DONE@_EXCEPTIONB DATAPTR
      DB 11001000B
      DW 0
      DW $_DONE@_EXCEPTIONB_DTR1

LABEL $$_P DATAPTR
      DB 00010010B
      DW $_P_CTR1
      DW $_P_CTR2
      DW $_P_DTR1
      DW $_P_DTR2
      DB 2 ; Raises spec
      DW @TableAddr__EXCEPTIONA_
      DW @TableAddr__EXCEPTIONB_

LABEL $$@TABLE_EXAMPLE DATAPTR
      DB 00000100B
      DB 2 ; # Try's
      DB 1 ; # Regions in try
      DW $@TABLE_EXAMPLE_SR3
      DW $@TABLE_EXAMPLE_ER3
      DB 1 ; Except nest level
      DB 2 ; Except, # Handlers
      DW _$L0
      DB 1 ; # Exception types
      DW @TableAddr__EXCEPTIONA_
      DW _$L1
      DB 1 ; # Exception types
      DW @TableAddr__EXCEPTIONB_
      DB 2 ; # Regions in try
      DW $@TABLE_EXAMPLE_SR1
      DW $@TABLE_EXAMPLE_ER1
      DW $@TABLE_EXAMPLE_SR2
      DW $@TABLE_EXAMPLE_ER2
      DB 0 ; Except nest level
      DB 0 ; Finally
      DB 1 ; Finally nest level
      DW _$L2
ENDS _EXCEPT@1
```

```
SEGMENT _EXCEPT@2 BYTE PUBLIC 'EXCEPT'
      DW _INIT@_A
      DW $$_INIT@_A

      DW _DONE@_A
      DW -1 ; No info

      DW _INIT@_B
      DW $$_INIT@_B

      DW _DONE@_B
      DW -1 ; No info

      DW _INIT@_C
      DW $$_INIT@_C

      DW _DONE@_C
      DW $$_DONE@_C

      DW _INIT@_EXCEPTIONA
      DW $$_INIT@_EXCEPTIONA

      DW _TEST@_EXCEPTIONA
      DW -1 ; No info

      DW _DONE@_EXCEPTIONA
      DW $$_DONE@_EXCEPTIONA

      DW _INIT@_EXCEPTIONB
      DW $$_INIT@_EXCEPTIONB

      DW _TEST@_EXCEPTIONB
      DW -1 ; No info

      DW _DONE@_EXCEPTIONB
      DW $$_DONE@_EXCEPTIONB

      DW _P
      DW $$_P

      DW @TABLE_EXAMPLE
      DW $$@TABLE_EXAMPLE

      DW END_LAST_PROC
      DW -2 ; End marker
ENDS _EXCEPT@2

GROUP EXCEPT _EXCEPT@1, _EXCEPT@2
```

Figure 36 Extract of generated code for table_example program

12.9 Code for EH constructions and statements

This section specifies the code generated by the MEHL compiler for the EH constructions and statements.

12.9.1 **The try-except construction**

For the try-except construction the no overhead scheme discussed in section 11.1 has been chosen. The code generated at the end of each handler in order to terminate the exception is:

```
push offset <resume_addr>
jmp TERMINATE_EXCEPTION
```

TERMINATE_EXCEPTION is a runtime library routine that discards the current exception and destroys its exception object. Prior to jumping to the function the address of the location where execution must resume is pushed on the stack, in effect faking a return address as would have been pushed for a function call to TERMINATE_EXCEPTION.

The exception reference variables are allocated as local variables at the top of the local data on the stack in order of their except nest level. When two exception variables can not be used at the same time (same except nest level) they share the same space. In the assembler two or more variables can be made to share the same space by specifying all but the last variable at issue as an array of 0 elements of its type. For the two exception variables f and e in specified by the main procedure in figure 36 this is done as shown below:

```
LOCAL @@_f:WORD:0, @@_e:WORD
```

12.9.2 The try-finally construction

For the try-finally construction the generated code at the end of a finally section depends on whether the copy or test scheme discussed in section 11.2 is used. Both schemes are supported by the MEHL compiler. Which scheme is actually used for a specific try-finally construction is up to the compiler. The objective for both sequences of generated code is to stop user-code execution in case of an exception and transfer control to the EH runtime mechanism.

When a test scheme is used (Translation scheme 1) the following code sequence is generated at the end of a finally section:

```
        cmp [FINALLY_RAISE_NEST_LEVEL], <finally_nest_level>
        jb @@OVR
        call CONTINUE_RAISE_FROM_FINALLY
@@OVR:
```

When a copy scheme is used (Translation scheme 2) the following simplified code is generated at the end of a finally section:

```
call CONTINUE_RAISE_FROM_FINALLY
```

CONTINUE_RAISE_FROM_FINALLY is a runtime library routine. It is functionally comparable to RERAISE. The variable FINALLY_RAISE_NEST_LEVEL is a global variable declared in the runtime library.

12.9.3 The RAISE statement

The code generated for a *RAISE exception.constructor(arg1,arg2,...,argn)* statement is presented below. RAISE is a runtime library function which as stack arguments takes the constructor parameters plus the PC address at the raise point and as a BX register argument takes the address of the constructor.

```
push <arg1>
push <arg2>
...
push <argn>
mov bx, offset <exception type's constructor>
call RAISE
```

12.9.4 The RERAISE statement

The code generated for a *RERAISE* statement is presented below. RERAISE is a runtime library routine which as a stack argument takes the PC address of the re-raise point.

```
call RERAISE
```

12.9.5 The RETURN statement

The code generated for a *RETURN x* statement contained in a try construction is presented below (return statements not contained in try constructions are performed by simple jumps). RAISE_RETURN is a runtime library routine that raises an internal (non-propagating) RETURN exception, whose main objective is to ensure that finally statements are executed if the return statement is guarded by a try-finally construction. RAISE_RETURN takes the PC address of the return statement as a stack argument, the address of the normal return entry point in BX and an eventual function result argument in register AX.

```
( mov ax,<function result value> )
  mov bx,<function return addr>
  call RAISE_RETURN
```

12.9.6 Call's to external functions

When non-MEHL functions which may indirectly raise an exception are called the MEHL compiler generates a call to a special runtime library operation called EH_STUB instead of a call to the external function. The EH_STUB operation takes care to push and pop a location marker and to call the external function at issue.

The code sequence generated to call the EH_STUB operation is:

```
push 0
push <arg1>
push <arg2>
...
push <argn>
mov bx,<Addr of external function>
mov si,<Total size of args to external function>
call EH_STUB
```

Arguments *arg1-argn* are arguments to the external function. Note that an initial word (with value 0) is pushed on the stack. The word is not designated for the external function. It is used internally by the EH_STUB operation.

13. The EH runtime mechanism

This chapter contains a pseudo code specification for the EH runtime mechanism used in this thesis by the MEHL compiler. It is not an entirely normal pseudo code specification because it covers the many important low-level issues which are closely connected to the 80286+ architecture and the code generated by the MEHL compiler.

Importantly the EH runtime mechanism must operate between two different worlds which are the user-program context and the EH mechanism context. Each context consists of a stack and a pair of stack and frame registers. The mechanism switches between the two contexts with the *SwitchToUserProgramContext()* and *Switch-ToExceptionDispatcherContext()* operations.

Registers (variables of type *REGISTER* in the pseudo code) must be used as an intermediary when moving information between the user-program and the EH mechanism contexts, since only one context is visible at a time. When the user-program context is active only the local environments of the user program are accessible. It must be active when retrieving user-program specific information from a local environment such as the return address for a function or when manipulating the user's registers. In contrary, when the EH mechanism context is enabled, only the local environments for the EH functions are accessible.

Some pseudo code operations can not possibly correspond to a function in the implementation because of context issues or because no local environment can be allowed for the operation. Such operations are specified as of type *CODE* rather then the usual *PROCEDURE* or *FUNCTION*. In most cases a *CODE* operation is specified when an operation is executed with the user program context as the active context. One example is the RAISE pseudo code whose prime responsibility is to switch to the EH mechanism context, set up a new local environment for the *RAISE_BODY* procedure and forward execution to the all important *RAISE_BODY* procedure.

In a few cases, through the use of a user-program context or a *CODE* operation, local data is referred to which are located outside the logical scope of the current operation! For that reason when a local variable is referred to which is not declared for the current operation, note the context in which it is referred to. For the user-program context, local variables specified for *the ARRAY_CTR_APPLY and ARRAY_DTR_APPLY* are referred to by the out of scope operations *MarkWithinArrayCtrApply* and *MarkWithinArrayDtrApply.* For the EH mechanism context, local variables belonging to the *RAISE_BODY* procedure are referred to by the out of scope operations *RERAISE, CONTINUE_RAISE_FROM_FINALLY* and *TERMINATE_EXCEPTION* and local variables belonging to environments of previous activations of *RAISE_BODY* are referred to by the operation T*erminatePreviousExceptions*.

Generally, when special hardware register and stack manipulations are needed the operations are represented in a combination of condensed assembler (ASM) statements and the use of variables of type *REGISTER*.

Notes for 2nd. ed: Bugs have been discovered in the pseudo-code specification, some of which still persist. In particular note:

1. This function EH_DEALLOCATE does not free the entire memory block after combining memory blocks. To solve this, the size entry in EH_BUFFER_NODE must be moved to the end of the node and the operations EH_ALLOCATE and EH_DEALLOCATE must be adjusted correspondingly.
2. A global rather then the correct <u>local</u> *finally_raise_nest_level* counter is used.

```
PROGRAM EH_RUNTIME_MECHANISM;

CONST
 THIS_REG = SI;
 THIS_REG_REL_OFS = +4;
 EXCEPTION_VAR_MUL_FOR_EXCEPT_LEVEL = -2;
 EH_BUFFER_SIZE = 10K;          (* Value not critical *)
 MAX_INFORMATION_MARKERS = 10; (* Ditto *)
 END_MARKER_VALUE = -2;
 NO_FUNCTION_TABLE_VALUE = -1;
```

```
 FUNCTION_NOT_FOUND = -1;

TYPE
 PROGRAM_TABLE_ENTRY = RECORD
                         begin_range: CODEPTR;
                         info_ptr: POINTER TO FUNCTION_TABLE;
                        END;
 FUNCTION_TABLE = (* See figure 27 for details *);
 LOCATION_MARKER = RECORD
                    call_point: CODEPTR;
                    saved_BP: STACKPTR;
                   END;
 REGION = RECORD
           start_addr: CODEPTR;
           end_addr: CODEPTR;
          END;
 PROC_TYPE = (FUNCTION, METHOD, CONSTRUCTOR, DESTRUCTOR);
 EXCEPTION_TYPE = (RAISE_EXCEPTION_FLAG, RAISE_RETURN_FLAG);
 HANDLER_TYPE = (EXCEPT, FINALLY, NO_HANDLER);
 RAISES_SPEC = (ANY, NONE, RAISES_SET);
 EXCEPTION = OBJECT
              vmt_ptr: DATAPTR;(* Called @Mptr__Exception_ in impl *)
              ... (* Data for exception object of varying size *)
             END;
 RTTI_MARK = (MARKED, UNMARKED);
 RTTI_DATA = RECORD
              name: ARRAY OF BYTE; (* Of varying size *)
              name_length: BYTE;
              ancestor : POINTER TO END OF RTTI_DATA;
              id_mark : RTTI_MARK;
             END;
 EH_BUFFER_NODE = RECORD
                   size: WORD;
                   ... (* Data for exception of varying size *)
                  END;
 EH_BUFFER_TYPE = ARRAY [1..EH_BUFFER_SIZE] OF BYTE;
 ADDRESS_TYPE = (ABSOLUTE, BP_RELATIVE, THIS_RELATIVE);
```

The constants *FIRST_PROGRAM_TABLE_ENTRY* and *LAST_PROGRAM_TABLE* are set to point to the start and end of the main program table. *FIRST_PROGRAM_TABLE_ENTRY* and the constant *PROGRAM_TABLE_LEN* is initialised by the linker. The constant *LAST_PROGRAM_TABLE* is calculated initially by the EH mechanism. *FIRST_PROGRAM_TABLE_ENTRY* and *LAST_PROGRAM_TABLE_ENTRY* are used by the function *FindFunction* as start and end points for binary search. Note that the last entry in the main program table is always an end-of-module-marker which is why it is ignored here.

```
CONST
 FIRST_PROGRAM_TABLE_ENTRY: POINTER TO PROGRAM_TABLE_ENTRY
                             = Start of EXCEPT@2 segment.
 PROGRAM_TABLE_LEN: WORD = Length of EXCEPT@2 segment.
 LAST_PROGRAM_TABLE_ENTRY: POINTER TO PROGRAM_TABLE_ENTRY
                           =FIRST_PROGRAM_TABLE_ENTRY+PROGRAM_TABLE_LEN
                           -SIZEOF(PROGRAM_TABLE_ENTRY);
```

The global variable *finally_raise_nest_level* is tested at the end of the finally statements when a test translation scheme is used for the try-finally construction. The *finally_raise_nest_level* variable is set to the finally nest level by the function *execute_handler* prior to the transfer of control to the user program. It is saved by *RAISE_BODY* when an exception is initiated and restored when an exception terminates by *TERMINATE_EXCEPTION* etc.

```
VAR
 finally_raise_nest_level: WORD;
```

The global variables *eh_buffer* and *eh_buffer_pos* are used to manage global storage for exception objects. The variable *eh_buffer* holds block(s) of global storage for exception objects and the variable *eh_buffer_pos* is used as a free block pointer. The variables are manipulated by the function *EH_ALLOCATE* and the procedure *EH_DEALLOCATE*.

```
VAR
 eh_buffer: ARRAY [1..EH_BUFFER_SIZE] OF BYTE;
 eh_buffer_pos: POINTER TO EH_BUFFER_NODE = @eh_buffer;
```

The global variables *info_markes_buf* and *info_markers_pos* are used to manage global storage for information makers. The variable *info_markers_buf* holds the information markers and the variable *info_markers_pos* keeps track of the number of information markers stored. The variables are manipulated by the routines *PushLocationMarker* and *PopLocationMarker*.

```
VAR
 info_markers_buf : ARRAY [1..MAX_INFORMATION_MARKERS] OF LOCATION_MARKER;
 info_markers_pos : WORD = 1;
```

The variables *PROGRAM_SP*, *PROGRAM_BP*, *EH_STACK_SP* and *EH_STACK_BP* are used to save stack and frame pointers when a stack context is switched out. They are used by the routines *SwitchToExceptionDispatcherContext* and *SwitchToUserProgramContext*.

```
VAR
 PROGRAM_SP, PROGRAM_BP: STACKPTR;
 EH_STACK_SP, EH_STACK_BP: STACKPTR;
```

The *RAISE* entry point is called directly by the code generated by the MEHL compiler for the *RAISE exception.constructor(arg1,arg2,...,argn)* statement. *RAISE* expects the address of the exception's constructor in register bx and the arguments for the constructor followed by the address of the raise point on the stack. *RAISE* starts by retrieving the address of the raise point and by performing a context switch to the exception mechanism. Then execution is forwarded to the *RAISE_BODY* function that does the actual work.

```
CODE RAISE(bx: REGISTER);
VAR
 pc_addr: REGISTER;
BEGIN
 ASM POP pc_addr (* Retrieve PC addr at call point *)
 SwitchToExceptionDispatcherContext();
 RaiseBody(pc_addr, bx, RAISE_EXCEPTION_FLAG, 0);
END;
```

The *RAISE_RETURN* entry point is called directly by the code generated by the MEHL compiler for a guarded RETURN statement. *RAISE_RETURN* expects the address of the local return entry point in register bx and the address of the raise point on the stack. In addition, any return value associated with the RETURN statement is expected in register ax.

```
CODE RAISE_RETURN(bx: REGISTER; ax: REGISTER)
VAR
 pc_addr: REGISTER;
BEGIN
 ASM POP pc_addr (* Retrieve PC addr at call point *)
 SwitchToExceptionDispatcherContext();
 RaiseBody(pc_addr, bx, RAISE_RETURN_FLAG, ax);
END;
```

The *RERAISE* entry point is called directly by the code generated by the MEHL compiler for a RAISE statement with no parameters. *RERAISE* expects the address of the raise point on the stack. *RERAISE* starts by retrieving the address of the raise point, performing a context switch to the exception mechanism and checking for the legality of the re-raise operation. Then the *pc_addr* variable in the local environment for *RAISE_BODY* is altered and execution is transferred to the main loop inside the *RAISE_BODY* function.

```
CODE RERAISE();
VAR
 new_pc_addr: REGISTER;
BEGIN
 ASM POP new_pc_addr (* Retrieve PC addr at call point *)
 SwitchToExceptionDispatcherContext();
 IF (has no exception to re-raise) THEN
  Error("No exception to re-raise");
 pc_addr:= new_pc_addr; (* Set RAISE_BODY's pc_addr variable *)
 GOTO CONTINUE_ENTRY; (* Jump straight into RAISE_BODY *)
END;
```

The *CONTINUE_RAISE_FROM_FINALLY* entry point may be called directly by the code generated by the MEHL compiler at the end of the finally statements in a try-finally construction. *CONTINUE_RAISE_FROM_-FINALLY* expects the address of the call point on the stack.

```
CODE CONTINUE_RAISE_FROM_FINALLY();
VAR
 new_pc_addr: REGISTER;
BEGIN
 ASM POP new_pc_addr (* Retrieve PC addr at call point *)
 SwitchToExceptionDispatcherContext();
 finally_raise_nest_level:=save_finally_raise_nest_level;
 pc_addr:= new_pc_addr; (* Set RAISE_BODY's pc_addr variable *)
 GOTO CONTINUE_ENTRY; (* Jump straight into RAISE_BODY *)
END;
```

The *TERMINATE_EXCEPTION* entry point is called directly by the code generated by the MEHL compiler at the end of each exception handler. *TERMINATE_EXCEPTION* expects the resume address on the stack. *TERMINATE_EXCEPTION* starts by retrieving the address of the current exception object and by restoring the global finally_raise_nest_level variable using the local environment of *RAISE_BODY*. The default (virtual) destructor of the exception object is then called and the *RAISE_BODY* environment is discarded. Finally, execution is transferred to the resume address by an ordinary return instruction.

```
CODE TERMINATE_EXCEPTION();
VAR
 tmp: REGISTER;
BEGIN
 SwitchToExceptionDispatcherContext();
 tmp=exception_object_addr; (* From RAISE_BODY *)
 finally_raise_nest_level=save_finally_raise_nest_level; (* Ditto *)
 SwitchToUserProgramContext();
 Call default_exception_dtr(0, tmp);
 SwitchToExceptionDispatcherContext();
 ASM LEAVE (* Destroy RAISE_BODY function instance *)
 SwitchToUserProgramContext();
 ASM RET (* Jump to resume address *)
END;
```

The *EH_STUB* entry point is called directly by the code generated by the MEHL compiler when external functions are called. *TERMINATE_EXCEPTION* expects the address of the external function in register bx and the size of the arguments for the external function in register si. *TERMINATE_EXCEPTION* expects on the stack

an initial blank word followed by the arguments for the external function on the stack and the address of the call point. *TERMINATE_EXCEPTION* starts by moving the address of the call point to the initial word providing a single correct located return address (only one return address must be on top of the arguments for the external function). Then a location marker is pushed on the runtime stack structure for location markers, the external function is called and the location marker is popped.

```
CODE EH_STUB(BX: REGISTER; SI: REGISTER);
VAR
 pc_addr:CODEPTR;
 tmp: REGISTER;
BEGIN
 ASM POP pc_addr (* Retrieve PC addr at call point *)
 ASM MOV [SI+SP],pc_addr (* Place return addr for EH_STUB at
                           preallocated space located just
                           before args to external function *)
 PushLocationMarker(pc_addr,BP)
 ASM CALL BX
 PopLocationMarker();
 ASM RET; (* Normal return (to call point) *)
END;
```

The *EH_ALLOCATE* function is called directly by the code generated by the MEHL compiler at the beginning of constructers for exception object's. *EH_ALLOCATE* expects the size of the block on the stack. The function is used together with the function *EH_DEALLOCATE* for allocation and de-allocation respectively of global memory for exception objects. It is based on principles for the exception storage manager discussed in section 10.2.

```
FUNCTION EH_ALLOCATE(size: WORD): WORD;
VAR
 addr: WORD;
BEGIN
 IF eh_buffer_pos+size<@eh_buffer+EH_BUFFER_SIZE THEN
  (* Ok, not overflow - Store size in start of free memory block,
     adjust free memory block pointer and return address of not used
     part of memory block *)
  eh_buffer_pos^.size:=size; (* Store size in start of  mem *)
  addr:=eh_buffer_pos+2;(* First two bytes are used for size entry *)
  eh_buffer:=eh_buffer+2+size; (* Adjust free pointer *)
  RETURN addr;
 ELSE (* No room - Use general heap manager *)
  RETURN NEW(size);
 ENDIF;
END;
```

The *EH_DEALLOCATE* function is called directly by the code generated by the MEHL compiler at the end of destructors for exception object's. *EH_ALLOCATE* expects the address of the object on the stack.

```
PROCEDURE EH_DEALLOCATE(addr: WORD)
VAR
 node, end_node : ^EH_BUFFER_NODE;
BEGIN
 (* What kind of storage ? EH buffer memory or heap memory *)
 IF (addr>=@eh_buffer) AND (addr<@eh_buffer+EH_BUFFER_SIZE) THEN
  (* In range for EH bufffer : Is EH buffer memory *)
  node :=addr-2; (* Address is 2 bytes into the node *)
  end_node := addr+node^.size;
  IF end_node = eh_buffer_pos THEN
   ( Last memory block - remove *)
   eh_buffer_pos:=addr-2;
  ELSE
   (* Next to last memory block - Combine *)
```

```
   node^.size:=node^.size+(end_node^.size+2);
  ENDIF;
 ELSE (* Heap memory *)
  CALL DISPOSE(addr);
 ENDIF;
END;
```

RAISE_BODY is the main procedure in the EH runtime mechanism. It contains the main loop that locates handlers, validates and propagates exceptions and performs object cleanup. It also contains the local data for each exception.

RAISE _BODY starts by allocating and initializing its local data and by constructing the exception object (except for the special RETURN exceptions that have no associated exception object). Then the main loop is entered. The main loop starts by looking up the current PC address in the main program table.

If an entry for the PC address can be found in the main program table and the function is not a trivial one (a function table exists) then the following actions are taken:

1. The global information byte (also called procedure descriptor (pd)) is retrieved.
2. The stack pointer is reset to the correct BP relative position.
3. Memory for an eventual heap object is deallocated.
4. A handler for the current exception is searched for.
5. Exceptions for previous exception handlers exited by the current exception are destroyed.
6. If an exception handler was found control is transferred to that handler.
7. Propagation of the exception is validated.
8. Information about the destructible objects to destroy is retrieved.
9. Objects are destroyed.
10. It is checked whether the exception address is located in the constructor of a heap object.
11. The exception propagates.

If an entry for the PC address can't be found in the main program table the following actions are taken:

1. If the exception has occurred in the construction or destruction of an array of objects, relevant state information is retrieved and the exception is propagated.
2. If the exception address is located in the call sequence of the initialisation/deinitialisation functions for the different modules (including the program module) of a program the deinitialisation functions for the initialised modules are called and an error is issued.
3. If a location marker can be found the exception is propagated to the point specified by the location marker, otherwise an error is issued.

