Design and implementation of a declarative programming language in a reactive environment

Angelo Mottola

Advisor
Prof. Camil Demetrescu

Co-advisor
Prof. Irene Finocchi

Opponent
Prof. Daniele Nardi

A.A. 2004-2005
# Contents

Abstract x

Introduction xi

Subject of the thesis ........................................ xi
Previous related work ...................................... xii
Original contributions ..................................... xiii
Organization of the thesis ................................. xiv

I Design 1

1 The declarative paradigm 2
  1.1 Introduction ........................................... 2
  1.2 Common programming paradigms ...................... 3
    1.2.1 Imperative programming .......................... 3
    1.2.2 Object-oriented programming .................... 4
    1.2.3 Functional programming ......................... 5
    1.2.4 Logic programming ............................... 6
    1.2.5 Other programming paradigms .................... 7
  1.3 Declarative programming ............................. 8
    1.3.1 The declarative model ............................ 8
    1.3.2 Declarative languages ............................ 9
    1.3.3 Advantages and limitations ..................... 11
  1.4 Multiparadigm programming .......................... 12
  1.5 Conclusions .......................................... 14

2 Reactive callbacks 15
  2.1 Introduction .......................................... 15
  2.2 Reactive programming environments .................. 15
    2.2.1 Dataflow graphs ................................ 16
    2.2.2 FRAN ............................................. 18
2.2.3 Garnet and Amulet ............................................... 19
2.2.4 Signal and Lustre .................................................. 19
2.2.5 The hardware approach ......................................... 20
2.2.6 Learned lessons .................................................... 21
2.3 A new approach ...................................................... 22
  2.3.1 Objective ........................................................... 22
  2.3.2 Reactive callbacks ............................................... 23
  2.3.3 Cyclical dependencies .......................................... 26
  2.3.4 Discussion ......................................................... 28
  2.3.5 An application example ........................................ 29
2.4 Conclusions .......................................................... 33

3 The Alpha+ declarative programming language .................. 34
  3.1 Introduction .......................................................... 34
  3.2 What is Alpha+ ....................................................... 35
    3.2.1 A first example ................................................ 35
    3.2.2 Alpha+ as a multiparadigm language ....................... 36
  3.3 Fundamental concepts ............................................. 38
    3.3.1 Predicates ..................................................... 38
    3.3.2 Objects and classes ......................................... 39
    3.3.3 Memory model ............................................... 39
    3.3.4 Execution model .............................................. 40
  3.4 Basic Alpha+ constructs ......................................... 41
    3.4.1 Predicates ..................................................... 41
    3.4.2 Shortcuts .................................................... 46
    3.4.3 Classes ........................................................ 49
    3.4.4 Similarities between Alpha+ predicates and C functions 53
  3.5 Advanced Alpha+ constructs ..................................... 54
    3.5.1 Class setup and cleanup .................................... 55
    3.5.2 The foreach statement ...................................... 56
    3.5.3 Generic virtual predicates ................................ 58
    3.5.4 Extern classes ................................................. 59
    3.5.5 Polymorphism in Alpha+ .................................... 61
    3.5.6 Execution model revised .................................... 63
    3.5.7 A programming example .................................... 66
  3.6 Dynamic Alpha+ .................................................... 71
    3.6.1 Alpha+ and reactive callbacks ............................... 71
    3.6.2 The example of an animation ............................... 72
  3.7 Discussion .......................................................... 74
    3.7.1 Scope .......................................................... 75
    3.7.2 Advantages ................................................... 76
<table>
<thead>
<tr>
<th>Contents</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.7.3 Limitations</td>
<td>77</td>
</tr>
<tr>
<td>3.8 Conclusions</td>
<td>77</td>
</tr>
</tbody>
</table>

## II Implementation

### 4 The compiler

4.1 Introduction 79
4.2 Compiler architecture 81
  4.2.1 Modules and components 81
  4.2.2 Target machine code 83
  4.2.3 Alpha+ integration 84
4.3 The symbol table 85
  4.3.1 Overview 85
  4.3.2 Extending symbol table for Alpha+ support 89
4.4 The Alpha+ parser 95
  4.4.1 About scopes 96
  4.4.2 The parse process 97
  4.4.3 Parsing classes 98
  4.4.4 Parsing predicates 101
  4.4.5 succeed, fail and foreach 106
  4.4.6 Predicate calls 115
4.5 The Alpha+ linker 118
  4.5.1 The link process 118
  4.5.2 Class records 120
  4.5.3 Extending the linker 122
  4.5.4 Alpha+ glue code 123
4.6 Conclusions 125

### 5 The virtual machine

5.1 Introduction 126
5.2 Virtual machine architecture 127
  5.2.1 The virtual CPU 127
  5.2.2 The traps system 129
  5.2.3 Kernel and managers 130
  5.2.4 Extending the VM for Alpha+ support 133
5.3 The Alpha+ manager 134
  5.3.1 Overview 134
  5.3.2 Object instance context and live contexts tree 136
  5.3.3 The CreateAllObjects trap 141
5.4 The RC manager 143
List of Figures

2.1 Dataflow graph example. (a) program output; (b) user-defined constraints; (c) resulting graph; (d) reaction chain for when $GRect(1).h$ is modified .......................................................... 17
2.2 Reorganized dataflow graph for reactive callback $f()$ of listing 2.1 .......................................................... 24
2.3 More constraints in the same RC. (a) the RC function; (b) corresponding dataflow graph ........................................... 25
2.4 Cyclical dataflow graph for RC in listing 2.2 .......................... 26
2.5 Execution of program in listing 2.3. (a) Dataflow graph and RCs queue at end of initialization loop; (b-f) Reactive call-backs execution steps .......................... 31
3.1 Output of listing 3.1 .................................................. 36
3.2 Output for program in listing 3.25 .................................. 62
3.3 Live contexts tree for listing 3.26 ................................... 64
3.4 Output of the Alpha+ generic example of listing 3.29 ......... 70
3.5 An output frame of the animation for program in listing 3.31 . 74
4.1 Alpha+ workflow ................................................... 80
4.2 Compiler architecture layout ....................................... 82
4.3 Compiler architecture layout with Alpha+ support ............. 85
4.4 Data structures layout for example in listing 4.4 ................ 88
4.5 Predicate data structure representation ........................... 92
4.6 Symbols, scopes and type nodes for class in listing 4.9 ...... 95
4.7 The global Alpha+ scope ........................................... 96
4.8 Function call handling. (a) C source program; (b) portion of corresponding unbound object code; (c) linked object code; (d) machine state before jsr; (e) machine state after jsr .......................... 108
4.9 foreach statement example. (a) sample predicate; (b) corresponding unbound object code; (c) bound object code .............. 110
4.10 Stack layout before and after a bjsr instruction is executed . 112
List of Figures

4.11 Standard predicate call example. (a) sample predicate; (b) corresponding unbound object code; (c) bound object code; (d) stack layout on fail after jsr; (e) stack layout on succeed after jsr; (f) stack layout before clns ............................. 115

4.12 Class record layout. (a) standard class record; (b) global class record ............................................................... 120

5.1 VM architecture layout ......................................................... 127
5.2 VM architecture layout with Alpha+ support ................................. 133
5.3 Object instance context layout .............................................. 137
5.4 Live contexts tree and its connections for the Alpha+ program in listing 5.3 .......................................................... 139
5.5 Monitoring nodes data structures layout .................................. 146
5.6 Monitoring nodes removal .................................................... 149
5.7 Reactive callbacks triggering ................................................. 150
## List of Tables

4.1 Virtual CPU registers ............................................. 84
4.2 Class declaration parsing state machine ......................... 99
5.1 CPU traps .............................................................. 130
Listings

1.1 Ada implementation of the factorial function (imperative) . . 10
1.2 Prolog implementation of the factorial function (declarative) . 11
2.1 A sample C function . . . . . . . . . . . . . . . . . . . . . . . 23
2.2 A reactive callback containing a cyclical dependency . . . . . 26
2.3 C program computing Fibonacci series using reactive callbacks 30
3.1 A simple Alpha+ example . . . . . . . . . . . . . . . . . . . . 35
3.2 Factorial function using multiple paradigms in Alpha+ . . . . 37
3.3 Generic Alpha+ predicate example . . . . . . . . . . . . . . . 42
3.4 A constructor example . . . . . . . . . . . . . . . . . . . . . . 42
3.5 Example of a selector . . . . . . . . . . . . . . . . . . . . . . 43
3.6 The same virtual predicate behaving differently for two objects 45
3.7 Virtual predicates operating on the same object and scope . . 45
3.8 Compacted version of previous example . . . . . . . . . . . . . 45
3.9 Implicit succeed . . . . . . . . . . . . . . . . . . . . . . . . . . 47
3.10 Embedding an out parameter assignment . . . . . . . . . . . . 47
3.11 Implicit parameter type . . . . . . . . . . . . . . . . . . . . . 48
3.12 Implicit out parameter name . . . . . . . . . . . . . . . . . . 48
3.13 Implicit predicate body . . . . . . . . . . . . . . . . . . . . . 48
3.14 Equivalent forms of the same constructor . . . . . . . . . . . . 49
3.15 An example of class with just the interface portion . . . . . . 50
3.16 Tiny class sub objects example . . . . . . . . . . . . . . . . . 51
3.17 A more detailed class example . . . . . . . . . . . . . . . . . . 52
3.18 Class setup function . . . . . . . . . . . . . . . . . . . . . . . 55
3.19 Foreach example . . . . . . . . . . . . . . . . . . . . . . . . . . 57
3.20 line virtual predicate expanded . . . . . . . . . . . . . . . . 58
3.21 A standard virtual predicate . . . . . . . . . . . . . . . . . . . 58
3.22 Generic virtual predicate . . . . . . . . . . . . . . . . . . . . . 59
3.23 Obtaining object id via a generic virtual predicate . . . . . . 59
3.24 Extern class example . . . . . . . . . . . . . . . . . . . . . . . 60
3.25 An example of polymorphism . . . . . . . . . . . . . . . . . . 62
3.26 A simple Alpha+ program with multiple sub-object declarations 65

viii
3.27 The GCanvas Alpha+ class ........................................ 66
3.28 The GCanvas.h class header file ............................... 69
3.29 An Alpha+ example program using the GCanvas class .... 69
3.30 An example of dynamic constructor ............................ 71
3.31 Alpha+ animation example ....................................... 73
4.1 Structure of the generic scope node entry ...................... 86
4.2 Structure of the generic symbol node entry ..................... 86
4.3 Structure of the generic type node entry ....................... 87
4.4 A small C function .................................................. 88
4.5 Function type node also used by Alpha+ predicates ........... 90
4.6 mPredicate structure in the symbol node entry ............... 91
4.7 Field type node also used by Alpha+ classes .................. 93
4.8 mClass structure in the symbol node entry .................... 94
4.9 A simple class implementing a GUI alert dialog .............. 94
4.10 Small class highlighting the need for a two-pass parse process . 99
4.11 Multiple predicate cases .......................................... 103
4.12 Merged predicate cases in pseudo code ....................... 104
4.13 Predicate ID expression check code ............................ 105
4.14 Nested foreach ...................................................... 114
5.1 The virtual CPU environment structure .......................... 128
5.2 The bjsr trap handler function .................................... 135
5.3 A simple Alpha+ class hierarchy .................................. 138
5.4 The RC record structure .......................................... 144
5.5 The RC watch node structure ..................................... 147
5.6 Conditional monitoring in an RC ................................. 148
Abstract

Developing modern software applications is a complex task that requires integrating different programming styles and heterogeneous software technologies. While a procedural style can be very effective to specify the algorithmic part of a program, several other aspects, including parallelism and graphic user interface, are descriptive in nature, making declarative approaches a natural choice. Integrating different programming paradigms in a unified framework is therefore desirable, although it remains a difficult and challenging problem. In this thesis we investigate how to combine imperative, object-oriented and declarative programming styles in a novel approach using a dataflow environment as a common ground. The dataflow software architecture embodies the basic principle that changing the value of a variable should automatically force the recalculation of the values of other variables. The reactive nature of dataflow environments makes them a natural ground for supporting event-based object-oriented programming, while taking advantage of the descriptive nature of declarative languages to specify constraints between objects. Our study culminated in the design and implementation of a new programming language called Alpha+, which is a multiparadigm extension to C based on a dataflow execution model. The novel features of Alpha+ allow for a number of interesting applications; in our work we focus on interactive visualization and graphic user interface management, areas in which Alpha+ has proven to be an easy to use and effective alternative to existing approaches.
Introduction

Subject of the thesis

In this thesis we investigate how to integrate imperative, object-oriented and declarative features in a dataflow environment. Our study culminated in the design and implementation of a novel programming language called Alpha+.

The dataflow software architecture embodies the basic principle that changing the value of a variable should automatically force the recalculation of the values of other variables, that were previously marked as dependent upon the first one; this recalculation is often referred to as reaction in response of an event that changed the original variable. Perhaps the most widespread application of the dataflow architecture is embodied by spreadsheets; in a spreadsheet the user can specify a cell formula that depends on other cells; then when any of those cells is updated the value of the first cell is automatically recalculated. It is possible to establish multiple dependencies in which a change to a single cell would trigger a chain reaction, if the cell depends on another cell which depends on yet another cell, and so on. The same principle can be applied to applications development and execution, where the role of a cell is replaced by a program variable. The result is that the execution flow of applications running on top of a dataflow environment is data-driven rather than control-driven. Environments providing support for the dataflow architecture are often also referred to as reactive environments.

The declarative programming paradigm assumes programs to be descriptive in nature. Most classical programming paradigms put emphasis on how a problem should be solved. For example, the imperative, object-oriented and functional models all let the programmer specify the steps required in order to solve a particular task. A completely different approach is followed by the declarative paradigm; in this model, the programmer focuses on providing the what, leaving the how up to interpretation, which usually depends upon the context of the language.

Our programming language Alpha+ was designed to combine aspects of the declarative, imperative and object-oriented paradigms in a novel multi-
paradigm fashion. At the outer level, the language uses a pure declarative approach, allowing programs to be composed of *declarations of objects* and their properties, and *declarations of classes* of objects, possibly containing other objects to form a tree structure. Objects retain a state, and class methods are implemented using the same syntax and semantics of C, with some relevant extensions. Alpha+ classes support classical object-oriented features such as data encapsulation, information hiding and modularity. Function polymorphism is also supported and plays a major role in our language; differently from typical OO approaches though, Alpha+ applies late binding based upon the context in which an object appears, rather than its position within the class hierarchy. Objects get created and freed transparently to the user, hiding all the memory allocations and deallocations to the programmer.

Alpha+ programs are executed on top of a dedicated reactive environment, allowing it to establish reaction chains between objects states and their methods. The creation and destruction of objects themselves can be bound to external conditions, making the objects tree dynamic. In addition, we let Alpha+ code to be embedded inside C programs, allowing the states of Alpha+ objects to be dependent upon C variables. As a consequence of the use of a reactive environment, declarative Alpha+ programs provide dynamic features that allow for a number of interesting applications. In our work we focus on interactive data visualization in graphical scenes and user interface management, areas in which Alpha+ has proven to be an easy to use and viable alternative to existing approaches.

**Previous related work**

The dataflow architecture concept is not yet a fully explored research area nowadays. The most common approaches to the problem of constraints satisfaction – ensuring that variable dependencies are enforced causing the recalculation of other variables – usually involve the use of complex and expensive algorithms to build a topological graph of the dependencies and scan it. Existing efforts like FRAN, Garnet and Amulet [8, 19, 24, 34] use this method to provide a reactive environment and they ship in the form of toolkits, or extension libraries, for existing languages. The consequence of this choice is that they do not have complete control over the programming environment, and the constraints satisfaction algorithm must be explicitly executed via specific constructs in order to trigger reactions. Examples of standalone languages providing a reactive environment are instead provided by Signal [13] and Lustre [5, 11]; compared to other approaches, these introduce advanced techniques to reduce the constraints satisfaction algorithm execution
time, with the result of providing reactive systems that are suitable for real-
time tasks, where the software must react as quickly as possible to external
changes. Signal and Lustre are in fact currently employed in robotic ap-
pliances, critical control software in aircraft, helicopters, and nuclear power
plants. Probably the most promising approach is however provided by the
concept of dedicated dataflow hardware in the form of dataflow processors
[29]; an hardware approach eliminates the need to explicitly execute a con-
straints satisfaction algorithm as dependencies are automatically enforced by
the hardware at the lowest possible level.

With our project we aim to bridge the gap between existing software and
hardware dataflow environments, thru the use of a reactive virtual machine
having complete control over the environment; this machine will enforce de-
pendencies in a transparent way for the Alpha+ applications running on top
of it.

Original contributions

The original contributions of this thesis include both design and implement-
ation aspects. Design contributions include:

- Definition of an abstract reactive machine using an innovative approach
  based on the concept of reactive callbacks. A reactive callback is ba-
sically a function that performs some arbitrary computations. The
execution of this function causes all the variables it references to be-
come monitored by the reactive callback; a subsequent change of the
contents of these variables will automatically trigger the re-execution
of the function. In dataflow terms, a reactive callback constitutes both
the constraint and the data this constraint works with, and is used to
establish and satisfy dependencies between variables. The reactive ma-
chine implementing support for reactive callbacks provides the dataflow
environment on which Alpha+ programs are executed.

- Definition of the Alpha+ language rationale, based on a novel com-
bination of declarative, object-oriented and imperative programming
paradigms.

- Definition of the syntax of Alpha+ as an extension of C.

Implementation contributions include:

- Development of a compiler frontend module for Alpha+, using the Le-
onardo C Compiler as backend. The module is used to parse Alpha+
programs and to generate unbound object code.
Development of a linker module for Alpha+ object code. The linker outputs executable units to be later interpreted by a dedicated virtual machine, the Leonardo VM.

Development of a module for the Leonardo VM to support Alpha+ specific features at execution time.

Development of a module to support a reactive environment inside the Leonardo VM, thru the implementation of reactive callbacks.

We used the Leonardo Computing Environment (LCE) project as a basis for our work. This project aims to bring a full-featured ANSI-C compiler and dedicated virtual machine to provide a complete and self-contained development environment. Its features and availability as an open-source project made it a perfect candidate to become a canvas onto which to implement our ideas. More details on the project can be found online at its website [6].

Organization of the thesis

Following the contributions division in two distinct parts, the thesis is organized in a design part, composed of the first three chapters, and an implementation part composed of the last two chapters.

Chapter 1 introduces common programming paradigms, with focus on the declarative paradigm. Declarative languages are discussed presenting their advantages and limitations, and finally the multiparadigm approach is examined as a possible solution to merge and get the advantages of other paradigms.

Chapter 2 presents the dataflow software architecture, and discusses existing approaches to it. It then introduces a new method based on the concept of reactive callbacks, discussing the challenges the new approach had to face.

Chapter 3 introduces the Alpha+ programming language, its concepts and syntax. It then shows how the language can be used in real-world applications for static graphical scenes management and, thanks to the integration with the reactive environment presented in the previous chapter, for interactive scenes and animations.

Chapter 4 enters the implementation discussion, presenting how the Alpha+ compiler was developed; parsing, generating code and linking are all aspects examined in detail, with emphasis on the problems encountered in order to provide the unique features of the Alpha+ language, in the context of a reactive environment.
Introduction

Chapter 5 discusses how the VM was enhanced to support both Alpha+ specific features and, more importantly, the reactive callbacks system providing dataflow characteristics to the environment onto which Alpha+ programs are executed.

Ending the paper, in appendix A we report the Alpha+ language productions.
Part I

Design
Chapter 1

The declarative paradigm

1.1 Introduction

We begin our venture by considering this puzzling word, paradigm. If you consult a dictionary with the objective of better understanding the meaning of this term, you are likely to find a definition similar to the one reported by The American Heritage Dictionary of the English Language:

\textbf{par-a-digm NOUN:} 1. One that serves as a pattern or model.
2. A set or list of all the inflectional forms of a word or of one of its grammatical categories: the paradigm of an irregular verb.
3. A set of assumptions, concepts, values, and practices that constitutes a way of viewing reality for the community that shares them, especially in an intellectual discipline.

At first look this seems to have little to spare with programming languages. To understand the connection, we must note the term was brought into modern vocabulary through the influential book \textit{The Structure of Scientific Revolutions}, authored by science historian Thomas Kuhn [18]. Kuhn was the first to use the word to describe a set of theories, standards and methods that together represent a way of organizing knowledge, that is, of viewing the world, thus effectively setting the third term definition above. Kuhn thesis was that revolutions in science occurred when an older paradigm was examined, rejected, and then replaced by a new one.

Applying Kuhn’s vision of the term to programming languages, a paradigm represents a way to solve problems using a specific approach to them, determined by the unique world view the paradigm provides. Different categories of problems may be examined using the same approach, and this could lead to a valid solution in most cases, but the time required to formulate a valid
solution algorithm for the problem may depend upon the paradigm chosen with which the problem is being approached. By looking at the problem from a different point of view, provided by a different paradigm, the same task could be solved much more easily, both in terms of time and complexity of the algorithm required to solve it.

1.2 Common programming paradigms

Several programming paradigms have been invented to ease the solution of certain problem categories. In this section we will examine the basic characteristics of the most common ones.

1.2.1 Imperative programming

Many view the imperative paradigm as the “traditional” model of computation. The imperative model closely matches the actual execution of machine code on the underlying hardware. Computation is viewed as a task to be performed by a processing unit that manipulates and modifies memory. Memory can be envisioned as a sequence of cells, each holding a value, identified by an unique number or named symbol in the case of higher-level imperative languages. A program is meant to fetch values from memory, perform some kind of operations on them and store their modified version again in memory. When the program ends, the values held in memory represent the solution of the algorithm performed by the program execution.

Imperative programs can be envisioned as state machines; the state is given by the contents of the program memory cells. During execution the state continually changes as memory is modified, until it contains the final result. Imperative languages provide means to explicitly and easily control the execution flow, like loops, conditional branching statements and subprograms, also known as procedures, via unconditional branching statements.

The imperative model, being so close to the execution model of the underlying hardware, can express any computational task. The problem with this model however does not lie in its expressivity, but in the fact it is far from the way the human brain goes about solving problems. Thus formulating a problem in the imperative model can be a complex and tedious task, requiring a considerable effort by the programmer, that grows exponentially as the complexity of the problem increases. This fact led to the search and development of alternative programming paradigms that could better resemble our own way of thought when approaching different problems, effectively aiding programmers formulating complex tasks.
1.2.2 Object-oriented programming

The idea behind object-oriented programming (OOP) is that a computer program is composed of a collection of individual units, or objects, as opposed to the traditional imperative model view in which a program is a list of instructions to the computer. Briefly, each object is capable of receiving messages, processing data, and sending messages to other objects. Alan Kay, considered by many the father of object-oriented programming, developed the language Smalltalk, one of the first to popularize the object-oriented view; in its manual [12], he highlights the following fundamental concepts:

1. Everything is an object.

2. Computations are performed by objects communicating with each other via messages and their arguments, requesting that other objects perform specific actions.

3. Each object has its own memory, which consists of other objects. This memory constitutes the internal state of the object.

4. Every object is an instance of a class. A class simply represents a grouping of similar objects.

5. A class defines the interface with which its objects communicate. That is, a class is a repository for the behavior associated with its objects; all object instances of a class can perform the same actions.

Subsequent versions of Smalltalk (Smalltalk-76 and Smalltalk-80) also introduced an additional important concept:

6. Objects are organized into a singly rooted tree structure, called the inheritance hierarchy. Memory and behavior associated with an instance of a class are automatically available to any class associated with a descendant in the tree structure.

The nature of the OOP model forces the programmers to divide programs in coherent class modules that encapsulate the representation of one data type per module; classes describe the state (data values) and behaviors (actions) for objects. This is in contrast with the unordered collections of functions that call each other the imperative model provides.

For this and other reasons, object-oriented programming is claimed to give more flexibility, easing changes to programs, and is widely popular in large scale software engineering. Furthermore, proponents of OOP claim that
OOP is easier to learn for those new to computer programming than previous approaches, and that the OOP approach is often simpler to develop and to maintain, lending itself to more direct analysis, coding, and understanding of complex situations and procedures than other programming models.

1.2.3 Functional programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions; a program is itself a function definition, that calls other functions as its arguments. To quote Simon Thompson in the introduction to his book *Haskell: The Craft of Functional Programming* [32]:

“A functional programming language gives a simple model of programming: one value – the result – is computed on the basis of others – the inputs.”

Mathematical functions have great strengths in terms of flexibility and analysis. For example, if a function is known to be idempotent, then a call to a function which has its own output as its argument, and which is known to have no side-effects, may be efficiently computed without multiple calls.