```
PROCEDURE RAISE_BODY(pc_addr: CODEPTR;
                     exception_ctr: CODEPTR / return_addr: CODEPTR;
                     raise_kind: EXCEPTION_TYPE; user_ax: WORD);
VAR
 ptp: POINTER TO PROGRAM_TABLE_ENTRY; (* function table pointer *)
 lmp: POINTER TO LOCATION_MARKER;
 hpc: CODEPTR;

 info: POINTER;
 pd: BYTE; (* GENERAL_INFORMATION_BYTE *)

 handler_type: HANDLER_TYPE;

 dtr_count: WORD; (* # objects to destroy before propagation *)
 first_dtr: DATAPTR; (* Pointer to first destructor entry in table *)

 exception_object_addr: DATAPTR;

 except_nest_level: WORD = -1;
 finally_nest_level: WORD;
 save_finally_raise_nest_level: WORD = finally_raise_nest_level;
 last_heap_this : DATAPTR = 0;

 user_bp: STACKPTR;
```

```
BEGIN
 IF raise_kind=RAISE_EXCEPTION THEN
  exception_object_addr:=ctr_exception_object();

 CONTINUE_ENTRY: LOOP
  ptp:=FindFunction(pc_addr);
  IF ptp THEN (* Function found in main program table ? *)
  BEGIN
   info=ptp.info_ptr;

   IF info = -1 THEN pd:=0; ELSE (* Function table found ? *)
   BEGIN
    pd:= info^; (* Get procedure descriptor (general info byte) *)
    INC(info);

    SetUserProgramSPToFrame(ptp.begin_range);

    IF last_heap_this THEN
     DisposeHeapMemory(last_heap_this,info);

    handler_type:=FindHandler(pd, info, raise_kind,
                              exception_object_addr,
                              pc_addr, except_nest_level,
                              finally_nest_level, hpc);

    TerminatePreviousExceptions(user_bp, except_nest_level,
                                hpc, save_finally_nest_level);

    IF handler_type<>NO_HANDLER THEN
     ExecuteHandler(pd, handler_type, hpc,
                    except_nest_level, finally_nest_level,
                    exception_object_addr,user_bp);
    ENDIF;

    IF raise_kind = RAISE_RETURN_FLAG THEN
     ReturnControl(pd, return_addr, user_ax);
    ENDIF;

    IF GetDtrDestructInfo(pd, info, pc, within_arrray_apply,
                          dtr_count, first_dtr) THEN
      DestroyObjects(pd, first_dtr, dtr_count, within_array_apply,
                     array_count, array_addr);
    ENDIF;

    IF raise_kind = RAISE_EXCEPTION_FLAG AND
       (NOT PropgationAllowed(pd,info,exception_object_addr)) THEN
       Error("Propagtion of exception not allowed");

    (* In constructor of heap object *)
    IF GetProcType(pd)=CONSTRUCTOR THEN
     last_heap_this=CheckForHeapObject();

    pc_addr:= HijackReturnPCAndUnwind();
   END;
  ELSE  (* Function not found in program table *)
   IF pc_addr = ARRAY_CTR_APPLY_CALL THEN
    MarkWithinArrayCtrApply(array_addr, array_count,
                            within_array_apply);
    pc_addr:= HijackReturnPCAndUnwind();
   ELSE IF pc_addr = ARRAY_DTR_APPLY_CALL THEN
    MarkWithinArrayDtrApply(array_addr, array_count,
                            within_array_apply);
    pc_addr:= HijackReturnPCAndUnwind();
   ELSE IF pc_addr in main programs INIT-EXIT range THEN
    CleanupModules(pc_addr);
    Error("Unhandled exception");
   ELSE (* Completely unknown pc, must be an external function *)
    lmp:=PopLocationMarker();
    IF NOT lmp THEN (* Location marker found *)
     Error("Exception propagated to unknown location");
    pc_addr:=GetReturnPCAndUnWind(lmp);
   ENDIF;
  ENDIF;
 END LOOP;
END; (* RAISE_BODY *)
```

The function *FindFunction* uses binary search of the main program table in order to find a MEHL function with an address span which matches a given PC value. If a MEHL function is found it returns the address of the program table entry, otherwise it returns *FUNCTION_NOT_FOUND*..

```
FUNCTION FindFunction(pc: CODEPTR): POINTER TO FUNCTION_TABLE;
VAR
 start, end, mid : POINTER TO PROGRAM_TABLE_ENTRY;
BEGIN
 start:=FIRST_PROGRAM_TABLE_ENTRY;
 end:=LAST_PROGRAM_TABLE_ENTRY;
 REPEAT
  mid:=(start+end)/2;
  IF (start+end)%SIZEOF(PROGRAM_TABLE_ENTRY)<>0 THEN
   (* mid between two entries - Afjust *)
   mid:=mid-SIZEOF(PROGRAM_TABLE_ENTRY)/2;
  ENDIF;
  IF mid^.begin_range<pc THEN (* Below ? *)
   end:=mid-1;
  ELSE IF (mid+1)^.begin_range<=pc THEN (* Not below - Above ? *)
   (* Entry found - Check that it is not an end marker. *)
   IF mid^.info_ptr!=END_MARKER_VALUE THEN
    RETURN mid;
   ELSE
    RETURN FUNCTION_NOT_FOUND;
   ENDIF;
  ELSE (* Above *)
   start:=mid+1;
  END;
 UNTIL start>=end;
 RETURN FUNCTION_NOT_FOUND;
END;
```

The function *GetDtrDestructInfo* is used to locate the objects which must be destroyed before propagating an exception out of the current MEHL function. It does that by comparing the PC value for propagation with the function table's list of addresses for constructors and destructors for objects. Note that since the return address for calls to destructors is not specified directly in the static function table a code inspection technique is used to calculate it (the start address for each destruction is known but not the address after the each destruction which is the return address for the destructor)

```
FUNCTION GetDtrDestructInfo(pd:BYTE; info: POINTER;
                            pc_addr: CODEPTR;
                            within_arrray_apply: BOOLEAN;
                            VAR dtr_count:WORD;
                            VAR first_dtr: DATAPTR): BOOLEAN;
VAR
 object_count, i: WORD;
 first_ctr, ctr, dtr: POINTER TO WORD;
 code : CODEPOINTER TO BYTE;
BEGIN
 object_count:=GetObjectCount(pd,info);
 IF object_count>0 THEN
  first_ctr:=info;
  info:=info+object_count*4;
  ctr:=first_ctr;
  first_dtr:=ctr+object_count*2;
  dtr:=first_dtr;
  (* Check for exception in a constructor *)
  FOR i:=0 TO object_count-1 DO
    IF ctr^ = pc_addr THEN
      (* Exception in constructor *)
      dtr_count:=i;
      IF within_array_apply THEN
        (* Always include array in types to be destructed *)
```

```
        dtr_count:=dtr_count+1;
      ENDIF;
      first_dtr:=first_dtr+(object_count-dtr_count)*2;
      RETURN TRUE;
    ENDIF;
    ctr:=ctr+2; (* Next constructor address in function table *)
  END;
  (* Check for exception in a destructor *)
  FOR i:=1 TO object_count DO
   IF dtr^ THEN (* Null entry ? *)
    IF pc_addr>dtr^ THEN
      (* Inspect code in order to get the return address *)
      code:=dtr^;
      CASE instruction at code pointer OF (* Drop push THIS code *)
       LEA with 16bit displacement : code:=code+5;
       LEA with 8bit displacement : code:=code+4;
       PUSH 16bit immediate : code:=code+2;
       PUSH register : code:=code+1;
      END;
      IF instruction at code pointer = PUSH 16bit immediate THEN
       (* Skip additional code for destruction of ARRAY *)
        code:=code+2; (* Drop dtr addr, obj size, element count *)
        DO TWO TIMES
          CASE instruction at code pointer OF
           PUSH 16 bit immediate : code:=code+3;
           PUSH  8 bit immediate : code:=code+2;
          END;
        END;
      ENDIF;
      code:=code+3; (* Drop call instruction *)
      (* Now code is equal to the return address located just
         after the call-instructuon for the destructor. And
         now the return address can be compared with pc_addr*)
      IF code=pc_addr THEN
        (* Exception in destructor *)
        dtr_count=object_count-i;
        first_dtr:=dtr+2;
        IF within_array_apply THEN
          (* Always include array in types to be destructed *)
          first_dtr:=first_dtr-2;
          dtr_count:=dtr_count+1;
        ENDIF;
      ENDIF;
    ELSE (* Not possibly in range *)
     RETURN FALSE;
    END
   END;
   dtr:=dtr+2; (* Next destructor address in function table *)
  END;
 END;
 RETURN FALSE;
END;
```

The procedure *DestroyObjects* is used to destroyed the objects identified by *GetDtrDestructInfo*. When destroying array of objects it uses the procedure *EH_ARRAY_DTR_APPLY* which is similar to the *ARRAY_DTR_APPLY* procedure, except that it is placed in the code space for the EH mechanism.

```
PROCEDURE DestroyObjects(pd:BYTE: first_dtr:DATAPTR; dtr_count:WORD;
                         VAR within_array_apply: BOOLEAN;
                         array_count: WORD; array_addr: DATAPTR);
VAR
 code : CODEPOINTER TO BYTE;
 bp_relative: BOOLEAN;
 address : WORD;
 address_type : ADDRESS_TYPE;
```

```
 dtr_addr: CODEPTR;
 obj_size: WORD;
 count: WORD;

 tmp1, tmp2, tmp3, tmp4, tmp5: REGISTER;

BEGIN
 REPEAT
  code:=first_dtr^;
  IF code THEN
   IF instruction at code pointer = LEA THEN
     (* Destructor for local object or member-object.
        The displacement argument to the LEA instruction is bp
        relative for local objects and relative to THIS for sub-
        objects. What is the case for this object ? *)
     IF argument to LEA instruction is BP relative THEN
      address_type=BP_RELATIVE
     ELSE
      address_type=THIS_RELATIVE
     END;
     CASE size of displacement for LEA instruction OF
       16 bit immediate : address=(POINTER TO WORD (code+2))^;
                          code:=code+5;(* Skip lea+push *)
        8 bit immediate : address=(code+2)^;
                          code:=code+4;(* Skip lea+push *)
     END;
   ELSE IF instruction at code pointer = PUSH 16bit immediate THEN
     (* Destructor for a global object *)
     address=(POINTER TO WORD (code+2))^;
     code:=code+3;
     address_type=ABSOLUTE; (* Global object *)
   ELSE (* instruction at code pointer = PUSH 16bit register *)
     (* Destructor for ancestor object *)
     address=0;
     address_type=THIS_RELATIVE;
     code:=code+1;
   ENDIF;

   IF instruction at code pointer = PUSH 16bit immediate THEN
    (* Array of objects *)

    dtr_addr:=(POINTER TO WORD (code+1))^;
    code:=code+3;

    CASE instruction at code pointer OF
     PUSH 16 bit immediate : obj_size:=(POINTER TO WORD (code+1))^;
                             code:=code+3;
     PUSH  8 bit immediate : obj_size:=(code+1)^;
                             code:=code+2;
    END;

    CASE instruction at code pointer OF
     PUSH 16 bit immediate : count:=(POINTER TO WORD (code+1))^;
                             code:=code+3;
     PUSH  8 bit immediate : count:=(code+1)^;
                             code:=code+2;
    END;

    IF within_array_apply THEN
     address:=array_addr;
     count:=array_count;
     within_array_apply:=FALSE;
    END;

    tmp1:=address;
    tmp2:=address_type;
    tmp3:=dtr_addr;
    tmp4:=obj_size;
    tmp5:=count;

    SwitchToUserProgramContext();
    ASM PUSH 1
    IF tmp2=BP_RELATIVE THEN
     ASM PUSH tmp1+BP;
```

```
    ELSE if tmp2=THIS_RELATIVE THEN
     ASM PUSH tmp1+[BP+THIS_REG_REL_OFS]
    ELSE
     ASM PUSH tmp1
    ENDIF
    ASM PUSH tmp3
    ASM PUSH tmp4
    ASM PUSH tmp5
    CALL EH_ARRAY_DTR_APPLY
    SwitchToUserProgramContext();
   ELSE (* Ordinary object *)
    (* Retrieve address of destructor for ordinary object from
       pc-relative argument to the call instruction *)
    dtr_addr:=code+(POINTER TO WORD (code+1))^;

    tmp1:=address;
    tmp2:=address_type;
    tmp3:=dtr_addr;

    SwitchToUserProgramContext();
    ASM PUSH 1
    IF tmp2=BP_RELATIVE THEN
     ASM PUSH tmp1+BP;
    ELSE if tmp2=THIS_RELATIVE THEN
     ASM PUSH tmp1+[BP+THIS_REG_REL_OFS]
    ELSE
     ASM PUSH tmp1
    ENDIF
    ASM CALL tmp3
    SwitchToUserProgramContext();
   END;

  END;
  first_dtr:=first_dtr+2;
  DEC(dtr_count);
 UNTIL dtr_count=0;
END;
```

The function *FindHandler* is used to find an exception handler for a MEHL function. *FindHandler* checks the static information in the function table for an exception handler which guards the right PC range and which, for TRY-EXCEPT handlers, is of the correct exception type. The function expects that the *info* parameter points to the start of the information for exception handlers, if information exist. The static information for exception handlers has been designed for compactness and not for easy lookup. It is recommend to keep figure 27 close in view while inspecting *FindHandler*.

```
FUNCTION FindHandler(pd:BYTE; info: POINTER;
                     raise_type: EXCEPTION_TYPE;
                     exception_object_addr: DATAPTR;
                     pc: CODEPTR;
                     VAR except_nest_level: WORD;
                     VAR finally_nest_level: WORD;
                     VAR pc_addr: CODEPTR): BOOLEAN;

VAR
 try_count: BYTE;
 regions_count: BYTE;
 handlers_count: BYTE;
 exceptions_in_handler_count: BYTE;
 region : POINTER TO REGION;

BEGIN
 IF HasTryConstructions(pd)THEN
  try_count:=info^;
  INC(info);
  REPEAT
    regions_counter:=info^;
    region:=info+1;
    REPEAT
      IF pc is in range of (region^.start , region^.end) THEN
       info:=start of description of exception handlers for region.
```

```
        except_nest_level:=((POINTER TO WORD)info)^;
        handlers_count:=info^;(* Get Except(#handlers)/Finally spec *)
        INC(info);
        IF raise_type=EXCEPTION_TYPE AND handlers_count>0 THEN
          (* TRY-EXCEPT *)
          hpc:=(POINTER TO WORD info)^; (* Get addr of handler *)
          info:=info+2; (* Drop word *)
          UnMarkExceptionObjectType(exception_object_addr);
          REPEAT
            exceptions_in_handler_count:=info^;
            INC(info);
            REPEAT
              (* Matching exception handler ? *)
              IF IsRTTIforExceptionTypeMarked(info) THEN
               UnMarkExceptionObjectType(exception_object_addr);
               RETURN EXCEPT; (* Success *)
              ENDIF;
              info:=info+2; (* Drop word *)
              DEC(exceptions_in_handler_count);
           UNTIL exceptions_in_handler_count=0;
           DEC handlers_count;
          UNTIL handlers_count=0;
          UnMarkExceptionObjectType(exception_object_addr);
        ELSE IF (handlers_count=0) THEN
          (* TRY-FINALLY handler found *)
          finally_nest_level:=info^;
          hpc:=(POINTER TO WORD info+1)^; (* Get handler addr *)
          RETURN FINALLY;
        ENDIF;
       ENDIF;
       INC(region);
       DEC(regions_counter);
     UNTIL regions_counter=0;
     DEC(try_count);
   UNTIL try_count=0;
   info:=Location after data for exception regions/handlers.
 END;
 RETURN NO_HANDLER;
END;
```

The procedure *PropagationAllowed* checks the raises-set specification for a MEHL function and returns a flag that tells whether or not propagation is allowed. The procedure expects that the *info* parameter points to the start of the raises-set, if one is specified.

```
FUNCTION PropagationAllowed(pd:BYTE; info: POINTER;
                            exception_object_addr:DATAPTR): BOOLEAN;
BEGIN
 IF GetRaisesSpec(pd)=ANY THEN (* Consult general info byte *)
   RETURN TRUE;
 ELSE IF GetRaisesSpec(pd)=RAISES_SET THEN (* Ditto *)
   (* Raises-set specified *)
   MarkExceptionObjectType(exception_object_addr);
   FOR EACH exception object e in raises-set specified at info DO
     IF IsExceptionObjectMarked(e) THEN
       UnMarkExceptionObjectType(exception_object_addr);
       RETURN TRUE;
     ENDIF;
   END;
   UnMarkExceptionObjectType(exception_object_addr);
 ENDIF;
 RETURN FALSE;
END;
```

The procedure *MarkExceptionObjectType* and*UnMarkExceptionObjectType* marks and unmarks exception objects of a specified type. With marks active the function *IsExceptionObjectMarked* can be used to tell whether the marked exception type(s) is derived from the exception type tested for.

```
PROCEDURE MarkExceptionObjectType(exception_object_addr: DATAPTR);
VAR
 exception_obj : POINTER TO EXCEPTION;
 rtti : POINTER TO END OF RTTI_DATA;
BEGIN
 exception:=exception_object_addr;
 rtti:=exception^.vmt_ptr;
 DO (* Mark object type and object type's ancestors *)
  rtti^.id_mark:=MARKED;
  rtti:=rtti^.ancestor;
 WHILE rtti;
END;

PROCEDURE UnMarkExceptionObjectType(exception_object_addr: DATAPTR);
VAR
 exception_obj : POINTER TO EXCEPTION;
 rtti : POINTER TO END OF RTTI_DATA;
BEGIN
 exception:=exception_object_addr;
 rtti:=exception^.vmt_ptr;
 DO (* Remove mark from object type and object type's ancestors *)
  rtti^.id_mark:=UNMARKED;
  rtti:=rtti^.ancestor;
 WHILE rtti;
END;

FUNCTION IsExceptionObjectMarked(exception_object_addr: DATAPTR)
                                : BOOLEAN;
VAR
 exception_obj : POINTER TO EXCEPTION;
BEGIN
 exception_obj:=exception_object_addr;
 (* Return true if the exception object specified by the
    exception_object_addr parameter is a supertype for
    the marked exception object-type(s) *)
 RETURN IsRTTIforExceptionTypeMarked(exception_obj^.vmt_ptr);
END;

FUNCTION IsRTTIforExceptionTypeMarked(t: POINTER TO END OF RTTI_DATA)
                                     : BOOLEAN;
BEGIN
 RETURN id_mark=MARKED;
END;
```

The procedure *CleanupModules* performs cleanup of global destructible objects for the different modules in a MEHL program. The procedure is called in the event of an unhandled exception before terminating the program with an error. The procedure compares a PC value with the range of initialisation and deinitialisation calls in the start-up code for the program and calls the deinitialisation procedures for the modules which have been initialised (Then each deinitialisation procedure called takes care of destruction of the objects for the module). The procedure uses a code inspection technique which retrieves the individual addresses of the deinitialisation procedures to call from the arguments of the CALL instructions.

```
PROCEDURE CleanupModules(pc_addr: CODEPTR);
VAR
 p: CODEPTR;
BEGIN
 p:=EXIT-(INIT+3);
 IF p THEN (* init / deinit calls present ? *)
  IF (pc_addr>p/2+INIT+3) THEN (* in deinit call *)
   p:=pc_addr; (* issue all subsequent deinit calls *)
```

```
  ELSE (* in init call *)
   (* issue deinit calls for corresponding previous init calls *)
   p:=INIT+EXIT+3-pc_addr;
  ENDIF;
  WHILE (p<EXIT) THEN
   (* Get addr of deinit runtine from operand to
      CALL instruction in code and call the addr *)
   call deinit function at address p+3+(p+1)^;
   p:=p+3;
  END;
 ENDIF;
END;
```

The procedures *MarkWithinArrayCtrApply* and *MarkWithinArrayDtrApply* are used to mark that an exception has propagated out while constructing or destroying an array of objects and to calculate the elements which must be destroy by the EH mechanism. Both procedures calculate a starting address and element count for the sub-array of objects that must be destroyed. For this the procedures inspect the local environment for the helper procedures *ARRAY_DTR_APPLY* and *ARRAY_CTR_APPLY* (See description of these).

```
PROCEDURE MarkWithinArrayCtrApply(VAR array_addr: DATAPTR;
                                  VAR array_count: WORD;
                                  VAR flag: BOOLEAN)
VAR
 tmp,tmp2: REGISTER;
BEGIN
 SwitchToUserProgramContext();
 tmp:=ACA_count-ACA_icount; (* tmp = objects to destroy *)
 tmp2:=ACA_addr-ACA_size; (* tmp2 = addr of first obj to destroy *)
 SwitchToExceptionDispatcherContext();
 array_count:=tmp;
 array_addr:=tmp2;
 flag:=TRUE;
END;

PROCEDURE MarkWithinArrayDtrApply(VAR array_addr: DATAPTR;
                                  VAR array_count: WORD;
                                  VAR flag: BOOLEAN)
VAR
 tmp,tmp2: REGISTER;
BEGIN
 SwitchToUserProgramContext();
 tmp:=ADA_icount-1; (* tmp = objects left to destroy. *)
 tmp2:=ADA_addr-ADA_size; (* tmp2 = addr of first obj to destroy *)
 SwitchToExceptionDispatcherContext();
 array_count:=tmp;
 array_addr:=tmp2;
 flag:=TRUE;
END;
```

The function *CtrExceptionObject* is used at the start of the raise operation to construct the exception object. The parameters to the constructor are located on the user’s stack. The address of the newly created exception object is returned.

```
FUNCTION CtrExceptionObject(): DATAPTR;
VAR
 tmp: REGISTER;
BEGIN
 SwitchToUserProgramContext();
 tmp=Call exception_ctr(0);
 SwitchToExceptionDispatcherContext();
 IF tmp = NUL THEN
  Error(“Exception object could not be allocated”);
 RETURN tmp;
END;
```

The procedure *TerminatePreviousExceptions* is used to check for and terminate earlier exceptions for which the handling is terminated by an exception. Both the exception object and the *RAISE_BODY* frame located in the EH mechanism's stack must be destroyed. Memory for the exception object is destroyed by the destructor, the *RAISE_BODY* frame is not actually destroyed but only combined with the current (topmost) frame for the active exception. When checking for earlier exceptions which must be terminated is uses the technique described in section 8.2.

```
PROCEDURE TerminatePreviousExceptions(user_bp: STACKPTR;
          except_nest_level: WORD; hpc: CODEPTR;
          save_finally_nest_level: WORD);
VAR
 tmp: REGISTER;
BEGIN
 WHILE (previous RaiseBody activation record (PA) exist) DO
 BEGIN (* Exiting previous except/finally handler ? *)
  IF PA:user_bp = user_bp AND
     ((NOT hpc) OR except_nest_level<=PA:except_nest_level ) THEN
  BEGIN (* Succes- Same function instance,exiting previous handler *)
   finally_raise_nest_level:=save_finally_raise_nest_level;
   tmp:= PA:exception_object_addr;
   Delete PA by a combine operation.
   SwitchToUserProgramContext();
   Call default_exception_dtr(0, tmp);
   SwitchToExceptionDispatcherContext();
  ENDIF;
 END;
END;
```

The routine *ExecuteHandler* is used to transfer control from the EH mechanism to an exception handler in the user program. If the handler is located in a method the routine reads the address of the object from the user's stack frame as discussed in section 12.7 and initialises register SI, the *THIS* register used internally by the MEHL compiler. If the handler is a finally handler it sets the global *finally_raise_nest_level* flag to the nest level of the finally handler. Otherwise, if the handler is an ordinary exception handler, the address of the exception object is stored in the exception reference variable associated with the handler using the technique discussed in section 12.7 (the address will be stored in a dummy word if the handler has no argument). Finally execution is transferred to the exception handler using a jump instruction.

```
PROCEDURE ExecuteHandler(pd: BYTE; handler_type: HANDLER_TYPE;
                         hpc: CODEPTR; except_nest_level: WORD;
                         finally_nest_level: WORD;
                         exception_object_addr: DATAPTR;
                         VAR user_bp: STACKPTR)
VAR
 tmp,tmp2,tmp3: REGISTER;
BEGIN
 tmp:= is_method(pd);
 user_bp:=GetUserProgramBP(); (* Save user's BP *)
 IF handler_type = EXCEPT THEN
  tmp2:= except_nest_level;
  tmp3:= exception_object_addr;
  SwitchToUserProgramContext();
  (* Store Exception object addr in exception var *)
  ASM MOV [BP+EXCEPTION_VAR_MUL_FOR_EXCEPT_LEVEL*tmp2],tmp3
 ELSE (* handler_type = FINALLY *)
  finally_raise_nest_level=finally_nest_level;
  SwitchToUserProgramContext();
 ENDIF;
 IF tmp THEN (* If method retrieve this addr & set register var *)
  ASM MOV THIS_REG, [BP+THIS_REG_REL_OFS]
 ASM JMP tmp2 (* Transfer control to handler *)
END;
```

The function *CheckForHeapObject* is used to find out whether the address of an object belongs to the heap memory. For this it uses the non-standard heap management routine called *IS_HEAP_MEMORY*.

```
FUNCTION CheckForHeapObject(): DATAPTR;
BEGIN
 SwitchToUserProgramContext();
 ASM MOV tmp, [BP+THIS_REG_REL_OFS] (* tmp = this addr for object *)
 SwitchToExceptionDispatcherContext();
 IF IS_HEAP_MEMORY(tmp) THEN
  RETURN tmp; (* Signal for later *)
 ELSE RETURN 0;
END;
```

The procedure *DisposeHeapMemory* disposes of heap memory if a constructor for a heap memory object have just been exited (by inspecting that *last_heap_this* is non-zero) and if the constructor that was exited by an exception is not an constructor for a sub-object, local object or global object (all of which is implicitly constructed with the construction point stored in the function table).

```
PROCEDURE DisposeHeapMemory(VAR last_heap_this:DATAPTR;
                            info: DATAPTR);
BEGIN
 IF last_heap_this AND (pc_addr not found among implicit
    constructor calls stored at the function table located
    at info) THEN
  BEGIN
   DISPOSE(last_heap_this); (* Dispose of memory for heap object *)
  ENDIF;
 last_heap_this:=0;
END;
```

The routine *ReturnControl* is used by RETURN exceptions to transfer control from the EH mechanism to the user program. If the destination is located in a method the routine reads the address of the object from the user's stack frame as discussed in section 12.7 and initialises register SI, the *THIS* register used internally by the MEHL compiler. The value of the user's register AX, used for function results by the MEHL compiler, is restored to its original value before returning.

```
CODE ReturnControl(pd: BYTE; dest: CODEPTR; user_ax: WORD)
VAR
 tmp, tmp2, tmp3: REGISTER;
BEGIN
 tmp:= is_method(pd);
 tmp2:= dest;
 tmp3:= user_ax;
 ASM LEAVE (* Destroy RAISE_BODY function instance *)
 SwitchToUserProgramContext();
 IF tmp THEN (* If method retrieve this addr & set THIS register *)
  ASM MOV THIS_REG, [BP+THIS_REG_REL_OFS]
 ASM MOV AX,tmp3
 ASM JMP tmp2
END;
```

The function *HijackReturnPCAndUnwind* retrieves the return address and unwinds the top frame from the user's stack. Before returning the caller's PC value (the return address) *HijackReturnPCAndUnwind* validates that the caller's PC is not within the PC range of the EH mechanism. If that is so the constructor or destructor for an exception object has attempted to exit by raising an exception and a fatal error message is issued.

```
FUNCTION HijackReturnPCAndUnwind(): CODEPTR;
VAR
 pc_addr: REGISTER;
BEGIN
 SwitchToUserProgramContext();
```

```
 ASM LEAVE (* Remove local vars, set BP to previous function frame *)
 ASM POP pc_addr (* Retrieve PC addr from call point *)
 SwitchToExceptionDispatcherContext();
 IF pc_addr within pc range of runtime exception dispatcher THEN
  Error("Unhandled exception during exception handling");
 RETURN pc_addr;
END;
```

The function *GetReturnPCAndUnwind* inspects the information stored in a location marker and uses that information to restore the user's stack frame pointer to the value it had when the location marker was created. The function returns the PC value for the call to the external function that was responsible for the creation of the location marker (The PC value to be used for further propagation).

```
FUNCTION GetReturnPCAndUnwind(lmp: POINTER TO LOCATION_MARKER)
                             : CODEPTR;
VAR
 tmp, tmp2: REGISTER;
BEGIN
 tmp:= lmp^.saved_BP;
 SwitchToUserProgramContext();
 ASM MOV BP,tmp
 SwitchToExceptionDispatcherContext();
 RETURN lmp^.call_point;
END;
```

The procedure *SetUserProgramSPToFrame* sets the value of the user's stack pointer using the frame pointer and the size of the local data for the current frame. The procedure uses the code inspection technique discussed in section 12.5 in which the size of the local data is retrieved from the data size argument of the ENTER instruction at the start of a function.

```
PROCEDURE SetUserProgramSPToFrame(func_start: CODEPTR);
VAR
 tmp: REGISTER;
BEGIN
 tmp:=func_start+1; (* tmp = addr ENTER operand (local data size) *)
 SwitchToUserProgramContext();
 ASM MOV SP,BP-[tmp] (* SP = BP - size of local data *)
 SwitchToExceptionDispatcherContext();
END;
```

The routines *PushLocationMarker* and *PopLocationMarker* are used to push and pop a location marker to/from the EH mechanism's list of location markers. *PushLocationMarker* generates an error in case of overflow, *PopLocationMarker* returns zero when failed.

```
CODE PushLocationMarker(pc_addr:CODEPTR; bp_value:STACKPTR)
BEGIN
 IF info_markers_pos>=MAX_INFORMATION_MARKERS THEN
  Error ("Overflow. Can't call external function");
 ELSE
 INC(info_markers_pos);
 info_markers_buf[info_markers_pos]^.saved_BP:=bp_value;
 info_markers_buf[info_markers_pos]^.call_point:=pc_addr;
END;

CODE PopLocationMarker(pc_addr:CODEPTR; bp_value:STACKPTR)
                      : POINTER TO LOCATION_MARKER;
VAR
 tmp: POINTER TO LOCATION_MARKER;
BEGIN
 IF info_markers_pos>0 THEN
   tmp:=info_markers_buf[info_markers_pos];
   DEC(info_markers_pos);
   RETURN tmp;
```

```
 ELSE
   RETURN 0;
 ENDIF;
END;
```

The routines *SwitchToExceptionDispatcherContext* and *SwitchToUserProgramContext* are used to switch between the stacks for the MEHL program and the dispatcher of exceptions (the EH mechanism).

```
CODE SwitchToExceptionDispatcherContext()
BEGIN
 PROGRAM_SP:=SP;
 PROGRAM_BP:=BP;
 SP:=EH_STACK_SP;
 BP:=EH_STACK_BP;
END;

CODE SwitchToUserProgramContext();
BEGIN
 EH_STACK_SP:=SP;
 EH_STACK_BP:=BP;
 SP:=PROGRAM_SP;
 BP:=PROGRAM_BP;
END;
```

The function *get_object_count* returns the number of destructible objects associated with the current frame. Note that the parameter *info* must point to the word after the general information byte for this function to work properly.

```
FUNCTION GetObjectCount(pd: BYTE; VAR info: POINTER): WORD;

VAR
 count: WORD;

BEGIN
 IF NOT (bits 3-5 of pd = 111B) THEN
  RETURN (value of bits 3-5 of pd); (* 0-6 objects *)
 ELSE (* Overflow, get count from additional word *)
  count:=info^;
  info:=info+2; (* Drop word *)
  RETURN count;
 ENDIF;
END;
```

The function *GetProcType* returns the type of the current MEHL function.

```
FUNCTION GetProcType(pd:BYTE): PROC_TYPE;

BEGIN
 CASE bits 0-1 of pd OF
  WHEN 00B : RETURN FUNCTION; (* PROCEDURE / FUNCTION *)
  WHEN 01B : RETURN METHOD;
  WHEN 10B : RETURN CONSTRUCTOR;
  WHEN 11B : RETURN DESTRUCTOR;
 END;
END;
```

The function *HasTryConstructions* is used to check for try constructions in the current MEHL function. If the function returns true the caller must search the function table for a matching exception handler.

```
FUNCTION HasTryConstructions(pd:BYTE): BOOLEAN;

BEGIN
 RETURN (bit 2 of pd)=1; (* Has try's ? *)
END;
```

The function *GetRaisesSpec* returns information about the raises-set specification for the current MEHL function. If *RAISES_SET* is returned the caller must check the raises-set placed at the end of the function table.

```
FUNCTION GetRaisesSpec(pd:BYTE): RAISES_SPEC;

BEGIN
 CASE bits 0-1 of pd OF
  WHEN 00B : RETURN ANY;
  WHEN 01B : RETURN NONE;
  WHEN 10B : RETURN RAISES_SET;
 END;
END;
```

Though not strictly a part of the EH runtime mechanism the *ARRAY_CTR_APPLY* and *ARRAY_DTR_APPLY* helper routines used to construct and destroy array of objects are of considerable importance for the EH mechanism. When an exception is propagated out of a constructor or destructor for an object which is an array element the propagation will fall through these helper routines at the PC call points *ARRAY_CTR_APPLY_CALL* and *ARRAY_DTR_APPLY_CALL* respectively. As an exception falls through these helper routines the local environment in the user-program context is inspected in order to determine the array elements to destroy (see *MarkWithinArray?trApply*).

```
PROCEDURE ARRAY_CTR_APPLY(ACA_base_addr: DATAPTR; ACA_dtr:CODEPTR;
                          ACA_size: WORD; ACA_count: WORD)
VAR
 ACA_addr: DATAPTR = ACA_base_addr;
 ACA_icount: WORD = ACA_count;

BEGIN
 REPEAT
  ARRAY_CTR_APPLY_CALL: Call ACA_dtr(1, ACA_addr);
  ACA_addr:=ACA_addr+ACA_size;
  DEC ACA_icount;
 UNTIL ACA_icound=0;
END;

PROCEDURE ARRAY_DTR_APPLY(ADA_base_addr: DATAPTR; ADA_dtr:CODEPTR;
                          ADA_size: WORD; ADA_count: WORD)
VAR
 ADA_addr: DATAPTR = ADA_base_addr+(ADA_count-1)*ADA_size;
 ADA_icount: WORD = ADA_count;