Functional programming are mostly in contrast with imperative programming. Functional programming appears to be missing several constructs often (though incorrectly) considered essential to an imperative language such as C or Pascal. For example, in strict functional programming, there is no explicit memory allocation and no explicit variable assignment. However, these operations occur automatically when a function is invoked; memory allocation occurs to make space for the parameters and the return value, and assignment occurs to copy the parameters into this newly allocated space and to copy the return value back into the calling function. Both operations can only occur on function entry and exit, so side effects of function evaluation are eliminated. By disallowing side effects in functions, functional programming languages become stateless and provide referential transparency, meaning that the result of a function will be the same for a given set of parameters no matter where, or when, it is evaluated. Referential transparency greatly enhances modularity, as a function does not depend on the context where it is being executed, and eases both the task of proving program correctness and the task of automatically identifying independent computations for parallel execution.

Looping, another imperative programming construct, is accomplished through the more general functional construct of recursion. Recursive func-
tions invoke themselves, allowing an operation to be performed over and over, effectively replacing common imperative looping constructs.

Thanks to the brevity, expressiveness and availability of sophisticated data structures and algorithms, modern functional programming languages are now used in a wide range of scientific applications, from numerical analysis to visualization.

1.2.4 Logic programming

The point of logic programming is to bring the style of mathematical logic to computer programming. The classical propositional proof of a theorem consists of three parts:

1. a set of axioms, which are facts that are given to the problem and are assumed to be true.

2. a set of rules of inference, or rules that can be used to derive new information by combining known facts together.

3. a question or query to the problem.

The proof of the theorem consists of a path that produces the result of the question by applying the rules of inference to the given information. The process of constructing a proof is well-known, so logic is thought to be a reliable way to answer questions.

Logical programming languages automate this process, by letting programmers specify the three points above and deriving results accordingly. The language executor (a compiler or an interpreter) applies a solution algorithm which scans every possible combination of inference rules applied to the given information; the process may come to a situation where no more rules can be applied to the facts derived up to that point, and the result does not satisfy the original query. In this case, the algorithm has to apply backtracking: incorrect derived facts are discarded and the algorithm restarts back at the last point where all the known facts are assumed to be correct, but execution takes a different route by applying another rule, all until a valid solution is found or all inference rules are applied to every possible derived facts combination.

What is important to note in logical programming is the fact the programming model is declarative or non-procedural. Programmers simply provide axioms, rules of inference and a query, and the language executor takes care about applying a particular search algorithm to solve the problem. The fact
1. The declarative paradigm

the paradigm has a declarative nature implies that like functional languages, logical languages are atemporal.

Though they require high computational power to perform execution, the automatic problem solving capabilities of logical programming languages makes them particularly suitable for artificial intelligence applications. Actually, the growth of interest in artificial intelligence had an important influence on the development of the logical programming paradigm itself.

1.2.5 Other programming paradigms

There exist several different other less known programming paradigms, developed to better suit certain ranges of problems. A detailed list would be far too long and out of scope for this paper, so we briefly report some of them for completeness:

- **Constraint programming.** Similar to logical programming, this paradigm is more declarative than imperative in nature. The difference between the two is largely in their styles and approaches to modeling the world. In the constraint model the programmer provides a series of constraints that are guaranteed to be satisfied during execution by an underlying mechanism similar to the one used by logical programming executors.

- **Aspect-oriented programming.** This paradigm helps the developer separating concerns, or breaking down a program into distinct parts that overlap in functionality as little as possible. Similar in focus to object-oriented programming, it differs from this by using particular mechanisms to defy crosscutting concerns, or concerns that cannot be easily encapsulated using OOP languages.

- **Visual programming.** Particular kind of paradigm which assumes programs to be visually designed rather than written in text. This radical new approach has been made possible only in recent times with the increase of computer computational power and the introduction of flexible graphical user interfaces.

- **Pipeline programming.** A programming paradigm in which the programmer connects notional program modules into a flow structure, by analogy to a physical pipeline carrying reaction products through a chemical or other plant. The best-known example of this paradigm is the UNIX pipe system and its interaction with the shell scripts language.
1.3 Declarative programming

We will now leave common programming paradigms behind to focus on a particular kind of paradigm whose discussion we have omitted thus far: the declarative programming paradigm. We will then examine how declarative programming languages look like, their strong points and their weaknesses.

1.3.1 The declarative model

Three out of the four programming paradigms we previously examined share a strong common point: they put emphasis on how a problem should be solved. The imperative, object-oriented and functional models in fact all let the programmer specify the steps required in order to solve a particular task. A completely different approach is followed by the declarative programming paradigm; in this model, the programmer focuses on providing the what, leaving the how up to interpretation, which usually depends upon the context of the language.

The most direct consequence of this basic property of the declarative paradigm is that in a declarative programming language a program is a theory written in some suitable logical and structural form; this characteristic leads a reader of the program to immediately get a precise meaning for it. From the point of view of a programmer, it translates to programming lifted to a higher level of abstraction. At this higher level of abstraction the programmer can concentrate on stating what is to be computed, not necessarily how it is to be computed. In Kowalski’s terms [17] where:

\[ \text{algorithm} = \text{logic} + \text{control} \]

the programmer gives the logic but not necessarily the control.

This is similar to the concept behind logical programming, which in fact is said to be declarative in nature. The difference is the logical paradigm assumes rules of inference and a query for a program to work, whereas the pure declarative paradigm does not pose these limitations. It is often reported however that declarative programming includes logical programming as a sub-paradigm. To a certain degree, also functional programming is often considered as the second sub-paradigm of the declarative model, due to the fact they share some aspects like absence of state and referential transparency, as well as due to the intrinsic nature of functional programming which is based upon function declarations.

According to J. W. Lloyd [21, 22], declarative programming can be understood in a weak and in a strong sense. Declarative programming in the strong sense means that the programmer only has to supply the logical elements of
1. The declarative paradigm

an algorithm and that all control information is supplied automatically by the underlying system. We also refer to this case as pure declarative model. Declarative programming in the weak sense means that the programmer apart from the logic of a program also gives some control information to yield a more efficient program.

The key features of the declarative programming paradigm are:

- **Expression terms.** These are the basic blocks of a declarative program. An expression declares a particular aspect of the problem in logical terms.

- **Referential transparency.** We already encountered this feature in other paradigms. Applying it to the declarative model, this means that an expression denotes a value irrespective of the context.

- **Absence of state.** The declarative nature of the paradigm itself defies the notion of state; programs are made up of expression declarations so the concept of state is foreign in this view.

- **Implicit operational semantics.** The term-rewriting, unification and resolution is left to the program executor, which can be the language compiler or interpreter. Using Lloyd’s naming convention, this feature only applies to the strong declarative paradigm.

The increasing popularity of declarative programming languages stems from the inherent simplicity of the paradigm, whose high level of abstraction guarantees human readable programming styles.

1.3.2 Declarative languages

From the discussion above, it should be clear the declarative paradigm partially covers both logical and functional programming; the distinctions are never sharp. Maybe the most famous programming language with a declarative nature and that is widely used is Prolog [7], even though it is better known as a logical programming language. To better understand what the declarative paradigm means to the programmer’s point of view, we will now examine a very simple example, showing how the solution to the same problem can be implemented in both classical imperative manner as well as with a declarative approach. The declarative language of choice is Prolog, whereas we will use Ada [10] as the imperative one.

Consider the problem of the factorial mathematical function. Put in logical terms we have:
1. The declarative paradigm

\[ n! = \begin{cases} 
1 & \text{if } n = 0 \\
 n \cdot (n - 1)! & \text{else}
\end{cases} \]

To a programmer who has to solve a factorial problem using an imperative programming language like Ada, the most logical solution is to write a small recursive procedure implementing this algorithm. This will likely look as in the following listing:

```ada
with CS_IO; use CS_IO;

procedure FACTORIAL is
    N: integer;
    T: integer := 1;
    begin
        put("Input "); get(N);
        for K in 2..N loop
            T := T*K;
        end loop;
        put("Factorial "); put(N);
        put("is "); put(T); newline;
    end FACTORIAL;
```

Listing 1.1: Ada implementation of the factorial function (imperative)

The above comprises a series of steps which, if executed, will calculate the factorial of a given number. As shown in this example, when writing in an imperative language the programmer needs to concentrate on both the logic (what) and the control (how). As previously stated, the principal idea behind declarative languages is to separate out these two elements so that the programmer is free to concentrate on the logic of a problem without being distracted by control considerations, which are automatically handled by the underlying system. Keeping this in mind, listing 1.2 shows how the same factorial problem can be solved by a small Prolog program, using a declarative approach. Here we define the logic (the desired goal or result) and not the control (how we achieve the desired goal). Note how much the declarative implementation is close to the initial problem definition, and how both just specify what is to be achieved, or in other words, the logic of the problem.

The declarative programming languages and the declarative paradigm in general are getting more and more widespread attention in the last decade, also thanks to the increasing need for easy to use programming languages,
1. The declarative paradigm

factorial(0,F) .

factorial(N,F) :-
    N1 is N-1,
    factorial(N1,F1),
    F is N*F1.

Listing 1.2: Prolog implementation of the factorial function (declarative)

almost usable also by non-programmers. Examples of declarative languages that address this need are SQL and XML, which are common nowadays. SQL is a well-established language, and is the de-facto standard for databases management and querying. XML is expanding its area of influence, although often more for marketing reasons than for real practical needs, and is being used in various areas related to data management. Apart from database-related uses, other common uses of declarative programming languages are in problems related to artificial intelligence, constraint-satisfaction problems, configuration management and inter-process communication.

1.3.3 Advantages and limitations

The main property of the declarative programming paradigm is that the programmer focuses on what a problem requires to be solved, rather than how to solve it. The first immediately visible major advantage of this approach is that resulting programs are compact and easier to understand when compared to analogous imperative solutions. Listings 1.1 and 1.2 clearly show the declarative model requires less steps to achieve the same result.

The same example also shows another advantage of declarative programming: being programs so close to the initial logical problem definition, they allow for an high abstraction level, preventing the programmer from delving into irrelevant details. This makes the solution easier for humans to understand compared to most of the other paradigms which are often more machine-oriented. Another consequence is that complex programs are easier to write and manage in a declarative environment, as the underlying engine takes care of memory management and logical inferencing. Declarative languages are therefore particularly suited for rapid prototyping of data structures and of code to express complex ideas.

The explicit separation of the what and how of programs brings yet another benefit, as while the programmer concentrates on the what, the language executor handles the steps needed to solve the problem. This opens to
the executor the possibility for any kind of optimizations for specific kinds of problem solving.

Isolating the solver algorithm inside the language executor, which as previously stated can be a compiler or – as in most practical cases – an interpreter, effectively avoids to the programmer the need for reinventing the wheel each time a program needs to be implemented. A declarative language executor usually always works for a range of problems, and once properly written it can apply its logic to a wide variety of declarative specifications.

The disadvantages of the declarative programming paradigm are chiefly about efficiency. The interpretation phase must necessarily involve a very general approach to problem solving in a declarative environment in the range of problem the language is designed for. The isolation between actual implementation and problem abstraction is an important technique in order to get a broader picture of a computer system. But, sometimes it is important to understand the implementation rather than enjoy the isolation, especially if it is necessary to optimize an algorithm.

Aside to efficiency considerations, another important aspect to consider is expressiveness. Again due to the implicit separation of the what and how brought by the declarative paradigm, expressivity can be severely reduced, being limited to a strict range of problems. This limits the possible language applications and methodologies to solve problems, when compared to other programming paradigms.

### 1.4 Multiparadigm programming

We have seen how each programming paradigm is especially well suited for a range of applications. With the notable exclusion of the imperative model, which being so close to the underlying machine execution model can express any kind of problem, other paradigms are often more or less limited in their expressiveness. This is the price to pay to have a programming model that guarantees an approach more similar to the way the human brain works; the more a paradigm heads towards being more easy to master and away from the machine, the more chances will be for it to be limited to a strict range of action.

This huge limitation can be overcome by allowing multiple paradigms to coexist into the same programming language. By allowing the programmer to freely choose which paradigm to use depending upon the problem to be solved, these multiparadigm programming languages effectively give the advantages of each paradigm while strongly reducing their limited application range. The design goal of such languages is to allow programmers to use the
1. The declarative paradigm

best tool for a job, admitting that no one paradigm solves all problems in
the easiest or most efficient way.

Multiparadigm languages incorporating several different paradigms into
the same framework are still new and not widely used nowadays, and their
potential is not yet fully disclosed. We will briefly analyze two of them: Leda
and Oz.

- **Leda.** As the principal designer of this language Timothy Budd ex-
plains in his book *Multiparadigm Programming in Leda* [3], goal of the
language is to successfully mix imperative, object-oriented, functional
and logical programming paradigms. The idea of Leda is to provide an
unique framework in which programmers can work in a variety of styles,
freely intermixing constructs from different paradigms. Leda represents
a good example of seamless integration of the four major programming
paradigms we examined in section 1.2, providing benefits that would
not be easy to emulate using any single paradigm by itself.

- **Oz.** The most aggressive example of multiparadigm programming lan-
guage is Oz, which contains in a simple and well-factored way most
of the concepts of the major programming paradigms, including logic,
functional, imperative and object-oriented, but also constraint, dis-
tributed, and concurrent programming. Oz has both a simple formal
semantics and an efficient implementation, the Mozart Programming
System. Oz is a concurrency-oriented language, as the term was intro-
duced by Joe Armstrong, the main designer of the Erlang language.
A concurrency-oriented language makes concurrency both easy to use
and efficient. In addition to multi-paradigm programming, the major
strengths of Oz are in constraint programming and distributed pro-
gramming. Because of its factored design, Oz is able to successfully im-
plement a network-transparent distributed programming model. This
model makes it easy to program open, fault-tolerant applications within
the language. For constraint programming, Oz introduces the idea of
computation spaces, which allows user-defined search and distribution
strategies that are orthogonal to the constraint domain.

While providing the advantages of multiple paradigms reducing the limit-
ations of single ones, multiparadigm programming languages can suffer of the
inherited complexity brought by the unification of different models. Mixing
several paradigms when designing a programming language can prove to be
a daunting task, with the risk for example of producing a syntax which can
become obscure sometimes, with the effect of reducing the ease of use and
freedom of choice a multiparadigm approach should bring.
1. The declarative paradigm

1.5 Conclusions

In this chapter we have briefly examined the major computer programming paradigms, namely the imperative, object-oriented, functional and logical paradigms, ending with a focused overview of the declarative programming model. We have found this particular paradigm to be extremely well suited for rapid prototyping of both data structures and code to express complex ideas, which would require a much greater effort if developed using a different more classical imperative approach. Declarative languages are limited though, as we know different paradigms provide different views of the same problem to be solved, each with its advantages and limitations. To take advantage of the inherited benefits of different programming paradigms using the same language, multiparadigm programming languages have also been introduced.

In light of our knowledge of the different programming models, in chapter 3 we will introduce the Alpha+ declarative programming language, which is the subject of this thesis. Before that however, we will first introduce reactive environments and dataflow architectures, which are concepts required by the design of our own language.
Chapter 2

Reactive callbacks

2.1 Introduction

The concept of reactive programming is based on a particular kind of software architecture, that in the literature [1, 14, 28, 30] is better known as *dataflow architecture*. The basic idea behind the dataflow principle is that changing the value of a variable should automatically force the recalculation of the values of other variables, that were previously marked as dependent upon the first one; this recalculation is often referred to as *reaction* in response of an *event* that changed the original variable. Perhaps the most widespread application of the dataflow architecture is embodied by spreadsheets; in a spreadsheet the user can specify a cell formula which depends on other cells; then when any of those cells is updated the value of the first cell is automatically recalculated. It is possible for one change to initiate a whole sequence of changes as a chain reaction, if one cell depends on another cell which depends on yet another cell, and so on.

The same principle can be applied to programming languages, and we refer to these languages as *reactive programming languages*. In this chapter we will discuss existing techniques used in the implementation of reactive programming language, and we will introduce our own unique approach to the problem. In the next chapter will dedicate this approach to provide a reactive environment, with all its advantages, for the Alpha+ declarative language, subject of this thesis.

2.2 Reactive programming environments

There have been a few programming languages created specifically to support the dataflow architecture. In particular, most of them are visual program-
2. Reactive callbacks

Modern programming languages [23, 31] that allow to visually put links between the various entities, effectively establishing a directed graph representing the dependencies between existing objects. Other approaches involve the use of dedicated programming languages that allow to specify constraints between variables, forcing the automatic recomputation of these variables to satisfy the associated constraints. We say that a language or toolkit that enforces the use of a dataflow architecture gives to the programs a reactive environment in which they are executed.

2.2.1 Dataflow graphs

In dataflow programming languages, a one-way dataflow constraint can be formally written as an equation in the form:

\[ v = F(p_0, p_1, p_2, ..., p_n) \]

where each \( p_i \) is a parameter of the function \( F \), which is called a formula. If the value of any \( p_i \) changes during the program execution, the whole function \( F \) is automatically recomputed, causing the value of \( v \) to be updated accordingly. If \( v \) is changed as well during the program execution, the constraint is left temporarily unsatisfied, hence the constraint is called a one-way constraint.

The program executor of a dataflow programming language has to solve the dependencies that the various constraints create with each other; the solver algorithm typically uses a bipartite, directed graph that is called dataflow graph, or simply dependency graph, as it keeps track of dependencies among variables and constraints. There is a directed edge from a variable to a constraint if the formula of the constraint uses that variable as a parameter. There is a directed edge from a constraint to a variable if the constraint assigns a value to that variable. Formally, the graph can be represented as \( G = \{V, C, E\} \), where \( V \) is the set of variables, \( C \) the set of constraints and \( E \) the edges connecting variables and constraints. When a variable is modified, the one-way path in the dataflow graph that connects that variable to \( n \) other variables which depend on it thru constraints forms a chain of nodes in the form:

\[ R = \{v_1, c_1, v_2, c_2, ..., v_{n-1}, c_{n-1}, v_n\} \]

whose nodes are the variables \( v_i \in V \) interleaved by the constraints \( c_i \in C \) that use \( v_i \) as a parameter; we call this chain a reaction chain. Figure 2.1 shows an example of dataflow graph for three constraints that force the position and height of a rectangle depending on those of a second rectangle;
in the same figure it is also shown an example of reaction chain for when the height of the second rectangle is modified.

The dataflow graph is often directly managed by the programmer in visual programming languages, that use it to allow to visually construct constraints between entities. Visual dataflow environments are inherently limited though; there are in fact many cases where more powerful programming constructs are needed to deal with complex problems. There exist other languages that do not use the visual paradigm but rather apply other more usual programming models, and still provide a dataflow environment, usually thru dedicated extensions; these languages relegate the management of the dataflow graph to their internal runtime. Before analyzing how we designed our own new reactive environment, we will now examine some of these existing solutions to understand how they approach the problem of bringing up to date the constraints by evaluating the formulas.
2. Reactive callbacks

2.2.2 FRAN

FRAN (Functional Reactive Animation [8]) is the first solution that we will discuss; it is a graphics toolbox composed of a collection of data types and functions for composing richly interactive, multimedia animations. It provides means to distinguish between what is an animation and how it should behave, thanks to the use of a declarative approach. FRAN describes an animation thru temporal modeling: graphical objects are functions of time and are described to work in a certain temporal range.

A reactive environment is provided by FRAN by introducing the concepts of behaviors and events. A behavior is a reactive variable whose value changes trigger an event; an event is a function to be executed depending upon behaviors. Graphical objects use behaviors to automatically change their properties, achieving an animation that is function of both behaviors and time. The major problem of such an approach is to determine whenever an event is to be triggered, without sacrificing performance too much in costly algorithm scanning the whole set of declared behaviors for a given event. FRAN is in fact meant for graphical scenes animations, and to get a decent number of frames per second guaranteeing a smooth animation, a valid technique has to be used, especially when dealing with complex scenes where multiple objects are animated at once.

The developers of FRAN solved the problem by providing to the toolkit an implicit treatment of time: all events are specified as boolean function of time during which they are expected to occur. The nature of an event can then be exploited to eliminate search over intervals of time in which the event probably does not occur, and focus instead on time intervals in which the event may occur. The detection of such intervals is thus crucial to the system, and is performed using a technique called interval analysis, implemented using a divide and conquer approach applied to the temporal ranges associated to each event.

As we will find in the next chapter, FRAN is akin to Alpha+ in the fact they share an important design goal: they are both designed to provide an easy way to visualize data and animations in a reactive context. Their approach to the problem is extremely different though; FRAN is implemented as a toolkit, in the form of a library of functions and data types available for the Haskell programming language [25, 32]; as such, it is rather an high-level approach to the dataflow problem, with an unavoidable granularity of the intervals of time during which reaction occurs. This granularity is indeed a consequence of the interval analysis method involved, which applies the divide and conquer approach to an interval until the interval being divided becomes smaller than a certain degree of temporal accuracy. Alpha+ on the
other hand has been developed as a real programming language running on top of a dedicated reactive environment, whose approach we will study in section 2.3.

2. Reactive callbacks

2.2.3 Garnet and Amulet

Similar to FRAN, Garnet and Amulet \[19, 24, 34\] are graphical toolkits providing and taking advantage of a reactive environment to display graphical scenes and keep them updated automatically as the user interacts with them. Garnet is a Lisp-based toolkit first released in 1989, and used in over 80 projects; Amulet is a C++ successor to Garnet, released in 1994. Both have introduced a number of innovations in dataflow constraints, and have incorporated innovations from other constraint systems as well.

As with other approaches to dataflow programming, Garnet and Amulet also provide means to bind behavioral objects to property/value pairs; a change in a property is used to automatically control the behavior of an object, be it its appearance, position, and state. Properties are the reactive hooks in Garnet and Amulet, and are called slots. Constraints can be applied to properties, and this is often used to specify graphical behaviors, for example to ensure that a text label is always centered inside a rectangle.

To deal with the constraint satisfaction problem, various approaches have been tested by the developers. The algorithm that was finally chosen is the mark-sweep technique: this consists of two distinct phases. The mark phase starts at a set of changed variables, performs a depth-first search of the dataflow graph, and marks as out of date any constraint it visits. The sweep phase then evaluates out of date constraints whose values are requested by the application.

2.2.4 Signal and Lustre

Signal \[13\] and Lustre \[5, 11\] offer examples of standalone and flexible languages dedicated to programming reactive systems. These languages are directly based on the model of reactive system as being activated by input events and producing corresponding output events. In some way similar to FRAN, their approach is to divide the life of a reactive system into instants, which are the moments during which the system reacts; they allow the programmers to specify statements that depend upon instants. For example, a program may wait for the third instant where a given event occurs, and so on. The dataflow technique used approaches the problem using clocks as powerful control structures to manipulate data, where clocks are a form of temporal types. The clock of a flow defines the sequence of logical instants where it
bears a value. Clocks are used by the constraint satisfaction algorithm to determine whether a data needs to be recomputed; this is achieved by sampling constraints at discrete intervals of time, and ensuring at each step that they are satisfied.

Both Signal and Lustre belong to the category of synchronous programming languages: a synchronous programming language is a programming language optimized for programming a certain range of reactive systems: those that are assumed to be often interrupted and that must respond quickly. Many such systems are also called real-time systems, and are found often in embedded uses; they are of growing importance in the last decades. In the case of Lustre, the language began as a research project in the early 1980s. In 1993, it progressed to practical, industrial use, in a commercial product, as the core language of the industrial environment SCADE, developed by Esterel-Technologies [9]. It is now used for critical control software in aircraft, helicopters, and nuclear power plants.

2.2.5 The hardware approach

In all the approaches examined thus far, an important assumption has been made: all of the reactive environment solutions are built on top of common, standard hardware architectures and try to introduce software means to provide reactive support. The dataflow architecture itself, as it was conceived and as we introduced it in section 2.1, is a software architecture. Constraint satisfaction, which is the most important part of the reactive model, is always implemented as a set of procedures that at certain intervals of time ensure constraints are satisfied by re-evaluating formulas. Usual methods assume that a program attaches some procedures to a reactive variable, so that those procedures get automatically called whenever the variable is changed to keep constraints satisfied; these procedures are what keep the display updated in graphical toolkits like FRAN, Garnet and Amulet.

The nature of the dataflow architecture can be further exploited if the constraints satisfaction problem is parallelized, that is, if multiple computing units can re-evaluate formulas to keep constraints updated. A radically different approach using this parallel working model is currently being researched, and involves the use of dedicated hardware; these new hardware dataflow-oriented architectures are called dataflow processors [29]. The classical control-driven CPU architectural pipelining hazards limit the performance; these hazards are due to data dependencies and to structural reasons due to control flow. These can be avoided if succeeding instructions in the pipeline stem from different contexts, as in the case of multithreaded cores in dataflow processors. Rather than control-driven machines, dataflow pro-
cessors are in fact data-driven: an instruction on the CPU is enabled and then executed only when all of its operands are available.