BEGIN
 REPEAT
  ARRAY_DTR_APPLY_CALL: Call ADA_dtr(1, ADA_addr);
  ADA_addr:=ADA_addr-ADA_size;
  DEC ADA_icount;
 UNTIL ADA_icound=0;
END;
```

14. The System Design of the MEHL compiler

In this chapter the overall system design of the MEHL compiler will be presented. As this thesis is about handling exceptions and not about compiler design the coverage will in general be limited to the support for EH and objects.

14.1 The structure of the compiler

For the MEHL compiler a normal syntax-directed translation scheme has been chosen. As shown in figure 37 the compiler is a two pass compiler with a program analyser pass followed by a code generator pass.

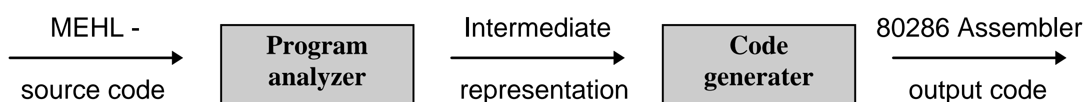


Figure 37 The flow in the compiler

The program analyser itself consists of two steps, a *scanner* and a *parser*. The *scanner* transforms the MEHL-source code into tokens, which are in turn passed on to the *parser,* which groups them into constructions as specified by the language's context-free grammar and then calls the associated semantic routines. When a syntactic structure is recognised by the parser, the semantic routines check the static semantics of the structure and initiate the synthesis task by generating an intermediate representation (IR) of the structure. In the IR constants are folded and common subexpressions are eliminated.

Instead of an ordinary machine-language program in binary form a more comprehensive symbolic 80286 assembler language has been targeted as the MEHL compiler output. Thus, the task of the code generator is to transform the shared IR into a symbolic assembler language. For more information about the target assembler language see Appendix F.

14.2 Compilation of multiple modules

A MEHL program can consist of several modules : One *program*-module and several *unit* modules, all of which must be compiled before a program can be assembled, linked and run..

The filenames of all modules in a program must be given as arguments to the MEHL compiler, which will deduce the correct compilation sequence and compile all interfaces, followed by a compilation of the program modules and all outdated or not yet compiled implementations.

14.3 The scanner.

The scanner recognises each of the reserved words of MEHL as distinct tokens. Numbers , identifiers and quoted text strings are tokenised into the tokens NUMBER, ID and STRING respectively.

14.4 The parser

The parser is constructed on the basis of an extended LALR(1) grammar of the MEHL programming language. As opposed to a formal grammar the tokens in this grammar can have a semantic value, and the groupings can have both associated semantic actions and a semantic value. The work of the semantic actions is divided into three parts: semantic checks, semantic returns and plain semantic actions.

Semantic checks are for checking the validity of language constructions such as type-correctness and examining if identifiers have been declared etc. Semantic returns are specified when a semantic value is needed in other productions. Finally, plain semantic actions are responsible for doing the actual work of creating the intermediate representation of the compiled program.

The semantic actions have as a major responsibility the task of generating an intermediate representation (IR) of the MEHL-program to be translated. The IR is formed by different types of objects, organised into hierarchies. For information about the MEHL IR refer to chapter 14.5.

Below a segment of the MEHL grammar, supplemented with semantic actions are presented. In the main only the non-general parts related exception handling and in some cases to objects and are presented.

The TRY-EXCEPT production:

statement → TRY MidRuleAction1 { statement } EXCEPT MidRuleAction2 { exception_handler } END

MidRuleAction1: Let *prev_try* = *Try* dag node for containing TRY construction.
Let *try* = new *Try* dag node.
Append *try* to dag node list for *activeBasicBlock.*
try_nest_level:=*try_nest_level*+1.

MidRuleAction2: *except_nest_level*:=*except_nest_level*+1.
Create new *EndTry(try)* dag node and append it to active *BasicBlock*.
Let *except* = new object *Except*(*try*), with:
except_nest_level (attribute) =except_nest_level (counter).
current_procedure→append_exceptional(*except*).
Open new top symboltable for local exception variables in continuation (on top of) the current variable scope.
Let *bb_resume* = new *BasicBlock*, with *basic_block_access*=GLOBAL
current_procedure→switch_to_eh_basicblocks();
If *try_nest_level* >1: (* Are exception handler(s) guarded (nested) ? *)
  Let *start_except* = new dag node *StartExceptionalRegion()*
  Append *start_except* to dag node list for current *BasicBlock*.
  Let *end_except* = new dag node *EndExceptionalRegion*(start_except)
  prev_try→append_region(*start_except*→get_r*egion()*)
try_nest_level:=*try_nest_level*-1.

Action: If *end_except has been* defined append *end_except* to dag node list for the active *BasicBlock*..
current_procedure→switch_to_non_eh_basicblocks().
Close top symboltable for local exception variables.
Activate *bb_resume* basic block..
except_nest_level:=*except_nest_level*-1.

In the production *try_nest_level, except_nest_level* and *finally_nest_level,* are global counters used to keep track of the current nesting level of the guarded statements in a TRY construction, the nesting level of handler statements in a TRY-EXCEPT construction and the nesting level of the handler statements in a TRY_FINALLY construction. Also not defined in the production is c*urrent_procedure* which designates the *Procedure* object for the containing function.

In summary the production does the following:

- The dag nodes for the guarded statements are encompassed by a *Try* and a *EndTry* dag node respectively (Marking the code range of the guard).

- An *Except* object called *except* is created and appended to a list for the current procedure. The object is used by the productions for the exception handlers which insert handler specific information into it.

- A new topmost symbol table is created and activated while processing the exception handler(s) in order to keep track of the exception variables used for argumented exception handler(s).

- The *BasicBlock bb_resume,* representing the location where execution is normally resumed after the last guarded statement in the TRY-EXCEPT construction, is created. It is activated just before finishing the last action for the production. It is referred to by the exception handler productions.

- The active list of basic blocks is changed for the processing of the exception handler(s) so that dag nodes for handler(s) are entered into basic blocks outside the normal flow of control.

- If the TRY-EXCEPT construction is guarded by an outer TRY construction the dag nodes for the (nested) exception handler(s) are surrounded by a *StartExceptionalRegion* and a *EndExceptionRegion* dag node respectively (Marking an additional guarded code range for the containing TRY).

The TRY-FINALLY production:

statement → TRY MidRuleAction1 { statement } FINALLY
MidRuleAction2 { statement } END

MidRuleAction1: Let *prev_try* = *Try* dag node for containing TRY construction.
Let *try* = new *Try* dag node.
Append *try* to dag node list for active *BasicBlock*..
try_nest_level:=*try_nest_level*+1.

MidRuleAction2: *finally_nest_level*:=*finally_nest_level*+1.
Create new *EndTry(try)* dag node and append it to active *BasicBlock*.
Let *bb_finally* = new and activated *BasicBlock*, with :
basic_block_access = GLOBAL.
Let *finally* = new object *Finally*(*try*, bb_*finally*), with:
finally_nest_level (attribute) = *finally_nest_level (counter),*
except_nest_level (attribute) =except_nest_level (counter).
current_procedure→append_exceptional(*finally*).
try_nest_level:=*try_nest_level*-1.

Action: Let *after_finally* = new *Basicblock*, with: b*asic_block_access*=LOCAL.
If active basic block = *bb_finally* (Only one basic block for finally statements) and use of the copy approach is enabled (which is the default compiler option):
(* Use copy approach for finally statements - Code for the finally statements are copied to basic blocks outside the current basis blocks *)
current_procedure→switch_to_eh_basicblocks()
Let *bb_copy* = new *Basicblock*, with *basic_block_access*=LOCAL.
(* nested try in try for copy ? If yes add range in ranges guarded by previous try *)
If *try_nest_level* >= 1:
Let *start_finally* = new dag node *StartExceptionalRegion()* (* Start of range *)
Append *start_finally* to dag node list for *bb_copy*
prev_try→append_region(*start_finally→*get_r*egion()*) (* Append to previous try *)
Let end_finally = new dag node *EndFinally*(*start_finally*,*after_finally*),
with: *finally_nest_level* (attribute) = 0 (* Disable test *)
Else:
Let end_finally = new dag node *EndFinally*(*after_finally*) (* No range spcified *)
with: *finally_nest_level* (attribute) = 0 (* Disable test *).
Append *end_finally* to dag node list for *bb_copy.*
finally→set_finally_location(*bb_copy*) (* Change location of finally handler to copy *)
current_procedure→switch_to_non_eh_basic_blocks()
Else:
(* Use test approach for finally statements *)
Let *end_finally* = new dag node *EndFinally*(*start_finally*) (* No range specified*)
with: *finally_nest_level (attribute) = finally_nest_level (counter)*
Append *end_finally* to dag node list for active *BasicBlock.*
Activate the *after_finally* basic block*.*
finally_nest_level:=*finally_nest_level*-1

In this production, as for the TRY-EXCEPT production, *try_nest_level, except_nest_level* and *finally_nest_level,* are global counters used to keep track of the current nesting levels. Also, as for the TRY-EXCEPT production, *current_procedure* denotes the containing Procedure.

In summary the production does the following:

- The dag nodes for the guarded statements are encompassed by a *Try* (called *try*) and a *EndTry* dag node respectively (Marking the code range of the guard).

- The *BasicBlock bb_finally,* representing the entry point of the finally statements for an exception, is created. It is activated before processing the finally statements.

- A *Finally* object called *finally* is created and appended to a list for the current procedure. From the start it is associated to *try* and *bb_finally*. Later if the finally statements are copied the *bb_finally* association is changed to an association to the copied basic block.

- The *BasicBlock* after_finally, representing the location where execution is normally resumed after the last guarded statement in the TRY-FINALLY construction, is created. It is activated just before finishing the last action for the production.

- If the finally statements could fit into a single basic block and if the copy approach is enabled the then a copy approach is used otherwise a test approach is used.

  If a copy approach is used the basic block with the finally statements are copied and placed in a list of basic blocks outside the normal flow of control. If the TRY-FINALLY construction is guarded by an outer TRY construction the dag nodes in the copy is surrounded by a *StartExceptionalRegion* and a *EndException-Region* dag node respectively (Marking an additional guarded code range for the containing TRY). Otherwise a single *EndFinallyRegion* dag node is placed at the end of the list of dag nodes for in the copy.

  If a test approach is used a *EndFinallyRegion* dag node (with its finally_nest_level attribute set) is placed at the end of the list of dag nodes for in the copy.

The RAISE productions:

statement → RAISE

Action: Create new dag node *ReRaise* and append it to the dag node list
for the active *BasicBlock*.

statement → RAISE exception_type_id
['.' constructor_id '(' actual_parameters ')']

Check: Exception_type_id refers to an existing *ExceptionType*.
If constructor_id is specified:
Constructor_id refers to an constructor for the *ExceptionType*.
Actual_parameters match the formal parameters for the constructor.

Action: Let *E* = The *ExceptionType* referred to by exception_type_id.
Let *ctr* = The *MemberFunction* referred to by constructor_id, if specified, or
the default constructor for *E*.
Create new dag node *Raise*(*E*,*ctr*) and append it to the dag node list for the active
BasicBlock.

The raise productions are responsible for creating the RAISE and RERAISE dag nodes used to represent the raising and re-raising of exceptions. For the RAISE dag node eventual actual parameters, represented as individual *Arg* dag nodes, have been appended to the dag node list prior the RAISE action.

The exception_handler productions

exception_handler → '|' exception_type_id '(' var_id ')'
"=>" MidRuleAction statements

MidRuleAction: Let *E* = The *ExceptionType* referred to by exception_type_id.
Let *V* = new *ExceptionVariable(E)*, with
name = the string referred to by var_id
nest_level = *except_nest_level*
Enter *V* into current symboltable for variables.
If an *ExceptionVariable* with equal *nest_level* is not found in the sorted list of exception variables for the function retrived by *current_procedure→get_defined_-exception_variables*() then insert *V* into the list by executing *current_procedure→-exception_variable_defined*(*V*).
Create and activate new *BasicBlock B*, with
basic_block_access = GLOBAL.

Action: Create new *ExceptionHandler EH*(*V*,*B*).
except→append_handler(*EH*).
Create new dag node *EndExceptHandler(bb_resume)* and append it to the dag node list for the active *BasicBlock*.

Check: Success for *except->check*(*E*) (* Validate correct order of handlers *)

exception_handler → '|' exception_type_id { ',' exception_type_id } "=>" MidRuleAction statements

MidRuleAction: Create and activate new *BasicBlock B*, with
basic_block_access = GLOBAL

Action: Let *E* = The list of *ExceptionType's* referred to by exception_type_id's.
Create new *ExceptionHandler EH*(*E*,*B*).
except→append_handler(*EH*).
Create new dag node *EndExceptHandler(bb_resume)* and append it to the dag node list for the active *BasicBlock*.

Check: Sucess for *check_for_exception_types_overspecification*(*E*).
Succes for *except→check*(*E*). (* Validate correct order of handlers *)

The exception_handler productions do not define *except* and *bb_resume*. They are defined by the containing TRY-EXCEPT construction to be equal to the current *Except* object and the basic block where execution will continue after the TRY-EXCEPT. Also not defined is *current_procedure* which denotes the containing *Procedure*.

In summary, the production with the argumented exception handler creates an exception variable, inserts it in the sorted list of exception variables for the current procedure if no other exception variable with the same nest level exist, creates and activates a new basic block *B* which initiates the handler and then creates an *ExceptionHandler* object and appends it to the handlers for the current *Except* object.

In summary, the production with the unargumented exception handler creates and activates a new basic block *B* which initiates the handler and then creates an *ExceptionHandler* object and appends it to the handlers for the current *Except* object.

The RETURN production:

statement → RETURN [simple_expr]

Action: Let *expr* = The dag node referred to by simple_expr, if specified, or zero.
Let *return_type* = The return type of the containing function, if a function
Let *return* = created dag node *Return*(*expr*) with :
raise_return_flag = TRUE if *finally_nest_level*>0 OR *try_nest_level*>0 OR *except_nest_level*>0, otherwise FALSE
Append *return* to the dag node list for the active *BasicBlock*.

Check: Simple_expr is specified for a function and not specified for a procedure.
Succes for *is_compatible*(*return_type*, *expr→get_type*()), if a containing function.

The return production produces the *Return* dag node from a return statement. Importantly, the production decides whether the return statement should raise a local return exception. This is only the case if a return statement is contained in a TRY construction.

The object-type production:

type → OBJECT ['(' object_type_id ')']
MidRuleAction { var_decl | method }
END

Check: Object_type_id, if stated, is a reference to a fully defined *ObjectType*.

MidRuleAction: Open new topmost symbol table *V* for variables and new topmost symboltable *F* for functions/procedures. If an object_type_id is stated, the new symbol tables will have scopes which are in continuation of the scopes associated with its ancestor, otherwise the scopes will be in continuation of the global scopes.

Action: Let A = The *ObjectType* associated with object_type_id, if specified, or zero.
If object_type_id is specified and object_type_id refers to an *ExceptionType* then create new *ExceptionType*(A, *V*, *F*) otherwise create new *ObjectType*(*A*, *V*, *F*).

Return: The newly created type.

The object-type production is used to construct object (and exception) types. The returned value is used when inserting the type in the global symbol table for types (not shown).

The procedure-method productions:

method → (PROCEDURE | CONSTRUCTOR | DESTRUCTOR)
[object_type_id '.'] procedure_id fparams ';'
[VIRTUAL ';'] [raises] body

Check: Object_type_id, if specified, represents a known object type. If object_type_id is specified, the current scope must be the global scope, and procedure_id must be declared (but not defined) as a method with a equal header. If object_type_id isn't specified, function_id must not previously be defined in the current scope (for the object-type).

MidRuleAction: Open new topmost symbol table scope *V* for formal parameters and local variables. If a object-type is stated, the new scopes will be in continuation of the scopes associated with that object-type, otherwise the new scopes will be in continuation of the global scopes.

Action: Let *O* = The object_type_id if specified, otherwise the type of the containing object decl.
Let *B* = Any list of basic blocks in body.
Create new *MethodProcedure* (*V*,*O*,*B*) with *name* = procedure_id &
method_descriptor = CONSTRUCTOR_METHOD,
DESTRUCTOR_METHOD, METHOD,
VIRTUAL_METHOD or VIRTUAL_DESTRUCTOR_METHOD.
Append the newly created *MethodProcedure* to the symbol table in *O*'s scope.

The procedure-method production is used for constructing methods which are not functions. The function-method production is much equal.

The raises production:

raises → RAISES exception_type_id { ',' exception_type_id }

Action: Let *E* = List of exception-types in raises-set.

Check: Success for *check_for_exception_types_overspecification*(*E*).

Return: *E*, except for the special non-restrictive case of a raise-set equal to {Exception} in which case "nothing" is returned.

The raises-production processes the list of exception types in an optional raises-set. The operation *check_for-_exception_types_overspecification* is used to check that the raises-set does not contain unnecessary exceptions (an exception and the ancestor for the exception specified in the same raises-set). Finally, as a raises-set consisting of the standard superclass *Exception* isn't restrictive, any such raises-set is ignored.

14.5 The MEHL compilers intermediate representation

In this section an overview of the MEHL compiler's intermediate representation (IR) is presented with the focus on those parts in the front end that concern or have relevance for EH.

14.5.1 The object hierarchies

For a MEHL compiler module the intermediate representation generated by the front end, to be used by the code generator, is formed by five different types of objects, organized into hierarchies rooted by the superclasses *Node*, *Procedure*, *Variable*, *Type* and *Module*. The size of the object hierarchies ranges from 2, in the *Module* "hierarchy", to 25 objects in the *Node* hierarchy. An overview of the object hierarchies is presented in Appendix A. In Appendix B object models are specified in the [Rumbaugh91] OMT model for the EH related classes.

14.5.2 The Program and Module classes

The *Program* and *Module* classes are the chief holders of IR for each MEHL module. They contain a module name, a symbol table for functions, a symbol table for variables, a symbol table for types, a list of defined text strings and a list of defined objects.

The *Program* class is used for the main MEHL program module and a *Module* class is used for each MEHL unit. The difference between a *Module* and a *Program* is that a *Program* generates code for initialisation, specifies a program entry point and allocates stack, all of which a *Module* does not do.

14.5.3 The Variable hierarchy

The variable hierarchy contains five different kinds of variables: *GlobalVariable, LocalVariable, MemberVariable and ExceptionVariable*. The modeling of the classes is presented in object model 4 in Appendix B.

All variables have a name and an associated type. In addition an *ExceptionVariable* contains the nest level for the exception handler it belongs to. The nest level is used to sort all exception variables so that the code generator can place the exception variables in order of decreasing nest level at the start of the local data for a function.

14.5.4 The Procedure hierarchy

While a function conceptually is the most general, a procedure is with respect to the object model the most general component. The *Procedure* hierarchy contains 7 different kinds of procedures: *Procedure*, *Function*, *MethodProcedure*, *MethodFunction*, *MainProcedure*, *StandardProcedure* and *StandardFunction*.

Procedures of type *MainProcedure* are used for the initialisation and deinitialisation of modules. Procedures of type *StandardProcedure* and *StandardFunction* are used for the internal MEHL functions and procedures such as NEW and DELETE etc.

The modeling of the key classes *Procedure* and *MethodProcedure* are presented in object model 5 in Appendix B. As shown each procedure contains a name, an imported or exported specification, an external specification and the depth (max. level) of contained exception handlers.

Each procedure has a symbol table of local variables and formal parameters. The entries in the symbol table are ordered which means that variables can be iterated upon in the order they where inserted. Each procedure also has a main list of basic blocks containing the dag nodes for the ordinary statements in a procedure.

To support that code for exception handlers are generated outside the normal path of execution, the main list of basic blocks is augmented with an optional list of lists of basic blocks to be used for exception handlers (one list used for each level of exception handler(s) used). In addition, each procedure has an optional list of exception types for a specified raises-set, a sorted list of exception variables (sorted in order of nest level) and a list of *Except* objects containing data for the EH constructions used in the procedure.

A *MethodProcedure* inherits from *Procedure* and from the abstract class *Method,* which supplies the common data and operations for all object-methods (procedures or functions). A *Method* contains a description of the type of method (ordinary method, constructor, destructor and/or virtual) and have associated the type of the object of which it belongs to.

For the front end the key methods related to EH for *Procedure* are: *switch_to_eh_basic_blocks / - switch_to_non_eh_basic_blocks, append_exceptional* and *exceptional_variable_defined*.

- The methods *switch_to_eh_basic_blocks* and *switch_to_non_eh_basic_blocks* are provided for facilitate the switching of the active list of basic blocks for which the IR for statements is generated. The methods handle the main list of basic blocks and a list of lists of basic blocks (for exception handlers) using a stack discipline.
- The method *append_exceptional* appends an *Exceptional* object to a list of such objects for the procedure. *Exceptional* objects are used to represent the data for exception handlers. They are discussed in section 14.5.7.
- The method *exceptional_variable_defined* appends an *ExceptionVariable* to the sorted list of exception variables for the procedure.

14.5.5 The Type hierarchy

The type hierarchy contains seven kinds of types: *IntegerType, CStringType, ArrayType, PointerType, ObjectType, ExceptionType* and *StandardExceptionType*. The modeling of the classes is presented in object model 2-3 in Appendix B.

Each type contains an import/export designator. However the standard types *IntegerType*, *CStringType* and S*tandardExceptionType* are defined internally and can not be imported or exported as other types can.

For each type the method *is_compatible* is used by the front end to decide whether two types are compatible (equal for simple types, compatible according to normal object-oriented rules for object types). The methods *is_assignable*, *is_passable* and is*_instanceable* are used to decide whether a type can be assigned to, passed as a parameter or created as a variable.

Objects are represented by the *ObjectType* class (including exceptions which has *ObjectType* as a superclass). Each object has a symbol table of methods and a symbol table of member variables. The entries in the symbol tables are ordered so that the member variables can be iterated upon in the order they where inserted. Optionally an *ObjectType* may also have an association to an ancestor. An *ObjectType* also has a mangled name since, contrary to other types, individual code must be generated for each object for which a unique (describing) identifier is needed. Finally, it is specified whether an object is explicitly declared as a distinct type.

User defined exception types are represented by the *ExceptionType* class and the standard *Exception* type is represented by the *StandardExceptionType* class. For the front end the main difference between an object type and an exception type is that an exception type unlike object types can't be created as variables by the user, specified by letting the method *is_instancable* return false for *ExceptionType*. Each exception type has a *user_name* that is equal to the user specified type name for the type. *User_name* is used when generating runtime

type information for the exception (The *name* defined in *ObjectType* can't be used as it is really a mangled identifier targeted for the assembler code).

14.5.6 The Node hierarchy

The IR of MEHL statements is in essence based on the concept of directed acyclic graphs, Dags. More to the point, the hierarchy rooted by the abstract superclass *Node* constitutes a dag language for the statements in the MEHL intermediate representation[21]. An overview of this large hierarchy with a total of 26 non-abstract classes is shown in Appendix A.

The nodes in the *Node* hierarchy can be divided into the abstract superclasses *RootNod*e and *CommonNode*. The dag nodes which originate from the superclass *CommonNode* are those nodes that correspond to expressions in a MEHL program. They exist as children (direct or indirect) of those nodes that represent the actions, the nodes which originate from the abstract superclass *RootNode*. The modeling of the three topmost superclasses *Node*, *RootNode* and *CommonNode* (summarized) are presented in object model 5 in Appendix B. For their non-abstract subclasses the three superclasses provide the basic support for generating code from dags. Dags which are a bit extraordinary since an *Assignment* node rather than the usual labeling of a node is used to represent assignments. This and other aspects of the generation of code from dags is discussed in detail in Appendix E.

The dag nodes *Try, EndTry, StartExceptionalRegion, EndExceptionRegion, EndFinally, EndExceptHandler, Raise, ReRaise* and *Return* are directly connected with EH.

The *Raise* and *ReRaise* node are used when raising and re-raising exceptions. *Raise* has an exception type and an optional constructor for the exception type associated.

The *Return* node may or may not involve exceptions depending of the context of its corresponding return statement. The context is specified by the *raise_return_exception* flag. In any case it may have an associated return value (if contained in a function).

The objective for the *Try*, *EndTry*, *StartExceptionalRegion, EndExceptionRegion* and *EndFinally* classes is to facilitate that the code generator can insert labels in the target assembler code marking the start and end of guarded regions. These nodes encompass the dag nodes produced for the guarded statements.

The objective for the *EndExceptHandler* dag node and a secondary objective for the *EndFinally* dag node is to facilitate that the code generator can insert the assembler code to terminate or re-raise an exception at the end of a handler. Both classes hold information about the basic block which is the entry point for resumption if execution continues after the handler.

Note that for exception handlers (including copied handlers for a finally construction) code is generated outside the normal flow of control using basic blocks specially allocated for the handlers. Consequently for nested exception handlers the code for the guarded statements may be divided into several disjoining regions and as such any TRY construction can have not just one but any (non-zero) number of associated guarded regions of code.

In the IR the start and end of each region of guarded code are represented by the *Region* class which contains the start and end label to be used for the region's code range. A *Try* node has associated any number of *Region* classes (1+ the number of nested EH constructions in total). A *StartExceptionalRegion* node has associated one *Region* class (for the handlers in the construction). The classes *EndTry* and *EndExceptionRegion* has indirect access to its *Region* class through the association to their corresponding *Try* or *StartExceptionalRegion* classes.

The *EndTry* and *EndExceptionRegion* has associated their corresponding *Try* or *StartExceptionalRegion* classes. But for the *EndFinally* class there is not always an association to a corresponding *StartExceptionalRegion.* This is because *EndFinally* is always used to produce termination code for finally statements but only optionally to

[21] In accordance to [Fraser91] the term dag language refers to the composition of the intermediate representation into dag nodes..

mark the end of a region (only when a copy scheme is used). The termination code differs accordingly to the scheme used for translating the specific finally handler (copy or test). When a test scheme is used *EndFinally's* attribute *finally_nest_level* is set to the nest level of the finally handler, when a copy scheme is used *finally_nest_level* is zero (indicating for the code generator that a copy scheme is used).

14.5.7 The Exceptional hierarchy

The *Exceptional* hierarchy consists of the two non-abstract subclasses *Except* and *Finally* which are used to represent TRY-EXCEPT and TRY-FINALLY constructions as a whole. Each yields the information used in the static EH tables about the handler(s) for an EH construction. Information which does not depend on code positions and as such is not represented as dag nodes. When a function includes EH constructions the individual *Exceptional* objects, one for each construction, are appended to a list of EH constructions for the function.

The abstract superclass *Exceptional* provides the information common to all EH constructions being the except nest level, called *except_nest_level*, and an indirect association to the region(s) guarded by the EH construction through the corresponding *Try* dag node.

The simplest class is the *Finally* class which holds the finally nest level, called finally_nest_level, and the basic block which constitutes the entry point for the finally exception handler.

The *Except* class associates with individual information for each of the handlers contained in the corresponding TRY-EXCEPT construction. Each handler is represented by the *ExceptHandler* class. A *ExceptHandler* class consists of an association to the basic block which is the entry point for the handler and an association to either an *ExceptionVariable*, when a single argumented exception is handled, or a list of *ExceptionType's*, when one or more (unargumented) exceptions are handled. The *handler_kind* attribute specifies which type of exception handler it is.

The methods *append_handler* and *check* in *Except* are used by the front end. The method *append_handler* is used to append an *ExceptHandler* object to the exception handlers associated with *Except*. The method *check* is used for validating that the exception handlers are specified in the correct order. It checks all handlers associated to an *Except* object and returns false if a handler for more general exception(s) was specified before a handler for less general (derived) exceptions!

15. The code generator for the MEHL compiler.

In this section the design of the MEHL compiler's code generator is presented with the focus on those parts that concern or are relevant for EH.

15.1 Overview of the code generator

The task of the code generator is to transform the intermediate representation, presented in chapter 14.5, into INTEL 80286 code in Borland Turbo Assembler format.

The considerable work of translating the bodies of procedures and functions is done, using this dag language rooted by *Node* constitutes. In comparison, the hierarchies rooted by *Procedure*, *Module, Variable*, *Type* and *Exceptional* can primarily be considered as containers of relevant information. For those the translation to target code are simple.

The translation done from the IR to symbolic assembler by the code generator, contains roughly of the following steps:

1. Generate comments to describe file, title of module etc.

2. Generate necessary module independent assembler header.

3. Allocate stack, if it is a *Program*-module.

4. Define assembler objects (including exception objects) and allocate room for their virtual method tables (VMT's). For exception objects place runtime type information (RTTI) before the VMT.

5. Allocate space for strings and global variables.

6. If it is a *Program* module, generate start-up code for program.

7. Generate code for global procedures and functions defined in the module.

8. For objects defined in the module generate code for their methods.

9. Output code for EH's function table in segment for static table 1.

10. Output code for EH's main program table in segment for static table 2.

11. If it is a *Program* module, define program entry point.

In the above to-do-list for the code generator the order of the individual items corresponds to the output of code which is partitioned into the five segments: STACK, DATA, CODE, EXCEPT@1 & EXCEPT@2.

The source module independent assembler header mentioned in point 2, constitutes the following list of assembler directives:

- IDEAL
- P286
- MODEL SMALL, PASCAL
- JUMPS

These directives specify the assembler notation, the target processor, the memory model and default calling convention, as well as request automatic generation of non-local conditional jump equivalents.

The action specified in point 3 is equal to the action performed by a method called *gen_head_directives* in *Module* and *Program*. For a *Module* this method will simply output an empty stack segment declaration, for a *Program* this method will output a stack segment declaration with a stack definition..

The code elements mentioned in point 4-5 must be declared within a data segment definition. Point 4, the declaration of objects, allocation of VMT's and generation of RTTI for exception objects, is done by a static function called *gen_object_decls.* Point 5, the declaration of strings and global variables are done by the static functions *gen_strings* and *gen_non_local_data* respectively.

The action specified by point 6 are performed by the method *gen_startup_code* in *Module* and *Program*. For a *Module*, this method does nothing. For a *Program* this method generates a start-up code sequence to the code segment, which do the following things (in the symmetric order specified below):

A. Initialises the data segment register.

B. Calls the main initialisation procedure for each unit (initialisation + user code).

C. Calls the program module main procedure (initialisation + user code + deinitialisation).

D. Calls the main deinitialisation procedure for each unit (deinitialisation).

E. Terminates program.

Note that construction and destruction of global objects are performed by the initialisation and deinitialisation code and note that initialisation and deinitialisation, similar to construction and destruction of objects, are symmetric. The labels INIT and EXIT are inserted in the code at the start and end respectively of the calls for initialisation and deinitialisation procedure calls. The EH mechanism uses the labels and the sequence of calls to deduce what deinitialisations to perform in the event of an unhandled exception!

The code elements mentioned in point 7-8 must be declared within a code segment definition. The generation of code for procedures and functions defined in a module, as specified by point 7 corresponds to the action done by the static function *gen_procedures*. Point 8, the generation of code for implementations of the methods for objects defined in the module, corresponds to the action done by the static function *gen_object_methods*.

The static function and program exception tables are output in point 9-10 into the logical segments EXCEPT@1 and EXCEPT@2 respectively. Using two temporary buffers called *exception_table* and *exception_table2* these tables have been generated on the fly while generating code for the program.

The action specified in point 11 is equal to the action performed by the method *gen_end_directive* in *Module* and *Program*. For a *Module* this method will simply output an END directive, for a *Program* this method will output an END directive followed by a label indicating the program entry point (the startup code).

15.2 Use of registers

All accesses to machine & temporary registers by the code generator are handled by a set of register management functions. The general machine registers ax,cx,dx,si,di can be allocated, the rest of the general registers are reserved. Among the reserved registers is register bp which, as the 80x86 architecture promotes, is reserved as a stack frame register and register bx which is reserved as a general accessible volatile register!

Much thought has gone into the subject of assigning registers, how to pass return values and how to refer to members in objects. Primarily, due to a high performance cost for all memory accesses in the INTEL 80x86 architecture, it has been decided to optimise the use of registers in these areas. This implies the use of a register to pass return values from functions and the use of an object frame register when accessing members in an object.

Register ax is used to pass return values from functions! Although register ax is allocable, it is always used to yield the return-value of functions. This use of register ax agrees with most 80x86 compilers. The overlap

represents no problem as the code generator does not do any global optimisation, which cause all registers to be dead after a function call.

Note that use of register ax to pass return values from functions imply that the initial value of register ax must be preserved while destroying local objects and while raising a local return exception for the RETURN statement.

Register di will be used as an object frame pointer (*THIS* register) for objects. However since not all parts of a MEHL program involve objects it makes little sense to have a machine register constantly reserved. Instead it will be allocated/deallocated on entry/exit of each method.

Note that the *THIS* register is in effect an internal register variable used for methods. Its value must be reloaded whenever it might be destroyed or not defined correctly including when entering an exception handler located in a method. To avoid forcing the compiler to generate code for a reload at the start of each handler the EH mechanism does this automatically before transferring control to the handler!

15.3 translation of a function / procedure

The generation of code for each function and procedure is handled by the operation *gen_code* for *Procedure*. Different code are generated for a procedure, a function or a method. The difference is primary located in the prologue code and epilogue code generated for each procedure, function and method for which the operations *gen_prolog_code* and *gen_epilog_code* is responsible. For more information including pseudo-code for the prologue and epilogue code refer to section 15.3.1. The pseudo-code for the gen_code operation is shown below:

```
gen_code
BEGIN
   IF NOT (is_prototype() OR is_external() OR is_imported()) THEN
      IF is_exported() THEN
         output(“PUBLIC ” + get_full_name());
      output(“PROC ” + get_full_name());

      LET parms = string of declarations of explicit formal parameters;

      IF (procedure IS a Method) THEN
         output (“ARG ” + params + “@@STATUS:WORD, @@THIS:WORD”);
      ELSE
         output (“ARG ” + params + “@@THIS:WORD”);
      ELSE IF params not empty THEN
         output( “ARG ” + formal parameters);
      ENDIF;

      try_count:=0; (* Init global counter for TRY constructions *)
      LET exception_ctrs_dtrs_table = empty code buffer;
      LET exception_regions_table = empty code buffer;

      (* Generate code for procedure *)
      WITH standard output send to tempoary buffer tmp_buf DO
         antal_ctrs=gen_prolog_code(); (* Generate prolog code for procedure *)
         gen_basic_blocks(get_basic_blocks()); (* Generate ordinary basic blocks *)
         antal_dtrs=gen_epilog_code(); (* Generate epilog code for procedure *)
         FOR EACH list of basic blocks i in get_eh_basic_blocks_list() DO
            gen_basic_blocks(i); (* Generate basic blocks for handler(s) *)
         END;
         FOR EACH Exceptional object in get_exceptionals() DO
            gen_code(); (* Generate static table information for EH constructions *)
         END;
      END;

      LET vars = string of declarations of explicit local variables
```

```
LET tmp_regs = string of declarations of register tempoaries used while generating
                    code for basic blocks.
LET evars = string of declarations of exception variables with one word for
                each level of TRY-EXCEPT constructions.

IF  (vars is empty AND temp_regs is emptry AND evars is empty) THEN
"LOCAL " + tmp_regs +vars + evars (* Declare local data *)

(* Now that local data has been declared,code for the procedure can be output'd:*)
output(tmp_buf); (* Copy tmp_buf to output *)
"ENDP " get_full_name

(* Specify start of function in prg table entry *)
To exception_table2 output("DW " + get_full_name());

LET ctr_dtr_count = MAX(ctrs_count, dtrs_count);
LET raises_set_size = get_raises_set()→get_size() if raises-set specified or -1.