It is clear this radical new approach to the dataflow architecture gives many advantages over other software based ones, including much faster execution model, removal of dependency checks inherent to common software constraints satisfaction algorithms, and consequent removal of cyclical dependency troubles to name a few. The downside is that the whole concept is young and still being researched, other than the fact such an approach requires new dedicated hardware that is likely to be expensive.

2. Reactive callbacks

2.2.6 Learned lessons

The above discussion of existing dataflow programming solutions shed light on the most common approaches to the constraints satisfaction problem. Specifically, we have found that:

1. Most solutions are implemented as extension libraries for existing programming languages. The constraints satisfaction algorithm is usually executed to perform housekeeping when needed.

2. A common technique to reduce the constraints satisfaction algorithm workload is to associate a time interval to constraints, specifying the period of time during which the constraint is likely to apply. The algorithm then simply avoids analyzing those constraints whose temporal interval does not overlap with current time.

3. The best approach to the problem is to use dedicated hardware. This however has downsides that limit its usefulness at present time.

Considering the fact the hardware approach is not viable for our implementation, to provide a reactive environment to the Alpha+ programming language we have to either use existing software approaches as discussed thus far, either introduce innovative concepts to better face the problem in our specific case. We chose the second path, bearing in mind the challenges a new approach to reactive programming can bring:

- **Arbitrary code.** A constraint should be able to provide an arbitrary piece of code that is legal in the underlying programming language; in particular, a constraint should be able to contain loops, conditionals, function calls and possibly recursion.
2. Reactive callbacks

- **Dereferencing.** A constraint should be able to reference variables indirectly via pointers, and the constraints satisfaction algorithm should transparently cope with the occurrence with no need for the programmer’s intervention.

- **Automatic constraint parameter detection.** A constraint should automatically detect its parameters \( p_i \) as it executes, so that the programmer should not explicitly declare the constraint’s parameters.

- **Cyclical dependencies.** Cyclical loops may require special treatment: specifically, cyclical dependency loop to satisfy a constraint or a series of constraints should be detected if a fixed point can be reached, and the cycle should be interrupted to ensure the convergence.

In the next section we will discuss how these problems are all solved by our own new unique approach.

### 2.3 A new approach

Having examined what existing reactive programming languages are, we can now focus on providing our own new approach to the reactive environments architecture. We will start defining our design goals, to later introduce the unique solutions we have envisioned for the project.

#### 2.3.1 Objective

Our task is to design a new reactive environment, into which we will later allow applications coded using the Alpha+ declarative programming language to be executed. Other than solving the four challenges highlighted in the previous section, our approach aims to reach an additional important goal:

- **Automatic triggering.** We want the code associated to the formula defining a constraint to be automatically re-executed when one of its parameters \( p_i \) is modified, either in response of a direct user intervention, either in response of another constraint evaluation.

This point alone begs to differ from traditional software-based approaches we have examined in section 2.2; traditional approaches assume there is a language or toolkit functionality that is explicitly dedicated to execute the constraints satisfaction algorithm. In our approach instead, we want this algorithm to be executed in the background, transparently to the programmer.
who has not to call explicit language procedures or use specific syntaxes to change a constraint parameter, but rather directly use the same syntax of the language as used elsewhere, as if accessing normal variables. As a direct consequence, it is reasonable to aim for the independence of the reactive environment from the programming language being used by the programmer.

To succeed in such an approach, we have to provide some kind of control at a lower level than that of the programming language using the reactive features. This control has necessarily to be applied at the memory level: we need a way to monitor individual memory cells, so that when referencing a variable that is a parameter of a constraint, the system identifies it and acts accordingly, automatically triggering the associated constraint code. This technique is somewhat akin the hardware approach examined in section 2.2.5, but it is implemented in software; we will discuss the actual implementation in chapter 5, for now we will assume the underlying software architecture has means to monitor individual cells of memory\(^1\), detecting when they are accessed in both read and write mode.

Proceeding in the discussion of our approach, we find that allowing to monitor memory alone does not provide a reactive environment yet. First we have to introduce a mean to both set a constraint \(F(p_1, p_2, \ldots, p_n)\) and the code to be executed when the variables that constitute the parameters \(p_i\) of \(F\) are changed. We will now introduce reactive callbacks, which are the solution to both these problems.

\[ \begin{align*}
\text{float} & \: x, \: y, \: z; \\
\text{void} & \: f() \{ \\
& \quad x = \cos(y) + \sin(z); \\
& \}
\end{align*} \]

\textbf{Listing 2.1: A sample C function}

---

\(^1\)Modern operating systems do not allow userland applications to actually monitor the application memory for our particular purpose; to circumvent the problem, we will use a combination of a compiler and a virtual machine. Our applications will be interpreted by a VM that will allow full process memory monitoring.
Suppose we want to make the variables reactive; that is, we want the function to be re-evaluated whenever $y$ and $z$ change, since those are the variables that are accessed for reading. A re-evaluation of $f()$ would update the contents of variable $x$ whenever $y$ and $z$ are changed. In our model, we can achieve this effect by marking function $f()$ as a reactive callback (RC for short). When a normal function is marked as a reactive callback, once it is executed all the variables that it accesses for reading are monitored; a subsequent write operation performed on monitored variables from any point in the program will automatically trigger the re-execution of the associated reactive callback function. If more than one reactive callback is monitoring the same variable, and such a variable is written to, all the monitoring reactive callbacks are scheduled for re-execution.

In typical dataflow terms, a reactive callback corresponds to the constraint $F$, and the monitored variables to its own parameters $p_i$; the dataflow graph for the reactive callback $f()$ is shown in figure 2.2.

The figure shows a reorganized dataflow graph, where we have put the reactive callback representing $F$ on the left, and all the involved memory accesses representing $p_i$ and $v$ on the right. This layout helps detecting the dependencies of the data from the reactive callback: an arrow directed from an RC to a variable represents a memory write operation the RC performs on the variable; an arrow directed from a variable to an RC represents a memory read operation performed on the former by the latter. Note that the same reactive callback can correspond to more than one constraint: it is sufficient that the function marked as RC performs more than one assignment, like in figure 2.3. When $F_1, \ldots, F_m$ constraints are applied in the same RC, a modification applied to any variable associated with parameter $p_i$ will automatically trigger the re-execution of the whole RC function. The execution of an RC does not happen immediately at the time a monitored variable is modified: the RC function is instead actually scheduled for execution into a dedicated queue. Each RC function scheduled in the queue is executed in no particular order until there are no more RCs, and when this happens control is given back to the user program.
2. Reactive callbacks

The concept of reactive callbacks eliminates the need for special functionalities dedicated to explicitly monitor variables. Complex constraints satisfaction algorithms that detect which variables have been modified and trigger execution of constraints code are also not needed, as the system automatically knows the list of functions to be triggered when a memory cell associated to a variable is modified; we in fact assumed that the application memory can be monitored, and that changes to a memory cell can be detected and can trigger functions execution.

The only interface such an approach needs to provide to the programmer is composed of four functions:

- **KRC_New**: installs a new reactive callback on the function passed as parameter, causing its immediate execution. This first execution marks referenced memory as monitored by the RC, so it can be automatically triggered again once the monitored memory is modified.
- **KRC_Delete**: removes a previously installed reactive callback. Memory cells monitoring the RC are marked as not monitoring it anymore.
- **KRC_Suspend**: causes the temporary suspension of an RC automatic triggering. Suspended reactive callbacks are not triggered anymore even if monitored memory is modified.
- **KRC_Resume**: resumes automatic triggering of an RC which was previously suspended by KRC_Suspend.

The **KRC_New** function is the most important one: particularly, it is important to note that this function causes the immediate execution of the

```c
float x, y, z;
int a, b;

void f() {
    x = cos(y) + sin(z);
    a = b + 10;
}
```

![Figure 2.3: More constraints in the same RC. (a) the RC function; (b) corresponding dataflow graph](image)
reactive callback function being installed; this step is of great importance, as the first execution marks the memory cells referenced by the variables accessed for reading in the callback as monitored by the RC itself. Without this step, the system would have no prior knowledge of what memory cells the RC monitors, and would not be able to perform automatic triggering.

The reactive system introduced via reactive callbacks works and is powerful enough to manage most situations, with one exception: we still need a way to detect when cyclical dependencies can be interrupted to avoid cascade triggering and consequent infinite loops.

### 2.3.3 Cyclical dependencies

We have found that when a reactive callback function is executed, the memory it uses for reading is marked as monitored by the RC itself. When this memory is written to by the program, the RC function will be scheduled for re-execution. In our design however, we have not forbidden to a reactive callback to trigger the execution of another reactive callback if during the execution of the first a memory cell monitored by the second is modified. Allowing a reactive callback to trigger another reactive callback is a required feature for a good reactive system, as it allows for reaction chains longer than one node. This however poses the problem of cyclical dependencies, where the reaction chain forms a loop.

A simple case of cyclical dependency for example occurs when a reactive callback both reads and writes the same variable, as in listing 2.2.

```c
int x;

void f() {
    x = x + 1;
}
```

**Listing 2.2:** A reactive callback containing a cyclical dependency

![Cyclical dataflow graph for RC in listing 2.2](image)
The associated dataflow graph with the above listing is shown in figure 2.4. Let us examine how this generates an infinite cycle: when function \( f() \) is installed as an RC via the \texttt{KRC\_New} function, \( f() \) is executed a first time to start monitoring variables. It finds the variable \( x \) is accessed for reading, so it links its associated memory cell with the current RC. Then it finds \( x \) is written to, and since \( x \) is currently monitored, the associated RC – that is the current RC – is scheduled for re-execution. When the first RC execution terminates, the reactive callbacks queue now contains the same RC, which has then to be re-executed before relinquishing control to the user program. This however repeats the cycle, with the result that the queue is never empty, and the user program is never resumed.

Such infinite cyclical dependencies are undoubtfully sign of bad programming in a reactive system. Other non infinite cycles could be allowed to the programmer however, as this can guarantee more expressive power; a loop may indeed be wanted by the programmer, so the reactive system should not try to avoid it, but rather ensure it converges to a fixed point. This is the approach we are going to take: we will allow cycles, but we will add a rule that will at least try to make the cycle break if a certain condition is met. To detect cycles break conditions non-trivial and costly algorithms cannot be used: in our case we need to act as quickly as possible, due to the fact this check is to be done at triggering time, and thus each time a monitored memory cell is modified. We cannot afford to perform costly algorithms each time a memory cell is modified, otherwise the performance of the whole system would be severely affected.

As our solution, we decided to apply this rule:

\begin{itemize}
  \item \textbf{Rule:} a write operation on variable \( v \) triggers the re-execution of a reactive callback function \( f \) if and only if:
  \begin{enumerate}
    \item \( \exists N = (v, f, e_{v,f}) \setminus N \in G \)
    \item \( [v]_{\text{new}} \neq [v]_{\text{old}} \)
  \end{enumerate}
  where \( [v] \) specifies the contents of variable \( v \).
\end{itemize}

Basically the rule allows a reactive callback to schedule the execution of another reactive callback only if, when writing to a monitored variable, the value of the variable already in memory differs from the value we are going to write into it.

This rule makes the most trivial cases of cyclical dependencies to converge, but is obviously not enough to make all cases to converge; a more correct solution would require to analyze the topological layout of the dataflow graph and identify cycles, but this method would be expensive in terms
of performance and as we said we need to be as fast as possible, so it is not affordable. Our compromise rule works best especially for small reaction chains of the form \( \{v_1c_1v_2\} \), where the check for equality ensures infinite cycles as in figure 2.4 are avoided if \( v_1 \) and \( v_2 \) are equal. If they differ however, as in the example of listing 2.2, applying the rule does not avoid the infinite loop.

Good use of cyclical dependencies in the reactive environment is left to the good sense of the programmer: as with the case of recursion in imperative programming languages, which are a powerful expressive tool but can cause infinite recursion if not used properly, we decided to leave to the programmer the task of avoiding infinite cyclical dependencies in our reactive environment thru correct programming. This is because allowing convergent cycles can give to the programmer more freedom and open a whole lot of additional possibilities for the reactive environment applications.

### 2.3.4 Discussion

Our new approach to the dataflow architecture solves all the problems as highlighted in section 2.2.6, as we will now examine:

- **Arbitrary code**: the concept of reactive callback ensures a complete function to be executed to define constraints. Being a normal function programmed in the underlying language, this can contain any type of statement, from conditionals to function calls and recursion. The assignments it contains become the constraints to be applied.

- ** Dereferencing**: the reactive environment works on memory cells rather than variables; this ensures the correct addresses to always be monitored, effectively bypassing any dereference problem.

- **Automatic constraint parameter detection**: our model works on a memory basis, where each monitored memory cell has a list of reactive callbacks to be triggered upon modification. If a variable whose address is watched is modified, the system will find the associated memory cell has at least a reactive callbacks attached to it, and will thus assume it to be a parameter to one of the constraints contained in it, causing its automatic scheduling for re-execution.

- **Cyclical dependencies**: by applying the rule discussed in section 2.3.3, our approach tries to make cyclical dependencies to converge to a stable solution, avoiding infinite loops if possible. The check for convergence is fast, ensuring efficiency; non convergent cycles are allowed, as we leave to the programmer the task of ensuring the full correctness of the
2. Reactive callbacks

program. This choice gives more responsibility but also more control and expressive power to the user.

- **Automatic triggering**: as examined above, no constraints satisfaction algorithms have to be explicitly executed to call the constraints code. This is again thanks to the fact our approach works on a memory cells basis.

By approaching the constraints satisfaction problem completely from a memory point of view, our model greatly differs from other software dataflow implementations. In doing so however we have assumed to have a rather low-level control on the hardware where the system runs, and this assumption cannot be made on modern systems unless a dedicated computing environment is introduced; in part II of this thesis we will examine how this affected our implementation. Being able to have full control on a program memory allows us to avoid using complex constraints satisfaction algorithms, as we have discussed, and the consequent simplification grants both easier maintenance of the systems and good performances.

We also have a great advantage compared to other common software dataflow implementations: our design is not bound to a particular language, as it works at a lower level, much more similar to a hardware approach than to a software one. Our reactive environment only provides an interface of four functions that can be called by the language that wants to use the environment; the dataflow implementation is then completely hidden to the language, and no explicit means to ensure constraints satisfaction have to be used.

### 2.3.5 An application example

To show the capabilities of our reactive environment approach, we will now introduce an application example. As we said, one of the advantages of our technique is that it is language-agnostic; that is, the reactive environment itself has no knowledge of the programming language with which programs running on top of it are coded. Assuming the environment is available, for a programming language to use it, it is sufficient to provide the interface introduced in section 2.3.2 as an available API. For our example we will use the C programming language and we will assume the availability of the `KRC.h` header file, providing prototypes for the reactive callbacks interface functions described in the aforementioned section.

Suppose we want our C program to compute the first entries of the Fibonacci series into an array $v$. A traditional approach would either use an initialization loop either recursion to compute $v$ values; with the support of
our reactive environment we can instead use a much different method, as highlighted in listing 2.3.

```c
#include <KRC.h>
#include <stdio.h>

#define NUM_ENTRIES 5

int v[NUM_ENTRIES];
void rc(int idx);

void main() {
    int i;
    for (i = NUM_ENTRIES-1; i >= 0; i--)
        KRC_New(rc, i);

    for (i = 0; i < NUM_ENTRIES; i++)
        printf("F(%d) = %d
", i, v[i]);
}

void rc(int idx)
{
    if (idx < 2) v[idx] = 1;
    else v[idx] = v[idx-1] + v[idx-2];
}
```

Listing 2.3: C program computing Fibonacci series using reactive callbacks

In the program we limit ourselves to compute five entries of the series, but this is only an arbitrary choice to better illustrate how the program works; the method can be easily extended to compute any number of entries.

When the program starts in the `main()` function, the only operation it performs is just to install function `rc` as a new reactive callback for every element of the series to be computed; here we can also note that we pass a parameter to the RC when installing it; this is an user-defined parameter that will be passed to the function when the reactive environment will trigger its execution. After installing the RCs, the program then just prints the values held in the array, assuming the reactive callbacks have been called and already performed the whole series computation. We will now examine how this is possible. A first consideration is that we install the RCs in backward order; since as we said when an RC function is installed its code
is automatically executed a first time, using an forward order RCs installation would cause the program to initialize the $v$ values trivially; instead we will show how even with backward installation the reactive environment will automatically recompute the series thanks to automatic triggering and the reaction chains established during the RC installation process. Just when the RCs installation loop ends, we have the dataflow graph shown in figure 2.5a, where the $v$ arrays holds the values $\{1, 1, 0, 0, 0\}$.

![Diagram of dataflow graph and RCs queue at end of initialization loop]

Figure 2.5: Execution of program in listing 2.3. (a) Dataflow graph and RCs queue at end of initialization loop; (b-f) Reactive callbacks execution steps

When a reactive callback is executed it monitors the memory cells it reads, and schedules the execution of the RC functions attached to the memory cells
2. Reactive callbacks

it writes. The scheduled RCs are put in a queue in no particular order; as an example, on the right portion of figure 2.5a it is shown the queue holding the RCs to be executed at the end of the RCs installation loops. RCs are in fact scheduled during the installation process as the \texttt{rc()} function both reads and writes the \texttt{v} elements, establishing dependencies between the memory cells and scheduling RCs executions. So when the RCs installation ends, the queue holds a list of RCs to be re-executed as the corresponding monitored variables have been modified.

The topmost RC in the queue is then executed: in our sample case it is the \texttt{rc()} function called for \texttt{idx = 3}. Function \texttt{rc(3)} executes \texttt{v[3] = v[2] + v[1]}; effectively updating the contents of \texttt{v[3]} that pass from holding 0 to 1. The modification of \texttt{v[3]} causes the re-scheduling of \texttt{rc(4)}, as this monitors \texttt{v[3]}; however \texttt{rc(4)} is already in the queue, so we do not put it there twice. The end of the \texttt{rc(3)} function execution sees \texttt{v} holding \{1, 1, 0, 1, 0\} as shown in figure 2.5b.

\texttt{rc(2)} is now on top of the queue, so it gets executed, resulting in \texttt{v} holding \{1, 1, 2, 1, 0\} as shown in figure 2.5c. Note that \texttt{rc(4)} has been re-enqueued for execution as \texttt{v[4]} depends upon \texttt{v[2]} that \texttt{rc(2)} modified.

Figure 2.5d shows the situation once \texttt{rc(4)}, previously on top of the stack, has been executed. \texttt{v[4]} is not monitored by any RC, so the queue does not change, and at this point it holds just \texttt{rc(3)}. This is then executed, modifying \texttt{v[3]} from 1 to 3 as shown in figure 2.5e; \texttt{rc(4)} monitors \texttt{v[3]} so this is enqueued again.

The re-execution of \texttt{rc(4)} updates the value of \texttt{v[4]} from 3 to 5, and empties the queue since \texttt{v[4]} is not monitored, and no more RCs are enqueued. The final stable layout of array \texttt{v} is shown in figure 2.5f. Since no more RCs are scheduled for execution in the queue, control is relinquished to the program, and execution restarts in the \texttt{main()} function, next to the RCs installation loops, but with the array \texttt{v} already holding the correct series values \{1, 1, 2, 3, 5\}.

The efficiency of the algorithm can be quickly derived from the above discussion. Assuming we have an array of \(n\) Fibonacci values \(\{v_1, v_2, ..., v_n\}\) to be computed, the RCs installation process takes \(O(n)\) time to be performed, and this produces a queue of RCs to be executed. The unordered execution of these RCs causes the \(v_i\) value to be updated at most in \(O(n)\) time; this however can schedule other RCs for dependent values. As a consequence, generally in \(O(n)\) time at least a new value is computed. The complexity of the algorithm to compute \(n\) values is then \(O(n^2)\).
2. Reactive callbacks

2.4 Conclusions

Reactive environments constitute an interesting research topic, and taking full advantage of them is still a challenge nowadays. We have examined how existing models approach the problem of constraints satisfaction thru dedicated languages or toolkits, and we have then introduced our own solution, reactive callbacks, highlighting its advantages over other models. In the next chapter we will introduce the Alpha+ declarative programming language, and we will see how the reactive environment we have discussed so far can be used by the language to provide support for interactive features.
Chapter 3

The Alpha+ declarative programming language

3.1 Introduction

Aside from the implementation difficulties, designing a new programming language is not an easy task for many reasons. First of all, there exist hundreds of different languages nowadays, most of which unknown to the public, each fitting in a different small niche. There are not widely known languages like Corn [15] and Lucid [2] for example, which work quite well for the applications range they were developed for. Nonetheless, there may still be need for new languages easing the solution of particular tasks.

Focus of the Alpha+ declarative programming language, subject of this thesis, is to provide a general purpose programming language, but particularly well suited for easy graphical representation prototyping, including animations and GUIs. When presenting the language we will focus on these applications, though it will appear clear that the language can be used for other tasks as well. Graphics and GUI programming has never been easy in any language, due to the complexity of rasterization and the need to cope with high levels of abstractions as found in graphics, especially when dealing with GUIs which require user interaction. Object-oriented programming languages brilliantly solve many problems, especially those related to the intercorrelations between GUI elements, but still, these are not exactly easy to use for the average programmer. We will see the choice of a declarative approach as provided by Alpha+ can address this need, by allowing the programmer to easily describe graphical scenes thru simple declarations.

In this chapter we will discuss the Alpha+ fundamental concepts and features, and we will provide usage examples to show its capabilities. We will
then introduce Alpha+ in a reactive environment provided by reactive callbacks, to support interactive features. By the end of the chapter, the reader will have a general view of the language, with knowledge of its advantages and limitations.

3.2 What is Alpha+

Alpha+ is a declarative programming language. The basic idea behind the choice of a declarative approach is that to build a graphical scene, it looks natural to declare the objects that compose the scene itself. As discussed in chapter 1, the declarative paradigm guarantees the separation of the logic from the control of a program; Alpha+ is a strong, or pure declarative programming language in this sense, as it does not provide any mean to control the program execution flow. As we will see, the execution model is fixed. This is usually enough to construct graphical scenes, but for GUIs and animations a kind of control by the program over the execution is likely to be needed; we will discuss these aspects later in the chapter though. For now we will concentrate on pure graphical scenes representations and how Alpha+ deals with them.

3.2.1 A first example

We will begin with a very simple example of what an Alpha+ program may look like. Listing 3.1 shows a program that displays a graphical scene composed of a red rectangle and a blue circle.

```plaintext
GRect(out MYRECT)
GCircle(out MYCIRCLE);
GRect(MYRECT).pos(out 75, out 50);
GRect(MYRECT).size(out 100, 120);
GRect(MYRECT).color(out RED);
GCircle(MYCIRCLE).pos(out 80, out 60);
GCircle(MYCIRCLE).radius(out 50);
GCircle(MYCIRCLE).color(out BLUE);
```

**Listing 3.1:** A simple Alpha+ example

This short example shows some important basic aspects of the language. The first observation is that the whole program is made up of *declarations,*
and there are no control structures, coherently with the declarative nature of
the language. The program just declares two objects and their properties\textsuperscript{1}. The first two lines look different from the others; this is because those lines declare the two new objects, one of class \texttt{GRect} and one of class \texttt{GCircle}. In Alpha+ these special declarations are called constructors. The next lines simply declare properties for the previously declared objects, like size, position and color. The result will be a scene in which exist a red rectangle positioned at coordinates (75, 50) having a size of (100, 120), and a blue circle at (80, 60) with a radius of 50, as shown in figure 3.1.

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{example.png}
\caption{Output of listing 3.1}
\end{figure}

Note the simplicity of the program layout, consequence of the declarative approach. Expressive power thru simple declarations is a key aspect of Alpha+; complex graphical scenes can be easily constructed by placing together many simple declarations. New classes of objects can be declared to represent new entities, and the mechanism can be also easily extended to pursue other goals different from graphical representations.