IF procedure IS a Method m THEN
    IF m→is_constructor() THEN LET pt = CONSTRUCTOR;
    ELSE IF m→is_destructor() THEN LET pt = DESTRUCTOR;
    ELSE LET pt = METHOD;
ELSE LET pt = PROCEDURE;

IF ctr_dtr_count>0 OR pt=CONSTRUCTOR OR try_count>0 OR raises_set_size<>-1
THEN
    LET ename = get_unique_exception_table_entry_name()
    (* Specify function table addr in prg table entry *)
    To exception_table2 output("DW  " + ename);

    (* Begin outputting function table *)
    To exception_table output("LABEL " + ename + " DATAPTR");

    To exception_table output(      "DB " + (* Specify general information byte *)
                            "00" if pt = PROCEDURE OR "01" if pt=METHOD OR
                            "10" if pt = CONSTRUCTOR OR "11" if pt = DESTRUCTOR +
                            bit-string for ctr_dtr_count if ctr_dtr_count<=6 OR "111" +
                            "1" if raises_set_size>0 OR "0" +
                            "1" if raises_set_size=0 OR "0");

    (* Specify construction/destruction adresses for objects *)
    IF ctr_dtr_count>0 THEN
        (* Null-entries must be inserted for global and member variables since
        construction  and destruction is not done in the same procedure : *)
        IF pt = CONSTRUCTOR OR procedure IS of type MainProcedure THEN
            REPEAT ctrs_count-dtrs_count TIMES
                To exception_table output(("DW 0");
            END;
        ENDIF;
        IF ctr_dtr_count>=7 THEN (* # specified inline in general information byte ? *)
            To exception_table output("DW " + ctr_dtr_count);
        ENDIF;
        (* Copy code in local ctr/dtr buffer to main function table buffer *)
        (* Copy ctr/dtr buffer to main function *)
        To exception_table_output(exception_ctrs_dtrs_table);
    END;

    IF try_count>0 THEN (* Specify info about exception handlers *)
        To exception_table output("DB " + try_count);
        To exception_table output(exception_regions_table);
```

```
            END;

            IF raises_set_size>0 THEN (* Specify raises-set *)
                To exception_table output("DB " + raises_set_size);
                FOR EACH ExceptionType e IN get_raises_set() DO
                    To exception_table output("DW @TableAddr_" + e→get_name());
                END;
            END;
        ELSE (* Specify no function table in program table entry *)
            To exception_table2 output("DW -1");
        END;
    END ELSE output("EXTRN " + get_full_name() + ":PROC"); ENDIF;
END;
```

15.3.1 Prolog and epilog code

The *gen_prolog_code* and *gen_epilog_code* methods exist in different variations. In procedures/functions the work to be done is basically limited to implicit calling of the default constructors/destructors of any locally scoped objects.

For procedures of type *MainProcedure* the locally scoped variables are equal to the global variables in the containing module. However the calling of constructors or destructors is limited to the *MainProcedure* in charge of initialisation or deinitialisation respectively (Recall that unit-modules have one *MainProcedure* for initialisation and one *MainProcedure* for a deinitialisation).

For methods for objects code are generated to allocate or deallocate memory from heap if requested, to construct or destroy any ancestor, to construct or destroy any members of type object and to initialise or reset a VMT pointer. Finally, as for non-methods, any default constructors or destructor of locally scoped objects are called. The prologue and epilogue code for exceptions is equal to the code for ordinary object, except for the calls to the heap-management functions which is substituted with calls to the exception memory management functions.

Since the prologue and epilogue code for methods is of primary interest with respect to EH and since the generation of prologue code and epilogue is similar only the pseudo code for the *gen_prolog_code* operation for methods is shown below:

```
gen_prolog_code()  (* for MethodProcedure/MethodFunction *)
BEGIN
    allocate_reg(si); (* Reserve THIS register *)
    IF is_constructor() THEN
        output(    "cmp [@@this],0" +
                "jne @@ovr" +
                "push SIZE " get_object_type()->get_name());
        IF get_object_type() IS ExceptionType()
            output("call EH_ALLOCATE"); (* Allocate memory for exception *)
        ELSE
            output("call NEW"); (* Allocate heap-memory for object *)
        ENDIF;
        output("or ax,ax" +
                "jz @@fin" +
                "mov [@@this],ax" +
                "@@ovr:");
    ENDIF;
    LET ctr_count = 0;
    LET ancestor = get_object_type→get_first_constructable_ancestor();
    IF ancestor set THEN
        LET name = get_unique_exception_table_ctr_name()
        output("LABEL " + name + " CODEPTR"); (* Label position in code *)
```

```
        To exception_ctrs_dtrs_table output(“DW “ + name); (* Specify position in table *)
        output(“push [@@this]”)
        output(“call +@Table_” + ancestor→get_name() “| ” +
                ancestor→get_default_constructor()→get_name());
        INC(ctr_count);
    ENDIF;
    load_this_reg(); (* si ← [@@this] *)
    IF is_constructor THEN
        ctr_count:=ctr_count+gen_variable_init_or_done(
                    get_object_type()→get_variables(),CONSTRUCTOR);
        IF get_object_type()→has_virtual_methods() THEN
            output(“mov [(” + get_object_type()→get_name() + “ PTR si “).@Mptr_” +
                    get_object_type()→get_name() + “],OFFSET @TableAddr_” +
                    get_object_type()→get_name());
        ENDIF;
    ENDIF
    RETURN ctr_count;
END;
```

15.4 gen_variable_init_or_done

All objects must be properly constructed when their scope is entered and destroyed when their scope is left in symmetrical order. Also, when constructing or destroying an object all contained object members must be constructed or destroyed respectively. Besides constructing and destroying simple objects, individual objects contained in arrays must likewise be constructed or destroyed.

The operation *gen_variable_init_or_done* does the implicit work of calling constructors or destructors in all the cases mentioned above. The operation has two arguments, one argument specifying a symbol table containing the variables that must be constructed or destroyed, and one argument specifying whether the objects should be constructed or destroyed.

The addresses of the constructor and destructor calls must be marked and placed in the static function tables for EH. Below the parts of the *gen_variable_init_or_done operation* that are relevant to EH can be observed:

```
gen_variable_init_or_done(variables, cd_spec) : NUMBER
BEGIN
    LET count = 0;
    FOR EACH Variable v IN variables DO
        IF (NOT v→is_imported()) AND (NOT v IS FormalParameter)
            AND (v→get_type()→get_first_non_array_sub_type() IS ObjectType) THEN
            (* object_type = type or element_type for arrays: *)
            LET object_type = v→get_type()→get_first_non_array_sub_type();
            IF (cd_spec=CONSTRUCTOR) THEN
                LET init_or_done_proc = object_type→get_default_constructor()
            ELSE (* cd_spec=DESTRUCTOR *)
                LET init_or_done_proc = object_type→get_default_destructor();
            ENDIF;
                IF a default constructor/destructor exist (init_or_done_proc set) THEN
                (* For destructors the code positions are marked before the call *)
                IF cd_spec=DESTRUCTOR THEN
                    IF (NOT get_type() IS ArrayType) THEN output(“push 1”);
                    LET name = get_unique_exception_table_ctr_name()
                    output(“LABEL ” + name + “ CODEPTR”); (* Label position in code *)
                    (* Specify position in table *)
                    To exception_ctrs_dtrs_table output(“DW “ + name);
                END;
                v→gen_push_addr();
```

```
                IF (get_type() IS ArrayType) THEN (* Array of objects *)
                    output("push offset +@Table_" + object_type→get_name() + "|" +
                            init_or_done_proc→get_name()); (* Push addr of element *)
                    output("push size " + object_type→get_name()); (*Push size of element*)
                    output("push type " + type→get_size()/object_type→get_size() (*push #*)
                    IF cd_spec=CONSTRUCTOR THEN (* Call array apply for ctr/dtr *)
                        output("call ARRAY_CTR_APPLY");
                    ELSE
                        output("call ARRAY_:DTR_APPLY");
                    ENDIF;
                ELSE (* ordinary (single) object *)
                    output("call +@Table_" + object_type→get_name() + "|" +
                            init_or_done_proc→get_name()); (* Call ctr/dtr *)
                END;
                (* For constructors the code positions are marked after the call *)
                IF cd_spec=CONSTRUCTOR THEN
                    LET name = get_unique_exception_table_ctr_name()
                    output("LABEL " + name + " CODEPTR"); (* Label position in code *)
                    (* Specify position in table *)
                    To exception_ctrs_dtrs_table output("DW " + name);
                END;
                load_this_reg();
            ELSE (* Non default constructor/destructor exist *)
                (* Place empty word for missing ctr/dtr *)
                To exception_ctrs_dtrs_table output("DW 0");
            END;
            INC(count); (* Count number objects/ array of objects *)
        END;
    END;
    RETURN count; (* Return number of objects/array of objects *)
END;
```

15.5 The function gen_object_decls

In the object-oriented assembler language used, all objects must be declared before they are used. The method *get_defined_object_types* in *Module* returns a list of each object type visible in a MEHL module (defined objects + imported objects). The *gen_object_decl* function iterates over the list of object types, outputting an assembler declaration of each object in turn. If the object type is not imported and the object has a VMT, room for the VMT is allocated for each object type, right the declaration and if the object is an exception object, RTTI is inserted using the operation *gen_rtti_code* for *Exception*. The pseudo code for the code generated for each object *O* in the list is as follows:

```
gen_object_decl(O: List of type ObjectType)
FOR EACH object o IN O DO
    output("STRUC " + o→get_name() +);
    (* Specify ancestor *)
    IF o→has_ancestor() THEN output(o→get_ancestor()→get_name() +);
    output(" METHOD {");
    FOR EACH method m IN o→get_methods() DO (* Declare methods *)
            IF m→is_virtual() THEN output("VIRTUAL" +);
            output(m→get_name() + ": WORD =" + m→get_full_name());
    END;
    "}"
    LET has_local_data = gen_non_local_data(o→get_variables());  (* Define members *)
    IF (NOT has_local_data) AND (NOT o→has_virtual_methods()) THEN
            output("DB 0");    (* Ensure that SIZEOF(o)>=1 byte *)
    ENDIF;
    output("ENDS " + o→get_name())
```

```
    IF (NOT o→is_imported()) AND (NOT o→has_virtual_methods()) THEN
        o→gen_rtti_code(); (* Generate runtime type information for exception types! *)
        output("TBLINST");    (* Allocate virtual method table *)
    ENDIF;
END;
```

15.6 Generating RTTI for ExceptionType's

For objects of type *ExceptionType* the operation *gen_rtti_code* is responsible for generating RTTI. The operation is also defined for objects, for which it does nothing. The pseudo-code for *ExceptionType's gen_rtti_code* operation is shown below:

```
gen_rtti_code()
BEGIN
    (* Specify user-specified name of exception type followed by length of name. *)
    output ("DB ' " + get_user_name() + "," + get_user_name()→get_length());
    (* Specify ancestor's RTTI addr *)
    output ("DW @TableAddr_" + get_ancestor()→get_name());
    output ("DB 0"); (* Allocate id byte for scratch-pad area *)
END;
```

Note that the *gen_rtti_code* operation is not used for generating RTTI for the pre-defined MEHL type *Exception*, which is declared in the runtime library, but only for user-specified (derived) exception types. Hence, all exceptions have an ancestor which is retrieved by the *get_ancestor* operation.

15.7 Generating code for the Exceptional classes

This section will discuss the code for the dag node related to EH as generated by the *gen_code* operation.

15.7.1 Exceptional

Below the pseudo-code for the abstract superclass *Exceptional's gen_code* operation can be observed. The operation generates the code for the parts of the static function tables which is common for the derived *gen_code* operation in the subclasses *Except* and *Finally*.

```
gen_code()
BEGIN
    (* regions = 1 + # guarded regions with exception handlers *)
    LET regions = get_try()→get_regions()→get_size();
    (* Specify # Guarded regions (1..255) *)
    To exception_regions_table output("DB " + regions);
    FOR EACH Region r IN get_regions() DO (* Specify ranges for guarded regions *)
        To exception_regions_table output ("DW " + r→get_start_label()
                                           "DW " + r→get_end_label());
    END;
    (* Specify except nest level (1..255) for construction: *)
    To exception_regions_table output("DB "  + get_except_nest_level());
END;
```

The above operation generates the parts of the static function table that describe the number of guarded regions, the ranges of the guarded regions and the except nest level for a TRY-EXCEPT or a TRY-FINALLY construction. The table entries are outputted to a temporary buffer called *exception_regions_table* which is used

on a procedure basis. The code in the temporary buffer is later retrieved by the *gen_code* operation for *Procedure*.

The operation *get_regions* is used to retrieve a list of all the guarded regions. The operation *get_size* is used on the list to retrieve the number of elements in the list. The operations *get_start_label* and *get_end_label* for *Region* is used to retrieve the labels marking the start and end of the code range for the region. The labels have previously been initialised while generating code for dags by the operation *gen_code* for the *Try* dag node.

15.7.2 Except

Below the pseudo-code for *Except's gen_code* operation can be observed.

```
gen_code()
BEGIN
    INHERITED→gen_code(); (* Call gen_code operation for Exceptional superclass *)
    (* Specify that its a TRY-EXCEPT construction with n handlers ! *)
    To exception_regions_table output("DB " + get_number_of_handlers());
    FOR EACH ExceptionHandler eh IN get_exception_handlers() DO
        eh→gen_test_code(); (* Specify handlers *)
    END;
END;
```

The above operation generates static table information that describe a TRY-EXCEPT construction. The first part of the static table is generated using the *gen_code* operation for the ancestor class. The remaining information specifies that the construction is a TRY-FINALLY construction, the number of handlers and individual information describing each handler. The operation *gen_test_code* for *ExeptionHandler* is used for describing each handler.

15.7.3 Finally

Below the pseudo-code for *Finally's gen_code* operation can be observed.

```
gen_code()
BEGIN
    INHERITED→gen_code(); (* Call gen_code operation for Exceptional superclass *)
    + (* Specify that its a TRY-FINALLY construction! *)
    To exception_regions_table output("DB 0");
    (* Specify finally nest level & *)
    To exception_regions_table output("DB " + get_finally_nest_level());
    (* Specify finally handler code addr *)
    To exception_regions_table output("DW " + get_finally_location()→get_label());
END;
```

The above operation generates static table information that describe a TRY-FINALLY construction. The first part of the static table is generated using the *gen_code* operation for the ancestor class. The remaining information which specify that the construction is a TRY-FINALLY construction, the nest level for the finally handler and the code-address of the handler is generated by the operation itself. Note that the table entries, as for *Exceptional's gen_code* operation, are outputted to a temporary buffer called *exception_regions_table.*

15.8 Generating code for the ExceptionHandler class

The operation *gen_test_code* generates code to describe an exception handler for the static table. The pseudo-code for the *gen_test_code* operation is shown below:

```
gen_code()
BEGIN
    (* Specify addr of user-handler and the number of exceptions handled by handler *)
    To exception_regions_table output("DW 0" + get_handler_location()→get_label() +
                                            "DB " + get_test_size());
    IF handler_kind = WITH_VAR THEN (* Argumented ? *)
        (* Argumented (1 exception handled) - Specify RTTI address of exception*)
        To exception_regions_table output("DW @TableAddr_" + get_type()→get_name());
    ELSE
        (* Not argumented (>=1 exception handled) - Specify list of RTTI addresses of
        exceptions*)
        FOR EACH ExceptionType e IN get_types() DO
            To exception_regions_table output("DW @TableAddr_" + e→get_name());
        END;
    ENDIF;
END;
```

The *gen_test_code* operation generates information which specifies the code-address for the exception handler, the number of exceptions handled by the handler and the RTTI addresses of the exceptions handled. The operation *get_handler_location* is used to retrieve the code-address for the handler, the operation *get_test_size* is used to retrieve the number of exceptions handled and the operations *get_type* and *get_types* are used to retrieve the *ExceptionType/ExceptionTypes* for the handler.

15.9 Code generated for individual dag nodes

This section will discuss the code for the dag node related to EH as generated by the *gen_code* operation. The code generated for each dag node will generally be optimal or near optimal (locally). As a consequence, most nodes must examine any arguments (child's of basetype *CommonNode*) they may have (0-2) to deduce the right code sequence to produce for that specific kind of operation. This means, that for some nodes the *gen_code* operation is more complicated then it could have been, had there been made no attempts to produce code of good quality. Also a factor which tends to complicate matters is the numerous limitations in the 80x86 instruction set all of which the various *gen_code* methods must reckon with.

15.9.1 Raise

Below the pseudo-code for *Raise's gen_code* operation can be observed.

```
gen_code()
BEGIN
    output("mov bx,offset +@Table_" + get_object_type()→get_name() +
            "|" + get_procedure()→get_name() + (* bx ← Addr of ctr of exception object *)
            "call RAISE"); (* Call runtime library routine for raising new exception*)
END;
```

15.9.2 ReRaise

Below the pseudo-code for *ReRaise's gen_code* operation can be observed.

```
gen_code()
BEGIN
     (* Call runtime library routine for re-raising current exception*)
    output("call RERAISE");
END;
```

15.9.3 Try

Below the pseudo-code for *Try's gen_code* operation can be observed.

```
gen_code()
BEGIN
    INC(trys_count); (* Global counter used by operation gen_code for Procedure *)
    FOR EACH Region r IN get_regions() DO (* Set labels for all guarded regions *)
        r→set_region_labels(); (* Set start and end labels for region r *)
    END;
    (* Output start label for outer (primary) region *)
    output("LABEL "  + get_region()→get_start_label() + " CODEPTR");
END;
```

The labels for all region guarded by a TRY construction are initialised by the above operation, but only a single code label, marking the start of the primary guarded region, is actually generated by this operation (Labels for remaining code regions are generated by *gen_code* operations for *StartExceptionalRegion* & *EndExceptionalRegion*). The operation *get_regions* is used to retrieve a list of all the guarded regions and the operation *get_region* is used to retrieve the primary guarded region. Note that as an important side effect a global counter of try constructions is incremented (for use by operation *gen_code* for *Procedure*).

15.9.4 EndTry

Below the pseudo-code for *EndTry's gen_code* operation can be observed.

```
gen_code()
BEGIN
    (* Output end label for outer (primary) region for TRY construction *)
    output("LABEL "  + get_region()→get_end_label() + " CODEPTR");
END;
```

A single code label, marking the end of the primary guarded region initiated by the corresponding *Try* dag node, is generated by this operation.

15.9.5 EndExceptHandler

Below the pseudo-code for *EndExceptHandler's gen_code* operation can be observed.

```
gen_code()
BEGIN
     (* push resume addr *)
    output("push offset "  + get_after_handler_location()→get_label());
    (* Terminate exception & goto resume addr *)
    output("jmp TERMINATE_EXCEPTION");
END;
```

15.9.6 StartExceptionalRegion

Below the pseudo-code for *StartExceptionalRegion's gen_code* operation can be observed.

```
gen_code()
BEGIN
```

```
    (* Output start label for region containing excepton handler(s) *)
    output("LABEL "  + get_region()→get_start_label() + " CODEPTR");
END;
```

The code generated for the *StartExceptionalRegion* dag node generates a label to mark the beginning of the code for nested exception handler(s) which are guarded by another TRY construction. The *Region* associated with *StartExceptionalRegion* is retrieved by the *get_region* operation. Recall that at the time of use the region's labels has been initialised by a previous TRY construction.

15.9.7 EndExceptionalRegion

Below the pseudo-code for *EndExceptionalRegion's gen_code* operation can be observed.

```
gen_code()
BEGIN
    (* Output end label for region containing excepton habdler(s) *)
    output("LABEL "  + get_start_except()→get_region()→get_end_label() +
            " CODEPTR");
END;
```

A single code label, marking the end of the guarded region of exception handler(s) initiated by the corresponding *StartExceptionalRegion* dag node, is generated by this operation.

15.9.8 EndFinally

Below the pseudo-code for *EndFinally's gen_code* operation can be observed.

```
gen_code()
BEGIN
    IF get_finally_nest_level()>0 THEN (* Test scheme used for the finally handler ? *)
        (* Test scheme used - Test for the event of an exception. *)
        (* check for match *)
        output("cmp [FINALLY_RAISE_NEST_LEVEL]," + get_finally_nest_level()"
        (* If no match, branch to resume addr *)
        "jb " + get_after_finally_location()→get_label());
    ENDIF;
    output("call CONTINUE_RAISE_FROM_FINALLY"); (* I.e. Re-Raise *)
    (* Output end label for handler if it is guarded (nested) and if it is
       copied outside normal program flow*)
    IF get_start_except() THEN
        INHERITED→gen_code(); (* Call EndExceptionalRegion superclass' gen_code *)
    ENDIF;
END;
```

The code for the *EndFinally* dag node vary accordingly to the optionally specified nest level for the finally handler (specified only if test scheme is used) and to its optional association with a *StartExceptinalRegion* (associated for guarded finally handlers only).

The nest level for the finally handler is retrieved by the *get_finally_nest_level* operation, if it is zero the value is not actually specified indicating that the copy scheme rather then the test scheme is used. If a test scheme is used the nest level is a positive (non-zero) value and code is generated to compare the counter *FINALLY_RAISE_NEST_LEVEL* against that value and if no match to branch to the resume addr for the handler (that is: the entry point for the code translated for the statements <u>after</u> the TRY-FINALLY construction). Then, regardless of translation scheme used, code is generated to re-raise the exception.

Last, if the finally handler is guarded by an outer TRY construction, the inherited *gen_code* operation for *EndExceptionalRegion* is called in order to generate a code label to mark the end of the guarded region of the finally handler initiated by the corresponding *StartExceptionalRegion* dag node.

15.9.9 Return

Below the pseudo-code for *Return's gen_code* operation can be observed

```
gen_code()
BEGIN
   IF get_left() THEN gen_move(ax,get_left()); (* Place optional return value in ax*)
   IF get_raise_return_exception() THEN (* What kind of return specified ? *)
      output("mov bx,offset @@FIN" + (* bx ← return addr entry point *) *)
            "call RAISE_RETURN");    (* Call runtime library routine for raising return
      exception *)
   ELSE (* Ordinary return - don't raise return exception *)
      output("jmp @@FIN");
   ENDIF;
END;
```

The code either jumps to the end of the function or raises an local raise return exception. The operation *get_raise_return_exception* is used to check what kind of return statement code is being generated for.

15.9.10 Call

Below the pseudo-code for the superclass *Call's gen_call_code* operation can be observed. The *gen_call_code* provides an operation which generates the common code needed by the *gen_code* operations for *ProcedureCall* and *FunctionCall*.