3.2.2 Alpha+ as a multiparadigm language

The concepts of classes and objects introduced in the previous example clearly hint for an object-oriented second nature for Alpha+. Actually, the language not only features OOP characteristics, but also imperative constructs, effectively making it a multiparadigm programming language. As an

\textsuperscript{1}In the example we use \texttt{MYRECT} and \texttt{MYCIRCLE} as object identifiers; we will see Alpha+ only supports numerical identifiers, so it is assumed these are numerical constants defined at preprocessing time.
object-oriented language, Alpha+ works with classes, whereas as an imperative language it allows the use of classical imperative constructs like conditional branches, loops and function calls. In section 1.3.2 we have shown how the same factorial mathematical problem can be solved using both an imperative (via Ada) and a logical/declarative (via Prolog) approach; in listing 3.2 we solve the same problem with Alpha+ using three different paradigms, also proving how the language can be used for non strictly graphical programming tasks.

```java
class Program {
    fact_logical(0, out 1);
    fact_logical(n, out k) {
        if (!fact_logical(n-1, k)) fail;
        k = k * n;
        succeed;
    }
    fact_imperative(n, out k) {
        for (k = 1; n > 1; n--) k *= n;
        succeed;
    }
    fact_functional(n, out k) {
        if (n <= 0) k = 1;
        else fact_functional(n-1, k);
        succeed;
    }
}
```

Listing 3.2: Factorial function using multiple paradigms in Alpha+

Though the example shows how the factorial can be implemented in imperative, logical and functional fashions, the object-oriented nature of Alpha+ also becomes evident here as we declare the new `Program` class. By classical OOP definition, a class acts as a container for function members and instance data, and Alpha+ classes do respect this definition. Also note how imperative constructs like the `for` loop or the `if` statements, which effectively provide a mean to control execution flow, are only present inside declarations. This is important as it preserves the declarative nature of Alpha+; at the outer scope of a program, only declarations are possible, and this is why Alpha+ is considered to be a strong declarative programming language in primis, even if it features aspects borrowed from other programming paradigms, effectively making it akin to a multiparadigm programming language.
Readers familiar with the C programming language should have probably also noticed how Alpha+ imperative constructs are identical to their C counterparts. This is another important design decision took while designing Alpha+: the language builds upon C, trying to introduce as few new constructs as possible, while enhancing the semantics to better suit the language needs. This should greatly help C programmers to quickly familiarize with Alpha+, effectively reducing the steepness of the already low language learning curve.

3.3 Fundamental concepts

Now that we have got a general overview of how Alpha+ programs look like, in this section we will examine the key concepts behind the language. Being a declarative programming language and as seen in the previous section, an Alpha+ program is composed of: (i) declarations of objects and their properties, and (ii) declarations of classes of objects, possibly containing other objects. We will start defining predicates, which are the building blocks in object declarations. We will then continue with defining objects and classes to end up examining how declarations are supposed to be organized and interact in a runtime environment.

3.3.1 Predicates

An Alpha+ predicate is basically a boolean function accepting any number of parameters. There are three types of predicates:

Constructors: functions used to declare new objects of a particular class.

Instance predicates: functions declared inside an object class that are meant to return information related to a particular instance of an object of that class.

Virtual predicates: functions declared outside an object class that are used to define a particular behavior of an object.

Predicates declare objects and their properties, and correspond to the expression terms in the declarative model we discussed in section 1.3.1. We have already briefly seen constructor predicates in example of listing 3.1; we will however examine in detail each of these kind of predicates in section 3.4.1.
3.3.2 Objects and classes

Alpha+ objects are the basic entities with which programs work. An Alpha+ program declares new objects via constructor predicates, and then declares properties for these object instances via other predicates. In classic OOP style, an object belongs to a class, and it encapsulates instance data. Classes define the behavior of instance objects, and can hold:

**Instance variables:** variables relative to each object instance of the class. These are not directly accessible from outside the class, effectively providing a mean for information hiding.

**Sub objects:** child objects living in the context of each class object instance.

**Class predicates:** internal class predicates (of any of the three kinds as seen above), which can be used as selectors to get and set instance variables, as well as to define arbitrary behaviors for objects of this class and its sub objects.

Via classes, Alpha+ brings some basic typical OOP features to the language; the above brief discussion has introduced encapsulation and information hiding, and later in this chapter we will also introduce a particular kind of polymorphism. Inheritance is currently not allowed, but may be in the future. The concept of objects and classes brings an important consequence to the language: Alpha+ objects have a *state*, and this fact conflicts with two of the basic properties of the declarative programming paradigm as seen in section 1.3.1. The most obvious conflict is that Alpha+, though declarative in nature, does not feature absence of state like the declarative model assumes. As a direct consequence, referential transparency cannot be satisfied as well, as the presence of a state in objects can make them behave differently in the same context, providing the instance state differs.

Classes and their structure will be examined in detail in section 3.4.3.

3.3.3 Memory model

Every object instance declared in Alpha+ has an associated memory *context*. A context holds: (i) state of instance variables and (ii) references to sub object contexts. Each context also has an associated *scope*, relative to the class the object is instance of. This scope can contain predicates defining the behavior of child objects declared inside the class the object is instance of.

Contexts are organized in a tree structure, where the root context is special in that it is the only context that is not associated to any class. This
root context holds references to the global objects declared in the Alpha+ program, and though not associated to any class, it is associated to the global Alpha+ program scope, where predicates defining the behavior of global objects are declared.

Memory management for the contexts is automatic and transparent to the programmer, who does not have to explicitly allocate/deallocate them; declarations via constructor predicates are the only way to create new objects.

Though the concepts of context and scope may look very similar in Alpha+, there is a key difference: the context holds data for an object instance, whereas the scope holds code for predicates declared inside the class the object is instance of. More specifically, the context is defined at object memory level, whereas the scope is defined at syntax level, as we will see in section 3.4.3. Since there is a strong correlation between the two though, from now on we will use one term or the other indifferently, depending upon the topic being discussed.

### 3.3.4 Execution model

Declarative languages like Alpha+ do not have a linear execution model. When an Alpha+ program starts, the target of the language executor is to create a tree holding the contexts of declared objects, each with its sub contexts where applicable. We will call this tree the **live context tree**. Objects declared at the global scope are created first, allocating state memory and executing the initialization code for these instances. While each object instance is created, before creating the next one at the same scope all its sub object instances are created first, recursively, causing their initialization code to be executed as well. Algorithm 1 explains in pseudo code how the live context tree is created at Alpha+ program startup; the tree creation procedure is first called passing the global scope as $S$.

#### Algorithm 1 Live context tree creation

```plaintext
for each object declared at scope $S$
  allocate memory for object state
  call object initialization code
  let $S'$ be the scope of this object;
    recurs into this procedure using $S'$
```

At program end, the tree is deconstructed, freeing previously allocated memory and calling the deinitialization code for each object instance, using the same recursive approach as seen for contexts creation.
As the simplicity of the algorithm shows, the task of language executor is extremely limited. No term-rewriting nor complex solving algorithms apply here, as in other declarative programming languages like Prolog, but this is not a limitation, but rather an advantage for Alpha+. The executor can be extremely compact and optimized, relegating inference rules inside predicates, via the control structures provided by the imperative model.

### 3.4 Basic Alpha+ constructs

Let us see in detail how to work with basic Alpha+ constructs. Alpha+ builds over the standard ANSI-C language, adding as few new keywords and syntactical rules as possible; this can be found especially when dealing with the building blocks of the language: predicates.

#### 3.4.1 Predicates

Syntactically, a predicate is very similar to a standard C function; it is defined by specifying a signature, consisting of the predicate name and an optional list of parameters, followed by the predicate body enclosed within braces. All predicates return a boolean value; the new statements succeed and fail exit the predicate computation, returning true and false, respectively. The parameters list is expressed the same way as it is in C function prototypes, with one addition: by prefixing a parameter with the new out keyword, the parameter is passed to the predicate by reference, allowing the predicate to return multiple different values to the caller. out parameters do not need to be dereferenced when accessed; the dereference is automatic and transparent for the programmer. The predicate body can contain a list of C declarations and statements, with some restrictions and some enhancements to the syntax; we will come to this later though.

Listing 3.3 shows how a generic Alpha+ predicate looks like. In this short, preliminary example, MyPredicate receives two parameters: inParam is an integer passed by value, and outParam is a float passed by reference, onto which the predicate stores the result of a computation before returning true from the predicate with succeed.

**Constructor predicates**

Constructors are predicates used to declare new object instances of a certain class. Before introducing constructors though, it is important to state that
MyPredicate(int inParam, out float outParam) {
    outParam = inParam / 2.0;
    succeed;
}

Listing 3.3: Generic Alpha+ predicate example

object instances in Alpha+ are uniquely identified by both an integer identifier number and the scope into which their constructor is declared. Now, the signature of constructor predicates has the generic form:

\[
\text{ClassName(out int id)}
\]

A constructor in this form will create a new object instance of class ClassName. The Alpha+ language executor calls the constructor predicates and expects them to place a valid value \( n \) into the out integer parameter before exiting with succeed. Once the constructor succeeds, the executor creates a new object instance of the specified class, uniquely identified by \( n \) into the constructor scope. To declare a new object instance of class GRect for example, we can write:

GRect(out int id) {
    id = 1;
    succeed;
}

Listing 3.4: A constructor example

In this example we create an object of class GRect identified by the value 1 in the current scope. Obviously, the class must have been previously declared before a constructor can be used to declare new instances for that class; we will discuss class declarations in section 3.4.3.

A constructor that exits with fail does not create an object, effectively discarding any value previously stored into the out parameter. The constructor form as seen above is the most generic one, but constructor bodies can perform any kind of computations, so selective object instances creation can be possible.

It is also important to note that though constructors create new object instances, the new object context data cannot be accessed directly inside the
constructor predicate body. This is because the object instance does not yet exist during the predicate execution; the instance is created only once the predicate issues a `succeed`, exiting the predicate.

Constructors can be placed inside classes as we will see in section 3.4.3, as well as in the global scope; in this case created object instances will belong to the root node of the live context tree and will be the first to be created as discussed in sections 3.3.3 and 3.3.4.

**Instance predicates**

One of the object-oriented aspects of Alpha+ is information hiding: the class structure encapsulates data for an object instance, making it inaccessible from outside the object itself. The main purpose of instance predicates is to provide a mean to access this data; instance predicates are declared inside the object class scope, and thus can access its instance variables and sub object. Their signature has the generic form:

```
PredicateName([parameters list])
```

`PredicateName` is the identifier that will be used when referencing the predicate; it must be unique within its class. An instance predicate can have none up to any number of parameters expressed as already explained earlier in this section. As any other predicate, instance predicates can only return a boolean value via `succeed` and `fail`; by using `out` parameters though it is easy to return the object state to the caller. We will make this clear by expanding on our `GRect` class example.

```plaintext
class GRect {
    getPosition(out int _x, out int _y) {
        _x = x;
        _y = y;
        succeed;
    }
    body:
        int x, y;
}
```

**Listing 3.5:** Example of a selector

Listing 3.5 features an example of how a selector predicate could look like; the `getPosition` instance predicate of this example, as well as any instance
3. The Alpha+ declarative programming language

predicates, is declared within a class declaration. It works like a OOP selector as its purpose is just to return the contents of the \( x \) and \( y \) instance variables for object instances of class \( \text{GRect} \). We will discuss class declarations and instance variables in detail in section 3.4.3.

While instance predicates are extremely useful in the role of selectors, they can of course also be used for any task performing operations using object instance variables.

Virtual predicates

Virtual predicates have a central role in Alpha+ as they can be used to define specialized behaviors for individual object instances. The *virtual* comes from the fact they also realize another object-oriented feature of the language, by allowing a particular kind of polymorphism; we will examine this aspect in detail in section 3.5.5. The generic form of the signature for virtual predicates is:

\[
\text{ClassName( id ).PredicateName( [parameters list] )}
\]

Like instance predicates, virtual predicates are methods of a class, but contrary to the formers, they cannot be declared inside their class scope; the programmer has to declare them outside it. As a semantical consequence of the fact they are not declared within their class, these predicates cannot directly access the object context data – instance variables and sub objects if any – like instance predicates. Each virtual predicate operates on a specific object instance of class \( \text{ClassName} \), identified by the \( \text{id} \) numerical expression; the programmer can define multiple virtual predicates with the same \( \text{PredicateName} \) identifier for the same class, but operating on different object \( \text{ids} \), effectively allowing to specify completely different behaviors of the predicate for different object instances. We already encountered virtual predicates when we first introduced Alpha+ in listing 3.1; in such an example \( \text{size} \), \( \text{pos} \) and \( \text{color} \) are virtual predicates of class \( \text{GRect} \), while \( \text{pos} \), \( \text{radius} \) and \( \text{color} \) are virtual predicates of class \( \text{GCircle} \), though expressed in compact form. Listing 3.6 shows a more explicit example using the \( \text{size} \) predicate of \( \text{GRect} \), where the same predicate returns different results for two object instances.