```
gen_call_code()
BEGIN
   IF is_member_call()THEN (* Call of method or ordinary call *)
      LET method = The Method associated with the procedure retrieved by get_procedure();
      IF NOT (current_procedure is a method for the same object as the method to call)
      THEN
         LET object = The dag node that refers to the object for which the method is being
         called.
      ENDIF;
      (* Use stub to call external function ? *)
      IF get_procedure()→generate_stub() THEN
         (* Virtual method call ? - If yes, lookup addr in VMT*)
         IF method→is_virtual() THEN
            (* bx ← Addr of VMT retrieved by object's internal ptr to its VMT:*)
            IF object defined AND object IS IdExpr v AND v IS GlobalVariable THEN
               (* Object is global variable so the addr of the VMT is simple to retrieve:*)
               output("mov bx,[v→get_name() + ".@Mptr_" + get_object_type + "]");
            ELSE
               (* bx ← Addr of object: *)
               IF object defined THEN
                  IF object IS IdExpr v THEN
                     v→gen_load_addr(bx); (* Object is a variable. Get its addr *)
                  ELSE
                     gen_move(bx,object); (* Object is expr. Get addr from dag node *)
                  ENDIF;
               ELSE
                  gen_move(bx,si); (* Object is current object.Get addr from THIS reg*)
```

```
                ENDIF;
                (* bx ← Addr of VMT for object: *)
                output("mov bx,[(" + get_object_type→get_name() +
                      "ptr bx).@Mptr_" + get_object_type()→get_name() + "]");
            ENDIF;
            (* bx ← Addr of method retrieved from VMT(!): *)
            output( "mov bx,[(@Table_" + get_object_type→get_name() +
                  "ptr bx)." + get_call_name() + "]");
        ELSE
             (* Non-virtual method called so the addr is known at compile time! *)
            (*  bx ← Address of method(!): *)
            output("mov bx,offset +@Table_" + get_object_type→get_name() + "|" +
            get_call_name());
        ENDIF;
        output("mov si," + get_procedure()→get_parameters_size()); (* si ←Size of
        parameters *)
        output("call EH_STUB"); (* Call stub *)
    ELSE (* No call stub call necessary - ordinary call of method without use of stub*)
        (* generate <CALL ... METHOD> assembler code instruction (not shown) *)
        ...
    ENDIF;
  ELSE (* Ordinary procedure/function call (not method) *)
    IF get_procedure→generate_stub() THEN (* Use stub to call external function ? *)
        output("mov bx,offset " + get_call_name()); (* bx ← Addr of function to call *)
        (* si ←Size of parameters *)
        output("mov si," + get_procedure()→get_parameters_size());
        (* Call stub code *)
        output("call EH_STUB");
    ELSE
        output("call " + get_call_name()); (* Call function without use of stub *)
    ENDIF;
  END;
  (* If call is located inside a method, reload designated object frame register *)
  IF current_procedure IS a Method THEN load_this_reg(); (* si ← [@@THIS] *)
END;
```

Observe that the code generated for a function call vary for calls to external functions with non-empty raises-sets and for calls to ordinary MEHL functions. The operation *generate_stub* is used to check for an external function with a empty raises-set. If such a function is found the code stub EH_STUB in the runtime library is called with the address of the function and the total size of parameters for the function as arguments in registers bx and si.

The total size of parameters for the function is easily retrieved by the operation *get_parameters_size* defined in *Procedure*. In comparison, the address of the function is not always as simple to retrieve. For ordinary functions and non-virtual methods the address is known at compile time (link time actually) and it can be retrieved by trivial use of the <OFFSET *procedure*> operator. For virtual methods the address must be looked up at run-time in the VMT pointed to by the object's VMT pointer.

16. Specification of tests & test summary

A large number of tests have been made for the MEHL compiler and the runtime library, but only these that are relevant to EH will be mentioned here. In main the tests are divided into tests of the MEHL compiler's front end & back end and tests of the EH runtime mechanism. The following three sections discuss the overall test strategy for these parts. The final section presents a summary of the result of the test runs.

16.1 Testing the MEHL compiler's front end.

The test of the front end includes tests of scanner and parser, validating the generated IR representation, type-checking and validating that error handling is performed correctly.

- The test of the scanner and parser includes testing tokens and productions respectively. For EH these tests involve the keywords and constructions used for EH.

- For validating the generated IR a compiler option is used which specifies that the MEHL compiler should dump a readable text representation of the IR to a file. To enabled this all object components in the IR have a *print* operation which outputs a representation of the object. Although the support for IR tests is made available, and has been used during development, it is of little conclusive use unless problems are detected in the generated code for the back end (which is not the case as mentioned later). Thus, no independent study of the generated IR for test purposes will be made for this thesis.

- In must be tested that the front end type-checks that exception handlers are specified in the correct order, with the most general exception last (handlers for sub-classes before handlers for super-classes). Also it must be tested that the compiler type checks raises-sets and multiple exceptions for an exception handler to make sure that no more than one specification of a given exception type exist (including reacting to the case that both an exception type and its super-type are specified). In Appendix H the result of type-check tests is shown.

.

16.2 Testing the MEHL compiler's back end.

The test of the MEHL compiler's back is a test of the generated code with respect to a known IR. For EH the test of the code generated includes tests of the code for constructions and statements which are relevant to EH in addition to tests of the static tables. The tests include the following subjects:

- The code and static tables generated for the constructions TRY-EXCEPT and TRY-FINALLY. Including the code generated for exception handlers.
- The code and static tables generated for RAISE statements.
- The code generated for exception objects.
- The table information generated for functions with raises-sets.
- The code labels and static information generated for construction and destruction of global, local, member objects and ancestor objects.
- The code generated for calls to external functions which may indirectly raise exceptions.

To present these tests formally in this thesis will be rather painstaking since the size of the generated code is very large. However, since the resulting MEHL program will not perform correctly if the wrong code is generated, these tests will be performed indirectly while testing the EH mechanism at runtime.

16.3 Testing the EH runtime mechanism.

The EH mechanism is tested by running a number of compiled MEHL programs which has been written for test purposes. The test programs have been supplied with statements for text output that describe the actions performed while running the program, such as raising an exception, handling an exception, destroying an object etc. The following has been tested by test programs:

For TRY-EXCEPT & TRY-FINALLY constructions:

- Raising of exceptions with/without parameters
- Transfer of control to a handler.
- The THIS register for handlers in methods is initialised correctly upon entry to the handler.

For TRY-EXCEPT construction:

- Execution of correct handler for exception
- Propagation of exceptions
- That exception variables for exception handlers refers to the exception being handled.
- Nested exception constructions (nested handlers, nested try's).
- Exception objects are destroyed when the handler is terminated.

For TRY-FINALLY construction:

- That finally statements are always executed.
- Nested finally constructions.

For object cleanup:

- Local objects
- Sub-objects (member objects and ancestor objects).
- Global objects
- Array of objects
- Heap objects

The most interesting tests and the results of running these tests are shown in Appendix H.

16.4 Summary of test results

A summary of the results for the tests performed is shown in figure 38. As the figure shows, all tests have been successfully! The summary does also include the large number of additional tests that was performed but which is not documented.

Parts	Test result
MEHL compiler's front end	√
MEHL compiler's back end	√
Runtime library / EH mechanism	√

Figure 38 Status of test results

17. Conclusion

This thesis has presented a number of methods for handling exceptions in modern object-oriented languages. Handling of exceptions which, as have been considered in this thesis, is synchronous, termination based, multi - level and general parameterized.- In the tradition of C++ and Modula-3.

It has been pointed out that methods for exception handling (EH) can be classified according to the approach used for *transfer of control* and *object cleanup*. Two alternative standard approaches to EH have been discussed which have been called *the static table approach* and *the registration approach*.

It has been shown that neither of these standard approaches to EH meet all the requirements for handling of exceptions specified in this thesis. The problem being the large ever-present runtime overhead for the registration approach and the missing support for mixed-language programming for the static table approach. For minimal runtime overhead combined with mixed-language support this thesis has purposed a new unified method based on a modified static table approach with support of mixed-language programming through registration of calls to external functions which may indirectly raise an exception.

A compiler for the object-oriented programming language MEHL featuring EH with the proposed unified method has been developed for this thesis. The design of the EH runtime mechanism and the compiler support for EH have been discussed in detail.

The size overhead must be considered for any efficient implementation of EH, and in particular when memory is as limited as it is for the target system (DOS). Noticeably, it has been shown how code inspection techniques can be used to minimise the potentially very large amount of storage space used for the static tables. figure 39 shows a summary of the amount of space used by the implementation for the static tables for characteristic and commonly used language components and an estimate of the reduction of storage space due to the use of the code inspection techniques.

Table space for each:	With code inspection (as is)	Without code inspection (estimate)	Reduction
Procedure / function	4 bytes	6 bytes	33 %
Constructor / destructor	4+1 bytes	6+1 bytes	29 %
Local destructible object or array of destructible objects	4 bytes	8 bytes (or more)	50 %
Destructible sub-objects[22]	8 bytes	16 bytes (?)	50 %

Figure 39 Summary of static table space for characteristic language components

No unique feature for the target architecture has been used in the implementation of EH and as a consequence the method and the principles used for the implementation in this thesis is as such possible to port. Though the method can be ported to other architectures, the implementation of the method is machine specific and must be reproduced for each specific architecture.

The MEHL compiler implementation has been written in the C++ programming language together with the *YACC*-compatible parser generator *Bison* and the fast lexical analyser generator *FLEX*. The runtime library has been written in 80286 assembler using the *Turbo Assembler*. figure 40 shows an overview of the size for different components of the MEHL compiler and runtime library.

To retarget the MEHL compiler the back end and the runtime library must be rewritten. For those components the implementation's EH support covers 500/2200 lines of source code (equal to 23% of total) for the compiler and 1000/1300 lines of source code for the runtime library (equal to 77% of total). Consequently EH, as implemented in the MEHL compiler, will be relatively expensive to port.

[22] For each constructor/destructor pair defined for the containing object.

	Front end	Back end	Runtime library & EH runtime mechanism
EH	600 lines (Mostly C++)	500 lines (C++)	1000 lines (Assembler)
Other parts	6600 lines (Mostly C++)	1700 lines (C++)	300 lines (Assembler)
In total	7200 lines / 230K	2200 lines / 70K	1300 lines / 30K

Figure 40 Size of implementation of Exception Handling versus other parts of the MEHL compiler and runtime library.

The unified method comes with a zero runtime overhead in the normal case, when no external functions are called! However when external functions are called, for which the declarations show that they may raise exceptions indirectly, a slight overhead is introduced due to the execution of code stubs for dynamic registration of each call. In the 80286 architecture targeted by the MEHL compiler the execution of the code stub used to call to an external function takes 69 machine cycles, 62 cycles more than the 7 cycles it takes to execute an ordinary function call. Still, on the whole the overhead is insignificant since calls to external functions are infrequent and external functions which may indirectly raise exceptions are even more infrequent.

The unified method delivers optimal performance as long as no exceptions are raised but when exceptions *are* raised its performance is rather poor (as for any method based on a static table approach). figure 41 shows the runtime overhead when raising exceptions in the MEHL compiler implementation for two short examples.

The first example is a MEHL program that contains a loop of 10,000 guarded calls of a function P1 that in turn calls a function P2 that in turn call another function P3 and so on, until a chain of 10 function frames is on the stack. Then, if testing for raising exceptions, an Exception is raised in P10 and handled by an exception handler contained in the main loop from where P1 was called.

The second example is a MEHL program that contains a loop of 10,000 guarded calls of a function P that contains two destructible objects. The function P, if testing for raising exceptions, raises an Exception that is handled by an exception handler contained in the main loop from where P was called.

10,000 x	Without raising exceptions	With raising exceptions	Raise overhead
Example 1 : 10 function calls in turn	48 ms.	566 ms.	1080 %
Example 2 : Call of function with 2 objects	33 ms.	378 ms.	1045 %

Figure 41 Overhead when raising exception in MEHL compiler implementation[23].

The first example is inspired by a similar example in [Cameron92]. Unfortunately no comparison can be made as the reference is unclear about the raise overhead. Comparative runs of two similar C++ examples on the *Borland C++ 4.0* compiler shows a raise overhead of 3700% and 4900% respectively for the two examples.

Although the runtime overhead when raising exceptions is substantial it is not that important as it has been the basic assumption for this thesis that EH is only used to handle (infrequent) errors. There is need for a study of programmers' use of EH to reveal whether this assumption is correct.

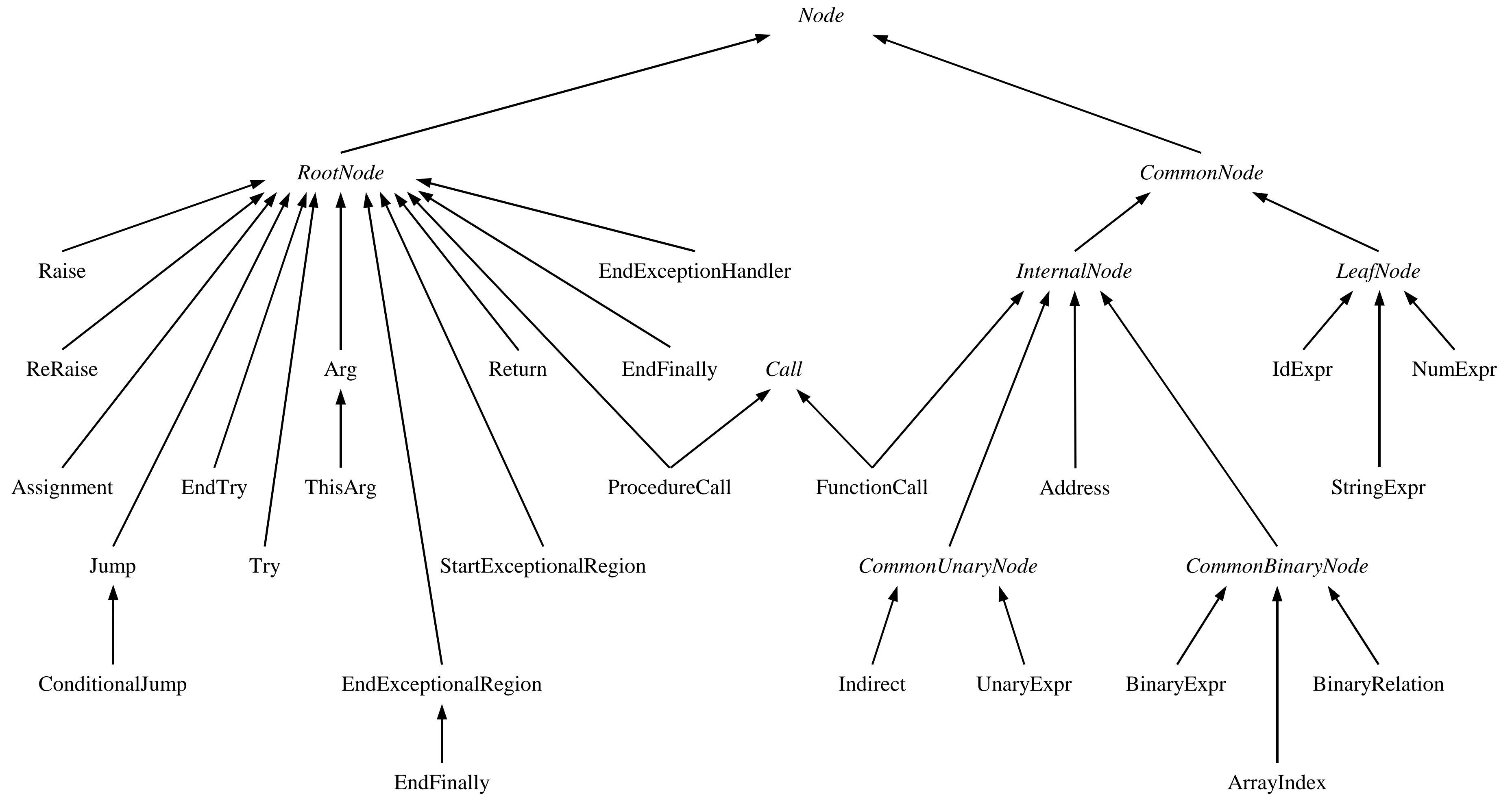
With respect to the initial requirements for EH the unified method is an optimal solution and as such there is little need for further theoretical work unless the requirements are changed or new non-standard architectures are targeted for which specialised solutions are applicable. However future work could be made in the design and implementation of EH. Most notable in trying to make EH support more or less retargetable and in improving the runtime performance of the raising of exceptions. Also the static tables for EH can be further reduced by using data compression techniques or by using inter-procedural analysis to eliminate unneeded entries.

In conclusion, this thesis has proposed a unified and very efficient method for handling exceptions in object-oriented languages. Efficient, but with a cost of poor performance when raising exceptions and being relatively

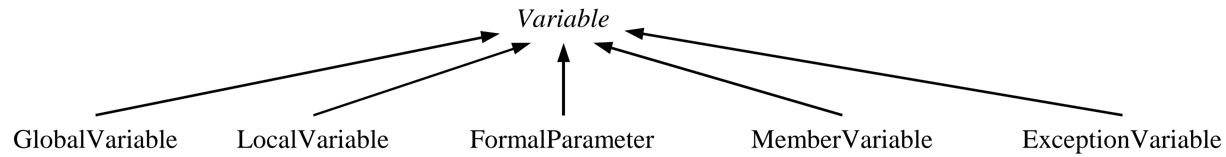
[23] All results are retrieved using *Turbo Profiler* running under DOS on a 66 Mhz 486DX2 PC.

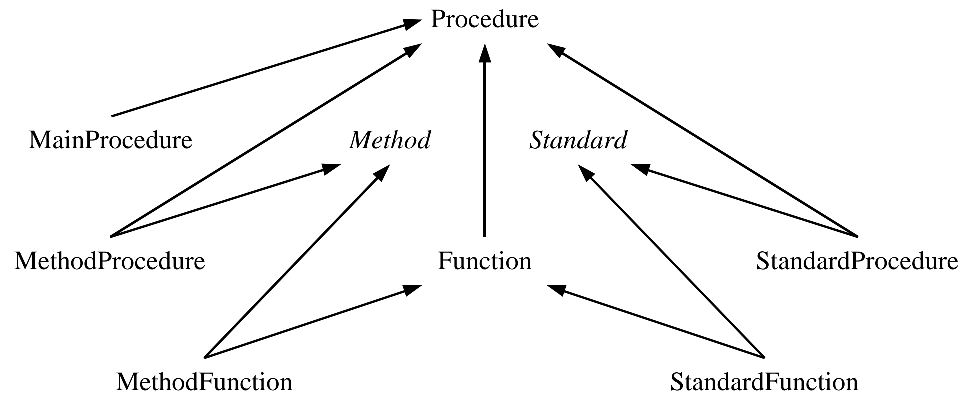
expensive with respect to the implementation effort required. Consequently, as all good things, an efficient method for handling exceptions comes with a price.

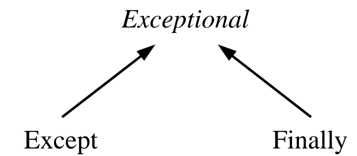
Appendix A - Node inheritance overview:

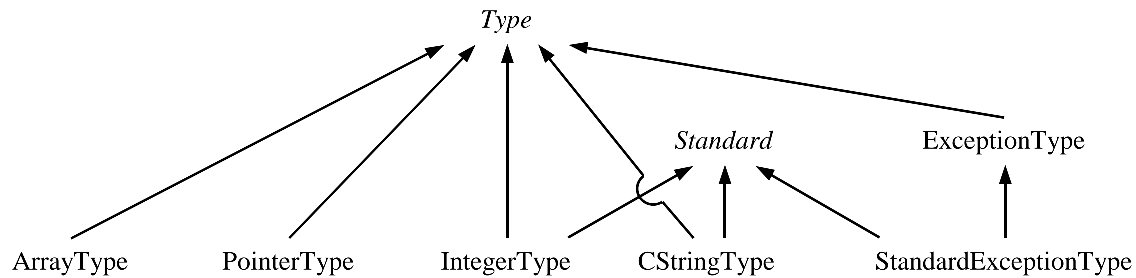


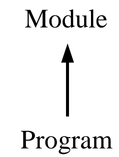
Various inheritance overviews:



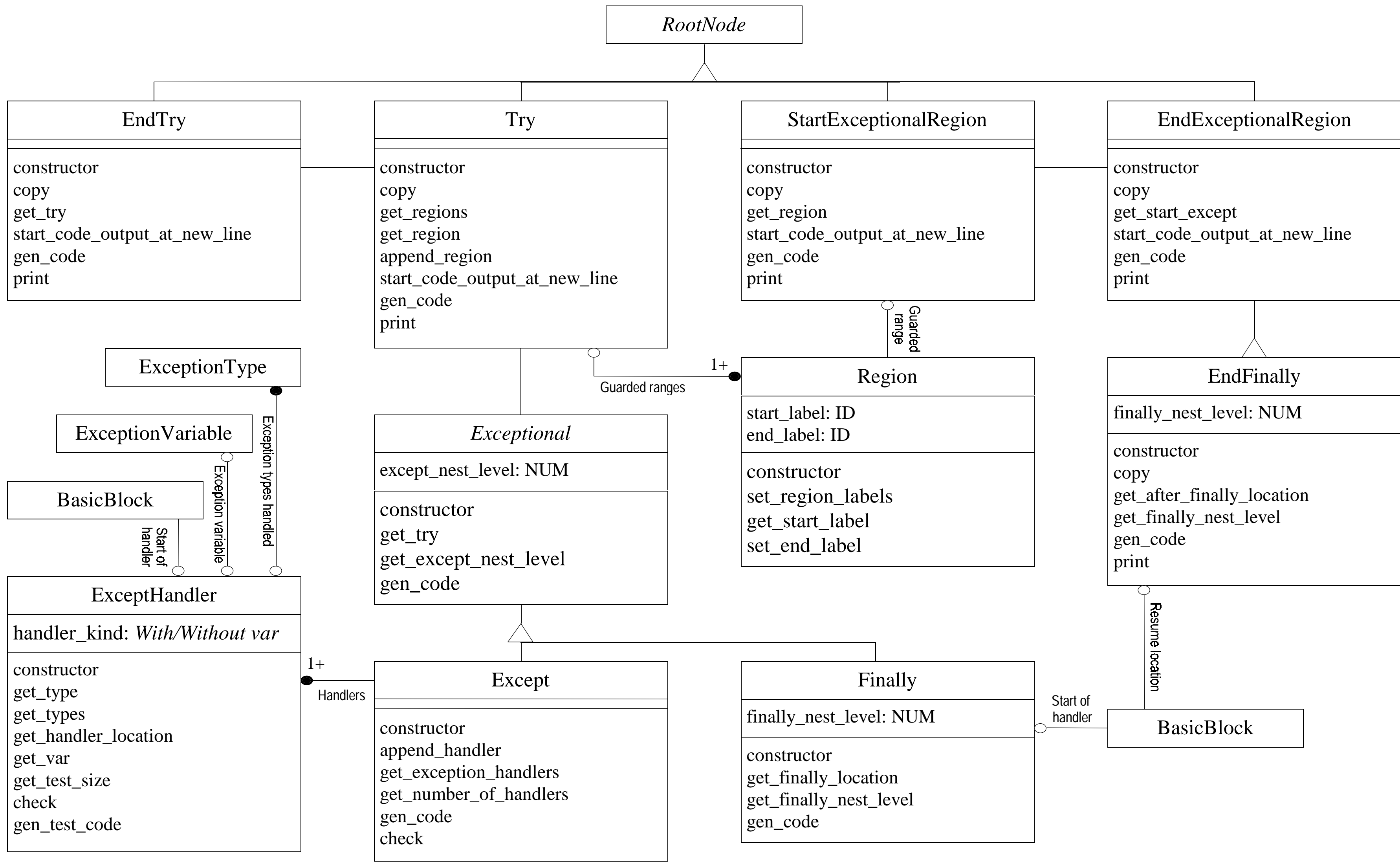




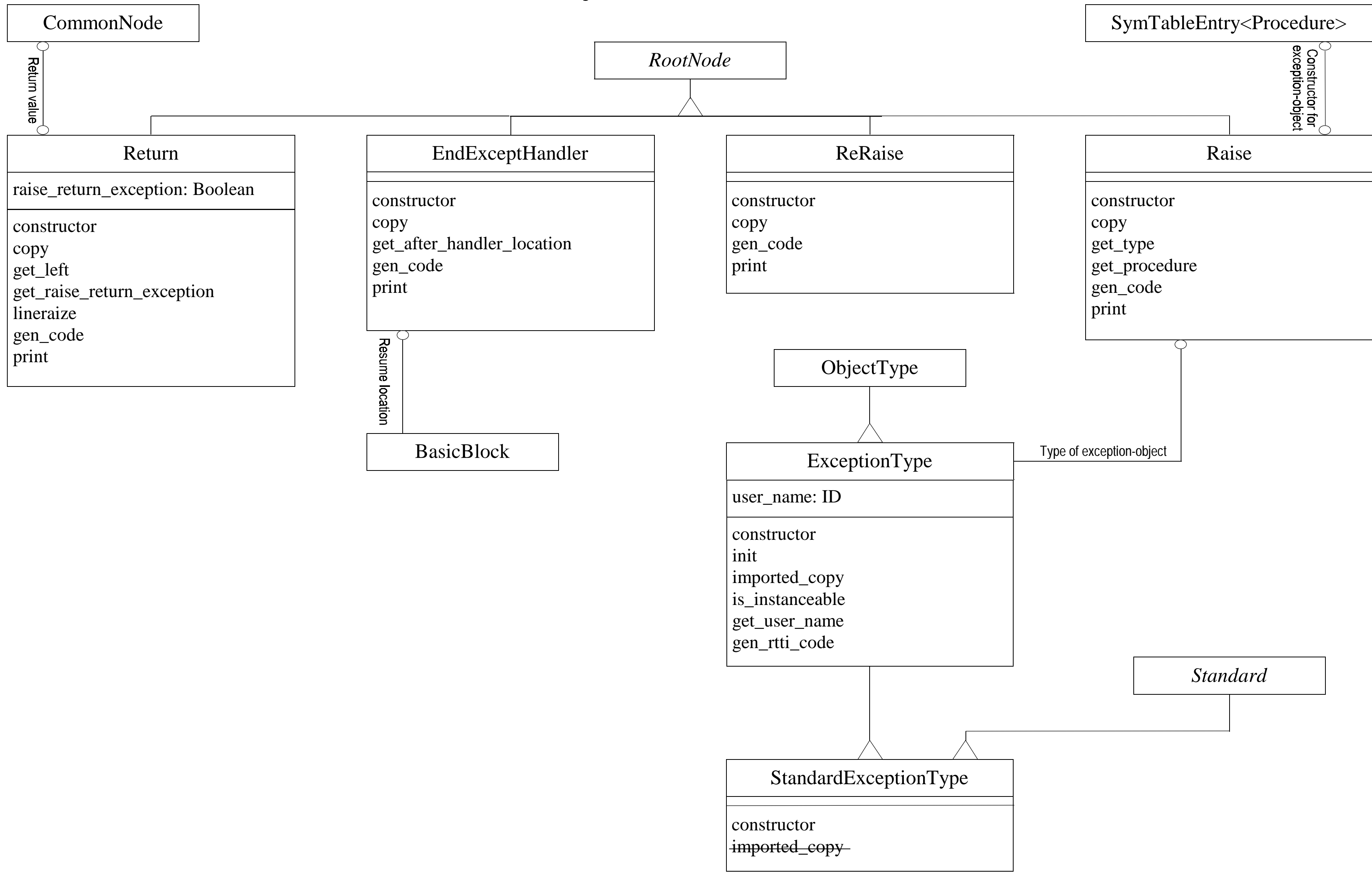




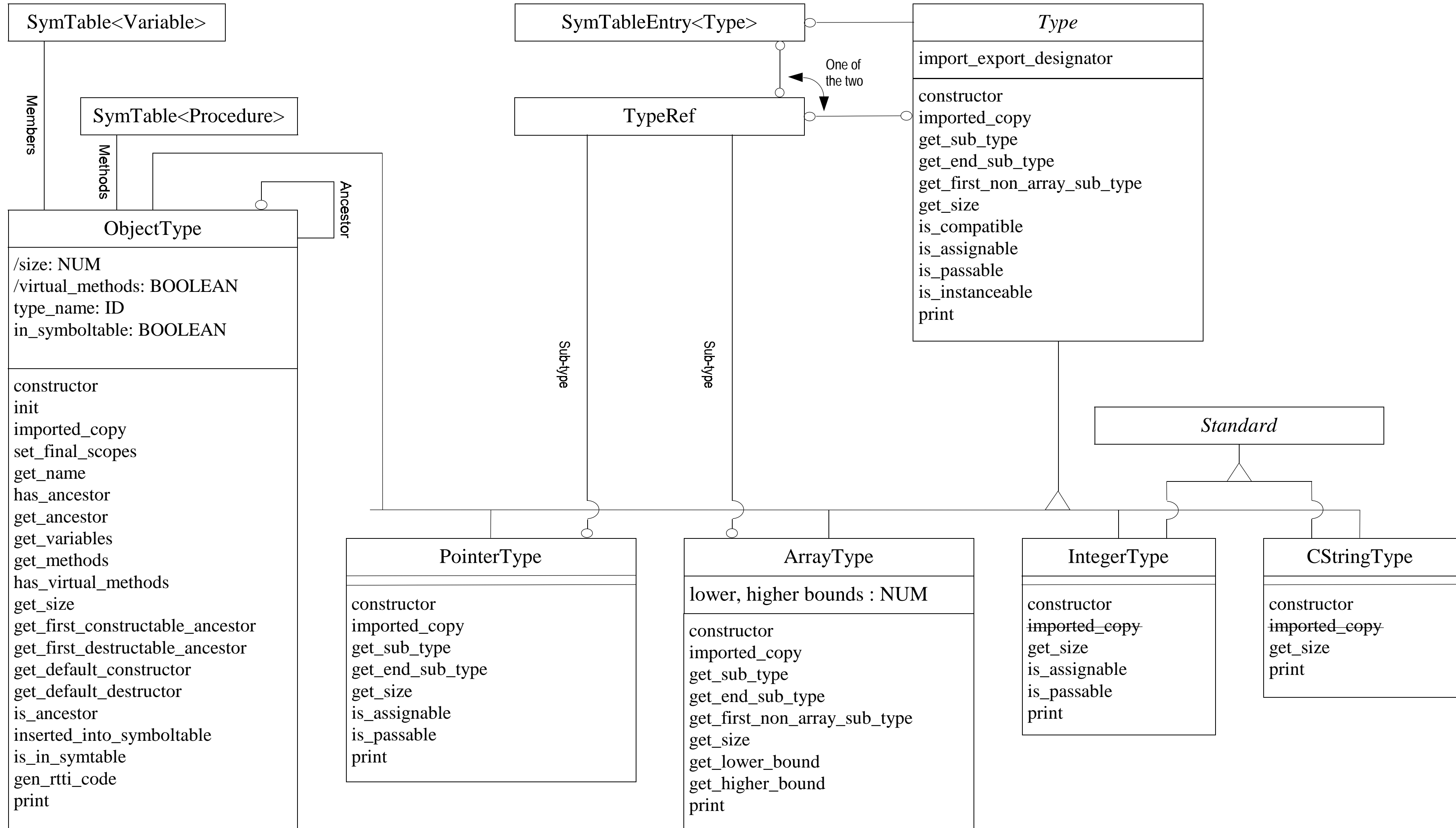
Appendix B - Object Model 1/5:



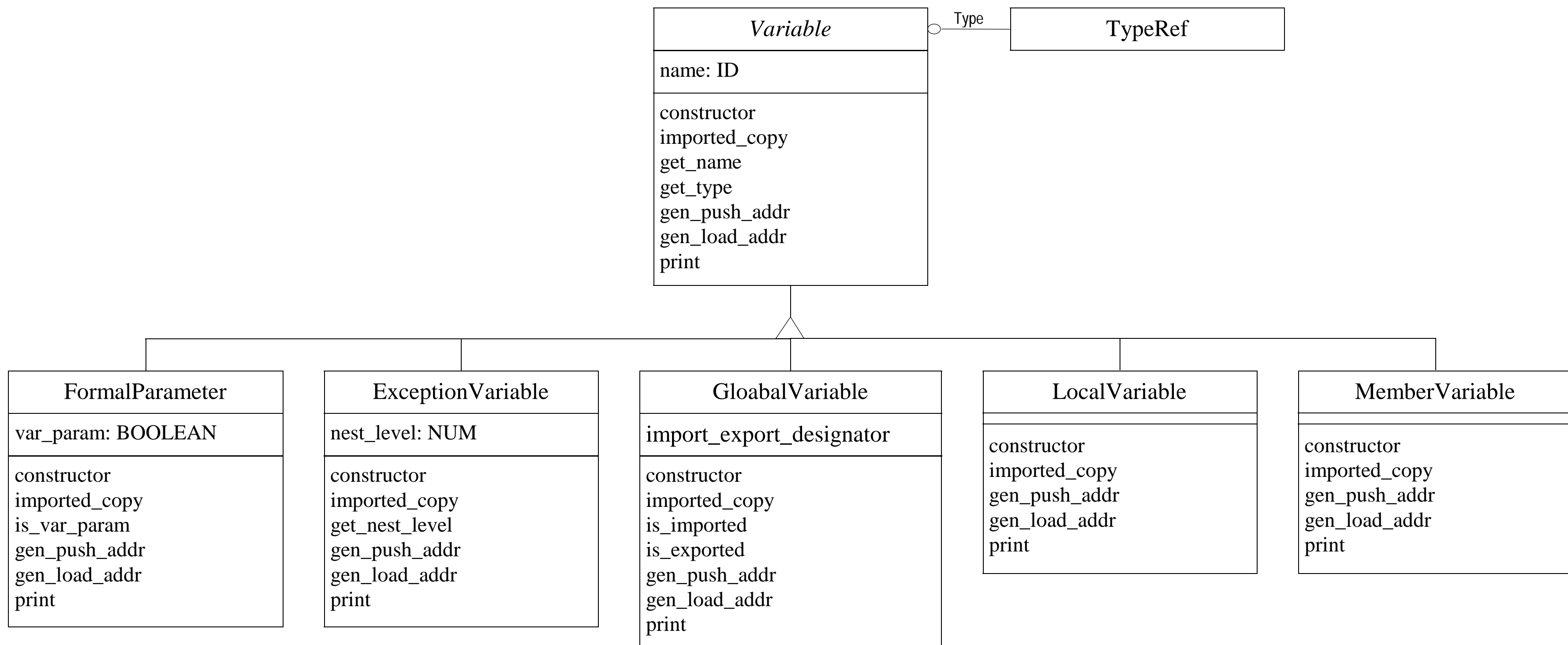
Object Model 2/5:



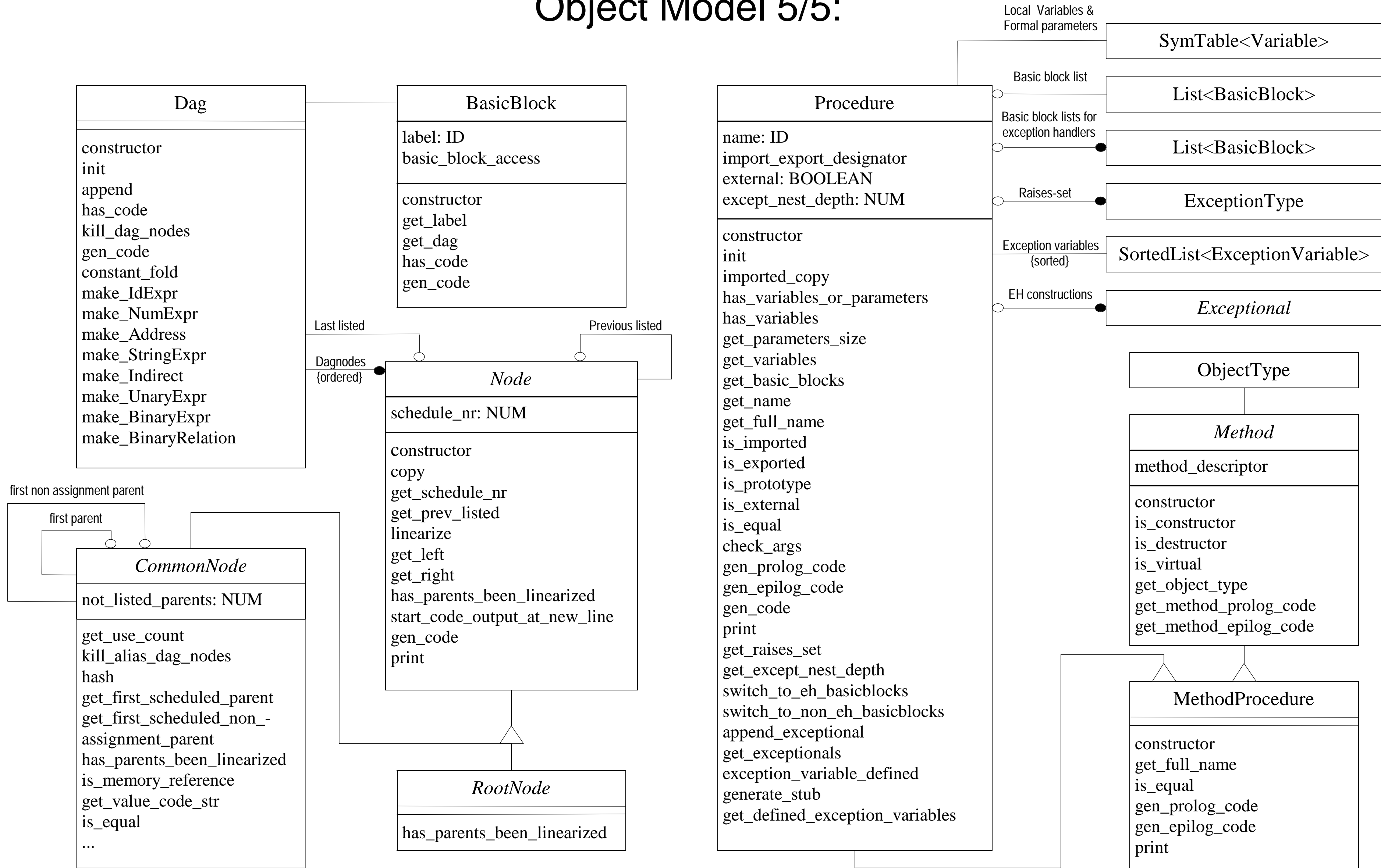
Object Model 3/5:



Object Model 4/5:



Object Model 5/5:



Appendix C The STDIO unit

This chapter presents the *stdio* unit which provides I/O support for MEHL programs.

Overview of the stdio unit

The *stdio* unit provides basic support for input and output of integers and strings in MEHL programs. Below the module declaration for the *stdio* unit can be observed:

```
UNIT stdio;

INTERFACE

PROCEDURE WriteInt(a:INTEGER) RAISES{}; EXTERNAL;
FUNCTION ReadInt: INTEGER RAISES{}; EXTERNAL;

PROCEDURE WriteString(s:PCSTRING) RAISES{}; EXTERNAL;
PROCEDURE ReadString(s:PCSTRING) RAISES{}; EXTERNAL;

IMPLEMENTATION

END.
```

Note, the stdio *unit* is an import unit, as all procedures/functions are declared external. In fact, all procedures/functions are implemented in assembler language.

Appendix D The MEHL runtime library

This chapter discusses the MEHL runtime library used in all MEHL programs. The runtime library is implemented in assembler language and it should be linked in with all MEHL programs. In addition to the EH mechanism the runtime library includes code for heap-management, for construction and destruction of arrays and internal routines for outputting error messages etc. The appendix will specify the general routines provided by this library apart from the EH mechanism which has been discussed elsewhere.

Heap-management

Among the most important internal operations provided by the runtime library are the heap management routines listed below:

- NEW(size) : addr
- DISPOSE(addr)
- GETFREE: size
- IS_HEAP_MEMORY(addr): bool

The runtime library maintains a linked list of allocated/freed memory blocks. The two heap management routines *NEW* and *DISPOSE* allocate and deallocate memory blocks from that list.

The function *NEW* allocates a specified number of bytes from the internal heap and returns a pointer to the allocated bytes. If the heap is fragmented it allocates memory from the smallest memory block. If the smallest memory block is larger then the memory block requested, it is split into two memory blocks, one block of the requested size and one surplus block.

The procedure *DISPOSE* deallocates the specified, previously allocated, block of memory. If the heap is fragmented and the freed memory block is a neighbour to another free memory block it joins the two blocks in the process.

The function *GETFREE* reports the number of free bytes on the heap. A simple counter of free memory is used to implement this function.

The function I*S_HEAP_MEMORY* is an internal function only. It is used internally by the EH mechanism to check whether a particular address belongs to the heap memory. To do that the function simply has to compare the address with the address range of the linked list of heap blocks.

Other internal routines

- ARRAY_CTR_APPLY
- ARRAY_DTR_APPLY

The *ARRAY_CTR_APPLY* and *ARRAY_DTR_APPLY* routines are used to construct and destroy arrays of objects. The pseudo-code for these routines is shown with the pseudo-code specification for the EH mechanism.

Appendix E Generating Code from Dags

The MEHL compiler is organized towards generating code using dags, directed acyclic graphs of computations. The object *Dag* contains an ordered sequence of roots in a dag made from the statements in its containing basic block. The dagnodes are organized into a hierarchy rooted by the abstract superclass *Node* which constitutes an object oriented dag language for the IR of MEHL statements.

The dags discussed in the compiler literature [Fisher91] and [Aho86] are composed in a slightly different way then in the MEHL compiler. In the literature, assignments in dags are represented by allowing a node to be labeled with the name of a variable. In contrast, assignments in the MEHL intermediate language are represented as distinct *Assignment* nodes.

Originally it was the intention to use the literature's representation of assignments by labeling nodes, but in this representation the general case of expressions involving registers on both sides of the assignment is not directly supported.

The representation of assignments as dag nodes guarantees that registers on both sides of an assignment are distinct and that they yield the correct values at the time of the assignment. The introduction of an *Assignment* node to represent assignments has been inspired by [Fraser91].

The algorithm

The generation of code from a dag is done with the help of the heuristic algorithm in [Fisher91, section 15.7]. The algorithm consists of four steps:

1. Schedule nodes.
2. Allocated virtual registers to nodes.
3. Map virtual registers to real registers.
4. Generating code for nodes.

The first step, the scheduling of nodes, specifies in which order the nodes should be evaluated. The next step is the assignment of virtual registers to all dag nodes that employ a register (i.e. those nodes that originate from *InternalNode*). In the third step the virtual registers are mapped to hardware registers. Finally, in the last step, code for the dag is generated by calling *gen_code* in turn on each dag node scheduled.

Note that although the dags in the MEHL IR, due to the *Assignment* node, aren't composed exactly like the dags in [Fisher91], no changes are necessary in the algorithm.

Supporting the algorithm

To facilitate the generation of code from dags using the algorithm presented in the previous section a number of associations, attributes and methods have been formed.

The object Node

Associations:

- prev_listed_node: Node

Facilitate a <u>linearization</u> of nodes, ordered in correspondence to their schedule numbers, which enables efficient iteration upon the scheduled nodes.

Attributes:

- schedule_nr: Integer

Each node must include a schedule number to enable the scheduling of two nodes to be compared efficiently.

Operations:
- linearize(Node prev, Integer current_count)
- has_parents_been_linearized: Boolean
- get_prev_listed: Node

The method *linearize* is responsible for the scheduling (and linearization) of nodes in a dag. It schedules a node and its children in a pre-order fashion. All parents must be scheduled before a child can be scheduled.- This can be checked by the method *has_parents_been_linearized.* The method *get_prev_listed* returns the associated node *prev_listed_node*.

An algorithm for the linearize operation can be observed below:

```
linearize(Node prev, VAR Integer schedule_count)
BEGIN
 IF (schedule_nr undefined AND has_parents_been_linearized) THEN
  LET schedule_nr=schedule_count
  LET schedule_count=schedule_count+1
  LET prev_listed=prev
  return linearize_childs(this node,schedule_count)
 END
 RETURN prev;
END
```

With the linearize method at hand, the total linearization of a dag can be handled in the following way:

LET last_listed = rightmost root-node in list contained in object *Dag*.
Iterate backwards upon dagnode roots in *Dag (*starting from rightmost node).
LET last_listed=n:linearize(last_listed,1*),*
where n is the current dagnode root.

The object CommonNode

Associations:
- first_parent: Node
- first_non_assignment_parent: Node

When allocating virtual registers it is necessary to identify the first scheduled parent of a node. This operation is enabled efficiently by each *CommonNode* having an association to its first scheduled parent. When generating code for an assignment it is necessary to check if there is a node scheduled before the assignment, that needs the initial value of the left side. An efficient performance of this operation is archived by each *CommonNode* having an association to its first scheduled parent aside from any *Assignment* node.

Attributes:
- not_listed_parents: Integer

A *CommonNode* can have any number of parents, so the method h*as_parents_been_-linearized* must have help in order to be able to give an answer. The attribute *not_listed_-parents* is used to keep count of parents not yet scheduled. When the dag is initially constructed this attributes must be correctly initialized to the total number of parents, whenever a parent is scheduled this attribute is decremented.

Operations:
- notify_a_parent_has_been_linarized
- get_first_scheduled_parent
- get_first_scheduled_non_assignment_parent

The operation *notify_a_parent_has_been_linarized* is used to decrement the *not_listed-_parents* attribute. Whenever a parent has been scheduled, it is called on its children. It is also the responsibility of the method *notify_a_parent_has_been_linarized* to set up the associations *first_parent* and *first_non_assignment_parent.*

The operations *get_first_scheduled_parent* and *get_first_scheduled_non_assignment_parent* returns the respective node associations *first_parent* and *first_non_assigment_parent*.

The object InternalNode

Attributes:
VirtualRegister v_reg / Register reg

The attribute *v_reg* is used to keep the virtual register assigned during step 2 of the algorithm. The attributed *reg* is used to hold the real register assigned to the node. As the virtual register and the actual register are not used at the same time the two attributes can be contained in a union.

Operations:
set_virtual_reg(VirtualRegister reg)
get_virtual_reg: VirtualRegister
has_virtual_reg: Boolean
set_reg(RegisterType reg)
get_reg: RegisterType

The methods above can set, inspect and check the virtual register and set and inspect the actual register.

Appendix F The 80286 processor

This appendix will present a very short introduction to the INTEL 80286 processor and discuss the memory model used for the MEHL compiler. The INTEL 80286 is basically a 16-bit processor. It can operate in different configurations, but only the basic real-mode, used in DOS are used.

The 80286 Registers

Memory is in the INTEL 80x86 architecture organized into segments, which each can be up to 64K long. The segment registers DS, CS, SS, ES each point to a segment. The segment register DS is a data segment register, which points to global data in a program.- All ordinary data memory references (direct, indirect general register etc.) are as default interpreted as 16 bit offsets from DS. CS is a code segment register that points to the machine code instructions in a program. Register SS is a stack segment register that points to the program stack. Segment register ES has no designated use.

The 80286 has eight general-purpose registers named AX, BX, CX, DX, SI, DI, BP and SP. Each of the general-purpose registers can store any 16 bit value, can be loaded from- and written to memory, and can be used in arithmetic and logical operations. However, some general registers have predetermined uses and the remaining general registers have different capabilities.

The general register SP is the stack register and BP is a designated stack frame register. The register AX is known as the accumulator.- It contains the first operand for all multiplications and divisions. Also importantly, only the registers BX, SI, DI, BP can be used with indirect and indexed-indirect addressing.

Common 80286 instructions

The most common instructions in 80286 assembler language include:

- MOV dest., source (copy source into dest)
- LEA dest., op (yields always address (offset) of op.)
- PUSH op
- POP op
- CMP op1, op2 (compare op1 with op2)
- JE (jump equal), JNE (jump not equal), JA (jump above), JB (jump below) dest.
- JMP dest. (jump to routine).
- CALL (call subroutine)
- RET (return from subroutine).
- ADD dest., source
- SUB dest., source

- MUL op
- DIV op
- NEG op (change sign).
- ENTER size of local data (make, enter stack frame)
- LEAVE (remove, exit stack frame)

Note that in the INTEL assembler syntax the destination register is always specified before the source! For two operand instructions, only one operand can be a memory reference.

The multiplication of two 16 bit operands yields a 32 bit result. The MUL instruction, when doing 16 bit multiplication, expect one operand in AX. It saves the upper 16 bit result in DX and the lower 16 bit result in AX.

In 16 bit divisions the dividend is always 32 bit. The DIV instruction expects the upper 16 bit value of the dividend in DX and the lower 16 bit of the dividend in AX. The result 16-bit quotient is placed in AX, and the 16-bit remainder in DX.

Memory Models

Several alternative models for organizing memory exist for INTEL 80x86 programs. Depending on the data, stack- and code needs of a program, different ways of combining the different segments, which constitute a program, can be decided upon.

For the MEHL compiler a small memory model has been targeted. In this model all the program code in the code segment(s) resigns in one physical segment. The global data in the data segment(s) and the program stack in the stack segment reside in one mutual, physical segment.

The MEHL compiler's heap manager will allocate its data from the joint data- and stack physical segment.- Thus, in the model used, all data references, regardless of scope or origin, will refer to the same segment.

Appendix G The Turbo Assembler language

This appendix will present an introduction to the the features used in Borland Turbo Assembler which is unusual for assemblers and which is used by the MEHL compiler's code generator

Declaring and using procedures

The *PROC..ENDP* directives are used to declare procedures and functions, including their optional parameters and local variables. Below the basic syntax for the procedure construction is presented:

```
PROC name
[ARG id “:” type_id {“,” id “:” type_id} ]
[LOCAL id “:” type_id {“,” id “:” type_id}]
<... code ...>
ENDP name
```

Type_id can be BYTE, WORD or any user defined type-names. *Name* is the unique name for the procedure and *id* is a local name for a parameter or local variable. Any *id* prefixed with the two characters "@@" and specified inside the *PROC..ENDP* directives will be locally scoped.

Formal parameters and local variables are specified using the *ARG* and *LOCAL* constructions respectively. Each formal parameter and local variable will be assigned unique spaces within the procedure's activation record which is located on the stack. There is no specification for return values, as the common practice for the INTEL 80x86 architecture is to return values from functions in registers!

Internal prolog and epilog code will be included automatically in the procedure's body. The prolog and epilog code will setup and restore the stack frames, by saving and restoring the frame pointer register BP and reserving space for local variables on the stack. A reference to a formal parameter or a local variable is implicitly converted to a corresponding indexed-indirect address reference using the frame pointer, which must be reserved for that use, solely. The procedures and functions specified with the *PROC..ENDP* directives can be called by an ordinary CALL instruction. Any arguments to the procedure or function must be pushed on the stack prior to the call. When the PASCAL calling convention is used, the selfsame arguments will be removed automatically from the stack by the epilog code.

Declaring and using objects

The STRUC directive can be used to implement structures and objects. Below the basic syntax for the structure and object constructions are presented:

```
STRUC name [ancestor_name]
 [ METHOD “{”
  {[VIRTUAL] proc_id “:” WORD “=” impl_proc_id}
 “}” ]

 {var_id var_type default_value}
ENDS name
```

Name designates the name of the structure or object, *ancestor_name* the name of an optional ancestor object. For objects a list of static and/or virtual methods can be supplied. *Proc_id* is the name of a method and is always the name used when calling a method.

Impl_proc_id is an internal name for the global procedure that implements a specific method. To avoid confusion when multiple subclasses have their own implementations of a particular method, it is common practice to let: *impl_proc_id = name+"_"+ proc_id.* When the address of a particular implementation of a

method is needed, the recommended way of retriving the address, rather then take the address of *impl_proc_id*, is to use the expression: *"+@Table_"+object_name+" / "+ method_name.*

Var_id designates the name of a variable member of the structure or object. *Var_type* and *default_value* specify the type of the member and the default value for the member. Both must be specified in the same way as global variables.

For both methods and data members, names in structures and objects are always unique to the structure / object it belongs to (unless inheritance is used).

If an object has any virtual methods the assembler will construct a virtual method table for the object type and reserve place for a hidden member pointer to the table in the object. Each virtual method table is a list of addresses of implementation procedures, one address for each virtual method. A call of a virtual method is done indirectly using the virtual method table, pointed to by each instance of an object with its virtual method table pointer. Static (i.e. non-virtual) methods are called directly, as the addresses of their implementation procedures are always known at link-time.

Any type of object with virtual methods requires one instance of the corresponding virtual table located somewhere in the program. This is taken care of with the *TBLINST* directive. If the declaration of an object is followed by a *TBLINST* directive, room for the virtual method table for the corresponding type of object will be reserved automatically in the data segment.

The hidden virtual table member pointer, implicitly included in an object with virtual method(s), is named *"@Mptr"+object_name*. It is the programmer's responsibility, to ensure that the virtual table pointer is initialized, before any virtual methods are called. To do that, it is common practice for an object to supply a static initialization method, i.e. a constructor.

The recommended way of calling methods for objects is to use the *CALL..METHOD* construction, which generates code to call both static and virtual methods. The syntax for the *CALL..METHOD* construction is as follows:

CALL instance METHOD object_name ":" method_id USES register

Instance is the address of the instance of the object for which the method is to be called. *Object_name* is the name of the object's type and *method_id* the name of the method to be called. *Register* specify any general index register. It is used internally by the small code sequence generated when calling a virtual method using *CALL..METHOD*. When calling a static method, *register* is not used, and *CALL..METHOD* amounts to an ordinary CALL instruction.

Appendix H Tests and test results

This appendix presents the most interesting subset of the tests made on the MEHL compiler and its generated program (with respect to EH).

Testing type-checks in the MEHL compiler

The MEHL example below checks the compiler's response to type errors in raises-sets and exceptions specified for exception handlers.

```
PROGRAM type_check; (* Example with 3 type errors *)

TYPE
 ExceptionA = OBJECT(Exception) END;
 ExceptionA1 = OBJECT(ExceptionA) END;

PROCEDURE raises_set() RAISES {ExceptionA, ExceptionA1};

BEGIN
END;

PROCEDURE handlers();

BEGIN
 TRY
  raises_set();
 EXCEPT
  | ExceptionA, Exception =>
  | ExceptionA1 =>
 END;
END;

BEGIN
END.
```

Test output:

The MEHL program contains 3 errors. When testing, previously reported errors are corrected before recompiling the program. When compiling, the following 3 errors are reported in turn:

```
1. Error 'Over-specification in raises-set' in line 7 in
   file typ_chk.mod
2. Error 'Over-specification in exception handler' in line 19 in
   file typ_chk.mod
3. Error 'One or more exceptions already handled by previously specified
   exception handler(s)' in line 20 in file typ_chk.mod
```

Testing Exception Handling

The test programs below checks the handling of exceptions at runtime. The test outputs shown have been formatted for reasons of readability. All tests has performed as they should (eventually).

Test for nested try-except constructions, etc.

The test below tests for nested try-exception constructions, for destruction of exception objects and for exception variables.

```
PROGRAM nested_except_handlers;

USES Stdio;

TYPE
 E = OBJECT (Exception)
      raise_point: INTEGER;
      CONSTRUCTOR init(i: INTEGER); FORWARD;
      PROCEDURE spec_raiser(); FORWARD;
      DESTRUCTOR done(); VIRTUAL; FORWARD;
     END;

CONSTRUCTOR E.init(i: INTEGER);
BEGIN
 raise_point:=i;
 WriteString('Constructor for E, raised at raise_point ');
 WriteInt(raise_point);
 WriteString(', called');
END;

DESTRUCTOR E.done(); VIRTUAL;
BEGIN
 WriteString('Destructor for E, raised at raise_point ');
 WriteInt(raise_point);
 WriteString(', called');
END;

PROCEDURE E.spec_raiser();
BEGIN
 WriteString('The exception E being handled was raised from
             raise-point ');
 WriteInt(raise_point);
END;

PROCEDURE test(i: INTEGER);
BEGIN
 WriteString('Raise point 0 reached');
 IF i=0 THEN RAISE E.init(i); ENDIF;
 TRY
  WriteString('Try block entered - Raise point 1 reached');
  IF i=1 THEN RAISE E.init(i); ENDIF;
  TRY
   WriteString('Try block entered - Raise point 2 reached');
   IF i=2 THEN RAISE E.init(i); ENDIF;
  EXCEPT
   | E(a) => WriteString('Exception handler entered - Raise point 3
                          reached');
             a.spec_raiser();
             WriteString('New input location (>=3) to generate
                          exception ?');
             i:=ReadInt();
             IF i=3 THEN RAISE E.init(i); ENDIF;
             TRY
              WriteString('Raise point 4 reached');
              IF i=4 THEN RAISE E.init(i); ENDIF;
             EXCEPT
              | E(b) => WriteString('Exception handler entered -
                                     Raise point 5 reached');
                        b.spec_raiser();
                        WriteString('New input location (>=5) to
                                     generate exception ?');
                        i:=ReadInt();
                        IF i=5 THEN RAISE E.init(i); ENDIF;
             END;
             WriteString('Raise point 6 reached');
             IF i=6 THEN RAISE E.init(i); ENDIF;
  END;
  WriteString('Raise point 7 reached');
```

```
  IF i=7 THEN RAISE E.init(i); ENDIF;
 EXCEPT
 | E(c) => WriteString('Exception handler entered -
                        Raise point 8 reached');
           c.spec_raiser();
           WriteString('New input location (>=8) to generate
                        exception ?');
           i:=ReadInt();
           IF i=8 THEN RAISE E.init(i); ENDIF;
           TRY
            WriteString('Try block entered - Raise point 9 reached');
            IF i=9 THEN RAISE E.init(i); ENDIF;
           EXCEPT
            | E(d) => WriteString('Exception handler entered -
                                   Raise point 10 reached');
                      d.spec_raiser();
                      WriteString('New input location (>=10) to
                                   generate exception ?');
                      i:=ReadInt();
                      IF i=10 THEN RAISE E.init(i); ENDIF;
           END;
           WriteString('Raise point 11 reached');
           IF i=11 THEN RAISE E.init(i); ENDIF;
 END;
 WriteString('Raise point 12 reached');
 IF i=12 THEN RAISE E.init(i); ENDIF;
END;

BEGIN
 TRY
  WriteString('Input location to generate exception ?');
  test(ReadInt());
 EXCEPT
  | E(e) => WriteString('Main exception handler executed');
            e.spec_raiser();
 END;
END.
```

Test outputs:

Test run for raising exception at raise-point 0:

```
Input location to generate exception ? 0
Raise point 0 reached
Constructor for E, now raised at raise_point 0, called
Main exception handler executed
The exception E being handled was raised from raise-point 0
Destructor for E, raised at raise_point 0, called
```

Test run for raising exceptions at raise-points 1,9,10:

```
Input location to generate exception ? 1
Raise point 0 reached
Try block entered - Raise point 1 reached
Constructor for E, now raised at raise_point 1, called
Exception handler entered - Raise point 8 reached
The exception E being handled was raised from raise-point 1
New input location (>=8) to generate exception ? 9
Try block entered - Raise point 9 reached
Constructor for E, now raised at raise_point 9, called
Exception handler entered - Raise point 10 reached
The exception E being handled was raised from raise-point 9
New input location (>=10) to generate exception ? 10
Constructor for E, now raised at raise_point 10, called
Destructor for E, raised at raise_point 9, called
Destructor for E, raised at raise_point 1, called
Main exception handler executed
The exception E being handled was raised from raise-point 10
Destructor for E, raised at raise_point 10, called
```

```
Test run for raising exceptions at raise-points 2,3,8:
Input location to generate exception ? 2
Raise point 0 reached
Try block entered - Raise point 1 reached
Try block entered - Raise point 2 reached
Constructor for E, now raised at raise_point 2, called
Exception handler entered - Raise point 3 reached
The exception E being handled was raised from raise-point 2
New input location (>=3) to generate exception ? 3
Constructor for E, now raised at raise_point 3, called
Destructor for E, raised at raise_point 2, called
Exception handler entered - Raise point 8 reached
The exception E being handled was raised from raise-point 3
New input location (>=8) to generate exception ? 8
Constructor for E, now raised at raise_point 8, called
Destructor for E, raised at raise_point 3, called
Main exception handler executed
The exception E being handled was raised from raise-point 8
Destructor for E, raised at raise_point 8, called

Test run for raising exceptions at raise-points 2,4,5,9,10:
Input location to generate exception ? 2
Raise point 0 reached
Try block entered - Raise point 1 reached
Try block entered - Raise point 2 reached
Constructor for E, now raised at raise_point 2, called
Exception handler entered - Raise point 3 reached
The exception E being handled was raised from raise-point 2
New input location (>=3) to generate exception ? 4
Raise point 4 reached
Constructor for E, now raised at raise_point 4, called
Exception handler entered - Raise point 5 reached
The exception E being handled was raised from raise-point 4
New input location (>=5) to generate exception ? 5
Constructor for E, now raised at raise_point 5, called
Destructor for E, raised at raise_point 4, called
Destructor for E, raised at raise_point 2, called
Exception handler entered - Raise point 8 reached
The exception E being handled was raised from raise-point 5
New input location (>=8) to generate exception ? 9
Try block entered - Raise point 9 reached
Constructor for E, now raised at raise_point 9, called
Exception handler entered - Raise point 10 reached
The exception E being handled was raised from raise-point 9
New input location (>=10) to generate exception ? 10
Constructor for E, now raised at raise_point 10, called
Destructor for E, raised at raise_point 9, called
Destructor for E, raised at raise_point 5, called
Main exception handler executed
The exception E being handled was raised from raise-point 10
Destructor for E, raised at raise_point 10, called
```

Test of nested finally constructions

The test below tests for nested finally constructions.

```
PROGRAM finally_test;
USES stdio;
PROCEDURE g();
BEGIN
 TRY
  WriteString('In try for procedure g');
 FINALLY
  WriteString('In finally statements for finally handler for g');
 END;
END;
PROCEDURE f();
BEGIN
 TRY
  WriteString('In start of try 1 for procedure f');
  RAISE Exception;
  WriteString('In start of try 1 for procedure f');
 FINALLY
  WriteString('In start of finally statements for outer finally
               handler for f');
  TRY
   WriteString('In start of try 2 for procedure f');
   g();
   WriteString('In end of try 2 for procedure f');
  FINALLY
   WriteString('In finally statements for inner finally handler for f');
  END;
  WriteString('In end of finally statements for outer finally
               handler for f');
 END;
END;
BEGIN
 TRY
  f();
 EXCEPT
  | Exception => WriteString('Exception handled');
 END;
END.
```

Test output:

```
In start of try 1 for procedure f
In start of finally statements for outer finally handler for f
In start of try 2 for procedure f
In try for procedure g
In finally statements for finally handler for g
In end of try 2 for procedure f
In finally statements for inner finally handler for f
In end of finally statements for outer finally handler for f
Exception handled
```

Test for propagation, external function calls and exception handling :

The test below tests for the propgation of an exception through a call chain of three function calls A, B & C for which function B is an external function written in the C programming language!

PROGRAM ctest; # File ctest2.mod

```
USES ctest1,stdio;

BEGIN
 TRY
  A();
 EXCEPT
  | Exception => WriteString('Exception caught and handled');
 END;
END.
```

UNIT ctest1; # File ctest1.mod

```
INTERFACE

PROCEDURE A(); FORWARD;
PROCEDURE B(); EXTERNAL;
PROCEDURE C(); FORWARD;

IMPLEMENTATION

USES Stdio;

PROCEDURE A();
BEGIN
 WriteString('In procedure A');
 B();
END;

PROCEDURE C();
BEGIN
 WriteString('In procedure C');
 WriteString('Now raises exception');
 RAISE Exception;
END;

END.
```

/* CTEST2.C */

```
extern void __pascal _WriteString(char * const str);
extern void __pascal _C(void);

void  _B(void)
{
 _WriteString("In external function B");
 _C();
}
```

Test output:

```
In procedure A
In external function B
In procedure C
Now raises exception
Exception caught and handled
```

Test for object cleanup

An extened version of the example *cleanup_example* in figure 7 is used to test for local, global and sub-objects. The example has been extended with a main exception handler and output statements.

Test output:

```
Raise point ? 1
Ctr for A called
Exception raised in ctr for A
Exception handled

Raise point ? 2
Ctr for A called
Ctr for B called
Exception raised in ctr for B
Dtr for A called
Exception handled

Raise point ? 3
Ctr for A called
Ctr for B called
Ctr for C called
Exception raised in ctr for C
Dtr for B called
Dtr for A called
Exception handled

Raise point ? 4
Ctr for A called
Ctr for B called
Ctr for C called
In statements for procedure P
Exception raised in procedure P
Dtr for C called
Dtr for B called
Dtr for A called

Exception handled
Raise point ? 5
Ctr for A called
Ctr for B called
Ctr for C called
In statements for procedure P
Dtr for C called
Exception raised in dtr for C
Dtr for B called
Dtr for A called
Exception handled

Raise point ? 6
Ctr for A called
Ctr for B called
Ctr for C called
In statements for procedure P
Dtr for C called
Dtr for B called
Exception raised in dtr for B
Dtr for A called
Exception handled

Raise point ? 7
Ctr for A called
Ctr for B called
Ctr for C called
In statements for procedure P
Dtr for C called
Dtr for B called
```

```
Dtr for A called
Exception raised in dtr for A
Exception handled
```

Test for cleanup of array of objects

The program *cleanup1* tests for cleanup of arrays of destructible objects, both partial and non-partial cleanup.

```
PROGRAM cleanup1;
USES stdio;
TYPE
  A = OBJECT
       nr: INTEGER;
       CONSTRUCTOR init(); FORWARD;
       DESTRUCTOR done(); FORWARD;
      END;
VAR
 raise_nr : INTEGER;
 counter : INTEGER;
CONSTRUCTOR A.init();
BEGIN
 nr:=counter;
 WriteString('Ctr for A called for array element ');
 WriteInt(nr);
 IF raise_nr=nr THEN
  WriteString('Exception raised in ctr');
  RAISE Exception;
 ENDIF;
 counter:=counter+1;
END;
DESTRUCTOR A.done();
BEGIN
 WriteString('Dtr for A called for array element ');
 WriteInt(nr);
 IF raise_nr=-nr THEN
  WriteString('Exception raised in dtr');
  RAISE Exception;
 ENDIF;
END;

PROCEDURE P();
VAR
 a: ARRAY [1..4] OF A;
BEGIN
 WriteString('In statements for procedure P');
 IF raise_nr=0 THEN
  WriteString('Exception raised');
  RAISE Exception;
 ENDIF;
END;
BEGIN
 WriteString('Raise element nr (+ for ctr / - for dtr) ? ');
 raise_nr:=ReadInt();
 counter:=1;
 TRY
  P();
 EXCEPT
  | Exception => WriteString('Exception handled');
 END;
END.
```

Test output:

```
Raise element nr (+ for ctr / - for dtr) ? 1
Ctr for A called for array element 1
```

```
Exception raised in ctr
Exception handled

Raise element nr (+ for ctr / - for dtr) ? 2
Ctr for A called for array element 1
Ctr for A called for array element 2
Exception raised in ctr
Dtr for A called for array element 1
Exception handled

Raise element nr (+ for ctr / - for dtr) ? 3
Ctr for A called for array element 1
Ctr for A called for array element 2
Ctr for A called for array element 3
Exception raised in ctr
Dtr for A called for array element 2
Dtr for A called for array element 1
Exception handled

Raise element nr (+ for ctr / - for dtr) ? 4
Ctr for A called for array element 1
Ctr for A called for array element 2
Ctr for A called for array element 3
Ctr for A called for array element 4
Exception raised in ctr
Dtr for A called for array element 3
Dtr for A called for array element 2
Dtr for A called for array element 1
Exception handled

Raise element nr (+ for ctr / - for dtr) ? 0
Ctr for A called for array element 1
Ctr for A called for array element 2
Ctr for A called for array element 3
Ctr for A called for array element 4
In statements for procedure P
Exception raised
Dtr for A called for array element 4
Dtr for A called for array element 3
Dtr for A called for array element 2
Dtr for A called for array element 1
Exception handled

Raise element nr (+ for ctr / - for dtr) ? -4
Ctr for A called for array element 1
Ctr for A called for array element 2
Ctr for A called for array element 3
Ctr for A called for array element 4
In statements for procedure P
Dtr for A called for array element 4
Exception raised in dtr
Dtr for A called for array element 3
Dtr for A called for array element 2
Dtr for A called for array element 1
Exception handled

Raise element nr (+ for ctr / - for dtr) ? -3
Ctr for A called for array element 1
Ctr for A called for array element 2
Ctr for A called for array element 3
Ctr for A called for array element 4
In statements for procedure P
Dtr for A called for array element 4
Dtr for A called for array element 3
Exception raised in dtr
Dtr for A called for array element 2
Dtr for A called for array element 1
Exception handled

Raise element nr (+ for ctr / - for dtr) ? -2
```

```
Ctr for A called for array element 1
Ctr for A called for array element 2
Ctr for A called for array element 3
Ctr for A called for array element 4
In statements for procedure P
Dtr for A called for array element 4
Dtr for A called for array element 3
Dtr for A called for array element 2
Exception raised in dtr
Dtr for A called for array element 1
Exception handled

Raise element nr (+ for ctr / - for dtr) ? -1
Ctr for A called for array element 1
Ctr for A called for array element 2
Ctr for A called for array element 3
Ctr for A called for array element 4
In statements for procedure P
Dtr for A called for array element 4
Dtr for A called for array element 3
Dtr for A called for array element 2
Dtr for A called for array element 1
Exception raised in dtr
Exception handled
```

Test for cleanup of heap objects

The program *cleanup1* tests cleanup of heap memory if a constructor for a heap object is exited by an exception.

```
PROGRAM heap_cleanup;
USES stdio;
TYPE
 A = OBJECT
     END;
 B = OBJECT
      CONSTRUCTOR init(); FORWARD;
     END;
 C = OBJECT(B)
     END;
CONSTRUCTOR B.init();
BEGIN
 WriteString('Exception raised in constructor for sub-object B
              for object C');
 RAISE Exception;
END;
VAR
 pa: ^A;
 pc: ^C;
BEGIN
 WriteString('Free heap memory before make of new object A : ');
 WriteInt(GETFREE());
 NEW(pa);
 WriteString('Free heap memory before attempt to make new object C : ');
 WriteInt(GETFREE());
 TRY
  NEW(pc);
 EXCEPT
  | Exception => WriteString('Exception handled');
 END;
 WriteString('Free heap memory after failed attempt to make
              new object C : ');
 WriteInt(GETFREE());
 DISPOSE(pa);
 WriteString('Free heap memory after object A was destroyed : ');
 WriteInt(GETFREE());
END.
```

Test output:

```
Free heap memory before make of new object A : 10000
Free heap memory before attempt to make new object C : 9999
Exception raised in constructor for sub-object B for object C
Exception handled
Free heap memory after failed attempt to make new object C : 9999
Free heap memory after object A was destroyed : 10000
```

Appendix I References

Exception Handling related:

[Koenig90] Andrew Koenig and Bjarne Stroustrup, “Exception Handling in C++”, JOOP July/August 1990

[Cameron92] Don Cameron, Paul Faust, Dmitry Lenkov & Michey Mehta, “A Portable Implementation of C++ Exception Handling”, C++ Technical Conference 1992

[Philbrow90] P.C. Philbrow and M.P. Atkinson, “Events and Exception Handling in PS-algol” revised, The Computer Journal, Vol 33-2, 1990

[Lajoie94] Josée Lajoie, “Exception Handling - Supporting the runtime mechanism, Supporting first-class objects”, C++ Report, 1994

[Boling94] Eli Boling & Peter Kukol, “Underneath Structured Exceptions & C++ Exceptions”, Borland Internation Inc, 1994

+ *various personal correspondences from Internet*, including :

David Chase, Thinking Machines Corporation.
Phil Lucido, Microsoft.
John Ellis.

Compiler related:

[Fischer91] Charles N. Fisher & Richard J. LeBlanc, Jr, 1991, “Crafting a compiler with C”

[Aho86] Alfred V. Aho, Ravi Sethi & Jeffrey D. Ullman, 1986, “Compilers - Principles, Techniques, and Tools”

[Fraser91] Christopher W. Fraser & David R. Hansen, 1991, “A Code Generation Interface for ANSI C”

Pascal & Assembler language:

Borland Turbo Assembler v4.0 documentation, 1993.

Borland Pascal with Objects v7.0 documentation, 1992.

General nature:

[Stroustrup91] Bjarne Stroustrup, 1991, “The C++ Programming Language” 2ed

[Harbison92] Samuel P. Harbison, 1992, “Modula-3”

[Rumbaugh91] James Rumbaugh, 1991, “Object-Oriented Modeling and Design”

[Tanenbaum90] Andrew S. Tanenbaum, 1990, “Structured Computer Organization”