The virtual predicate \( \text{size} \) of \( \text{GRect} \) returns the size of a rectangle object instance; here we provide two specific \( \text{cases} \) for the predicate, one for object instance with \( \text{id} \ 1 \) and the other for object instance with \( \text{id} \ 2 \); in this example the two cases of the predicate effectively define the properties of two different object instances.
Declaring in the same scope two times the same predicate operating on the same object identifier is also possible, and has the effect of creating an unique predicate which is a merged version of previous ones; the same applies for multiple instances of the same predicate; listings 3.7 and 3.8 for example show the same predicate.

Aside from the fact virtual predicates must not be declared inside their class and thus cannot access object contexts, whereas instance predicates must be declared inside their class and are allowed to access them, there is another big difference between the two; as it will become clear in section
3.5.5, the real power of virtual predicates compared to instance predicates stems from the support for a kind of polymorphism. When calling a virtual predicate, the choice of which predicate case to be executed depends not only by the object instance onto which the call is issued, but also upon the scope of both the caller and the cases.

### 3.4.2 Shortcuts

Having introduced predicates and how these can be used to declare new object instances and their properties, the next natural step would be to introduce classes, again borrowed by the classical OOP paradigm. We will however postpone their discussion, to address a contradiction the careful reader could have noticed in some of the example listings we introduced thus far. In our very first Alpha+ example (listing 3.1) we declared some virtual predicates for instances of class \texttt{GRect} and \texttt{GCircle}. For example, we assumed the line:

\begin{verbatim}
GRect(10).size(out 100, out 25);
\end{verbatim}

To be a valid virtual predicate declaration. Later though, in example of listing 3.6, the same virtual predicate for another \texttt{GRect} instance has been written as:

\begin{verbatim}
GRect( 1 ).size( out int width, out int height ) {
    width = 20;
    height = 10;
    succeed;
}
\end{verbatim}

Both forms have been assumed to be valid, and this may have caused confusion. The compact form used \textit{language shortcuts} to achieve the same semantical meaning of the expanded, explicit form. Shortcuts have been introduced to facilitate the development of Alpha+ programs; the language allows different shortcuts for various common tasks; in this section we will examine each of them, showing how these affected our \texttt{size} virtual predicate for class \texttt{GRect}.

**Implicit succeed**

If no \texttt{succeed} statement occurs in a predicate body, the predicate is assumed to exit with \texttt{succeed}. For example, the first predicate case of listing 3.6 can be rewritten as in listing 3.9. The predicates are equivalent; the programmer
can use one version or the other indifferently. If the predicate body uses \textbf{succeed} at least once, this rule does not apply, and the predicate is assumed to exit with \textbf{fail} unless otherwise specified.

\begin{verbatim}
GRect(1).size(out int width, out int height) {
    width = 20;
    height = 10;
}
\end{verbatim}

\textbf{Listing 3.9: Implicit succeed}

\textbf{Embedded out parameter assignments}

When the value to be assigned to an \texttt{out} parameter can be computed using a single expression, it is allowed to embed the assignment directly into the predicate signature.

\begin{verbatim}
GRect(1).size(out int width = 20, out int height = 10) {
}
\end{verbatim}

\textbf{Listing 3.10: Embedding an out parameter assignment}

Listing 3.10 shows the same predicate seen above, with the \texttt{width} and \texttt{height} \texttt{out} parameters featuring an embedded assignment. This is equivalent to the predicate in listings 3.6 and 3.9.

\textbf{Implicit parameter type}

It is allowed to omit the type of a parameter in the predicate signature; if this is the case, the C \texttt{int} basic type is assumed. Our \texttt{size} predicate example can then become as shown in listing 3.11; it is important to note that only integer parameters can omit their type specifier, all other parameter types must still be explicitly specified, to avoid an implicit cast to provide an undesired effect. This requires the programmer to use this shortcut rule carefully.

\textbf{Implicit out parameter name}

When embedding an \texttt{out} parameter assignment, the parameter name becomes optional; that is, we can rewrite predicate of listing 3.11 as in listing
3. The Alpha+ declarative programming language

3.12, the forms are equivalent. This is only valid when dealing with \texttt{out} parameters that embed an assignment though.

\begin{verbatim}
GRect(1).size(out 20, out 10) {
}
\end{verbatim}

\textbf{Listing 3.12: Implicit out parameter name}

\begin{verbatim}
GRect(1).size(out 20, out 10) {
}
\end{verbatim}

\textbf{Listing 3.11: Implicit parameter type}

\textbf{Implicit predicate body}

If the programmer replaces a predicate body enclosed within braces and the braces themselves with a semicolon, is the same as if the predicate provided an empty body. Our \texttt{size} predicate can so be finally written as in listing 3.13, which is the same form as used in listing 3.1.

\begin{verbatim}
GRect(1).size(out 20, out 10);
\end{verbatim}

\textbf{Listing 3.13: Implicit predicate body}

Clearly, coupled with the implicit succeed shortcut rule, this has the power of specifying successful predicates easily.

Summarizing, all the predicates in listings 3.6, 3.9, 3.10, 3.11, 3.12 and 3.13 are syntactically equivalent: they all do the same job. The expressive power of shortcuts is especially useful when dealing with constructors, as shown in listing 3.14; the expressivity of the form

\begin{verbatim}
ClassName(out <value>);
\end{verbatim}

as reported at the bottom of such listing is so concise that it is usually all the programmer needs when declaring new objects, and it is the form that we will use from now on in our examples, unless otherwise stated.
3. The Alpha+ declarative programming language

Listing 3.14: Equivalent forms of the same constructor

```java
GWindow(out int id) {
    id = 3;
    succeed;
}
GWindow(out int id) {
    id = 3;
}
GWindow(out int id = 3) {
}
GWindow(out int id = 3);
GWindow(out id = 3);
GWindow(out 3);
```

3.4.3 Classes

The concept of classes in Alpha+ is directly borrowed from the classical OOP definition. Basically, a class acts as a container defining the data and behavior for instance objects of this class. At the data level, a class defines what will be contained in contexts attached to object instances of the class, such as instance variables and sub object references. At the program structure level, a class defines a scope into which predicates can be placed; as we will find, a class scope can contain any of the three kinds of predicates, even though with some restrictions.

The class syntax layout foresees a signature in the form of the `class` keyword followed by the name of the class, and by the class internal declarations enclosed within braces.

```java
class ClassName {
    // internal class declarations
}
```

The interior of a class defines a new, unique scope associated with the class itself, and is composed of one or two separated sections, named `interface` and `body` of the class.
Interface

When declaring a new class, its interface is the first section that has to be defined. As the name suggests, the interface contains the portion of a class that will be accessible from outside the scope of objects of the class. An interface can contain only two kinds of declarations: instance predicates and prototypes for virtual predicates. The formers must be in the same form as discussed in section 3.4.1, and as previously stated, they can directly access the object context data for the class into which they are declared. Virtual predicates declared in the class interface on the other hand cannot be fully specified like instance predicates. A virtual predicate `pred` for class `A` must be declared outside the class `A` scope, and as such it cannot access instance data, which is declared inside the class scope. However, class `A` must know it supports the `pred` virtual predicate, and thus a virtual predicate prototype is needed inside the class declaration. The natural place where this must be specified is inside the class interface section, considering a virtual predicate can be seen as a mean to add instance-specific behavior to objects. A virtual predicate prototype in the interface is specified by the `virtual` keyword followed by the predicate signature expressed the same way as with instance predicates; being just prototypes, these however must not provide the predicate body, and must end with a semicolon instead, contrary to instance predicate which must sport a full-featured predicate body right next to the signature. In virtual predicate prototypes, the only shortcut rule that applies is the implicit parameter type, all other shortcuts are not allowed. Listing 3.15 shows our `GRect` class declaration as an example; note the class features just the interface section, which is composed of one instance predicate and one virtual predicate prototype.

```java
class GRect {
    draw() {
        // call rectangle drawing functions
        succeed;
    }
    virtual size(out width, out height);
}
```

**Listing 3.15:** An example of class with just the interface portion
3. The Alpha+ declarative programming language

Body

The class body is the section of a class where object instance variables and sub object references are declared. The section begins when the `body` keyword immediately followed by a colon is found, and ends when the final closing brace ending the class declaration is found. Instance variables can be declared inside a class body exactly the same way as elements are declared inside a classical C struct construct, there are no limitations of any sort. Sub objects for objects of this class are declared by placing constructor predicates inside the class body; suppose objects of class A each need to have one sub object of class B: listing 3.16 shows how class A should look like.

```
class A {
  body:
    B(out 1);
}
A(out 1);
```

*Listing 3.16: Tiny class sub objects example*

Here we also declare an object of class A with ID 1 at global scope; this object has an internal sub object of class B, living inside its scope with ID 1. Note that even if both objects have identifier number 1, they refer to completely different objects, firstly because they are of different classes, secondly and most importantly because they are declared in different scopes: the object instance of class A is declared in the global scope, whereas the object instance of class B is declared in the scope associated with its parent object. Also note that in this example it is assumed class B has already previously been defined, as it is not allowed to declare objects if their class is unknown. As a side effect, the programmer cannot build recursive classes: class A cannot contain objects of class A as the class has not been completely defined yet.

Instance variables declared inside the body of a class can be accessed only from predicates within the class scope, being these either instance predicates declared in the interface, either constructors declared in the body. Virtual predicates can also be declared in a class body and access instance variables, but we will examine this case in section 3.5.5.

A class can have only the interface, only the body or both them. As an example, listing 3.17 shows a more detailed implementation of the class
GRect we already approached in past listings.

```java
class GRect {
    draw() {
        // call rectangle drawing functions
        succeed;
    }
    virtual size(out width, out height);
    body:
        int w, h;
}
GRect(1).size(out 25, out 10);
GRect(out 1);
```

Listing 3.17: A more detailed class example

In this example class GRect has a virtual predicate, an instance predicate and two instance variables only. In the global scope we declare an object of class GRect with ID 1, and we also declare the virtual predicate size for this object instance. The size virtual predicate declared here defines the properties of object instance identified by number 1 at global scope, which are supposed to end up being stored into the object w, h instance variables.

Declaring a virtual predicate however does not cause it to be executed; in our execution model as seen in algorithm 1 the language executor only deals with object declarations and constructs the live context tree. It should be expected at this point that constructor predicates, defining new object instances, get called by the executor, and at a first approach these may sound like good candidates where to place calls to virtual predicates like size, to set the properties of individual object instances. While it is true constructors get called by the executor to declare the new object instances, it is however not possible to call virtual predicates from them, since as previously stated, during constructor execution the new object instances do not exist yet, so calling a virtual predicate on them is impossible. A careful read of execution algorithm 1 foresees the existence of objects initialization code though, and this is actually where we will be able to issue our calls to virtual predicates; we will discuss setup as well as cleanup of object instances in section 3.5.

It has already been stated that supporting classes in the traditional OOP style has an important consequence for Alpha+: the language though declarative in nature, is not stateless, and this can open the way for many problems as referential transparency is not guaranteed anymore due to possible side-effects due to changes in objects states. It is to be noted though,
that Alpha+ objects can provide no instance data, effectively removing side-effects and obtaining an approach more akin the pure declarative model; the choice is left up to the programmer, and this thanks to the multiparadigm support built into Alpha+.

3.4.4 Similarities between Alpha+ predicates and C functions

In the previous sections we introduced predicates and we explained how these look similar to a boolean C function. Internally, predicates use a the same imperative paradigm as classical C functions, allowing conditional if statements, for loops, and almost all the C control constructs, including function calls. Predicates have a signature that contains an optional parameters list, and can have a body containing a list of declarations and statements. Both the parameters list and the body of a predicate share almost the same syntactical rules of C functions, with a few differences. As we have seen, Alpha+ parameters list offers an easy way to declare parameters passed by reference, allowing it to use them in the predicate body without the need to dereference them.

```c
int C_Function (int * outParam) {
  *outParam = 1;
  return true;
}
```

in Alpha+ becomes:

```alpha+
Alpha_Predicate(out int outParam) {
  outParam = 1;
  succeed;
}
```

Thanks to out parameters, Alpha+ predicates gain in readability compared to C functions. Alpha+ parameters list can also benefit from shortcuts as seen in section 3.4.1.

The body of a predicate follows exactly the same syntactical rules applying to C function bodies, with the following few differences:

- It is forbidden to use the return C statement in an Alpha+ predicate;
• The Alpha+ specific statements succeed and fail must be used to exit the predicate at any time with a true or false result, respectively.

• Alpha+ predicates can use the new foreach construct. We will examine how this works in section 3.5.2;

• An Alpha+ predicate can call other Alpha+ predicates. The most generic form of predicate call looks the same as a standard C function call, and is composed of the predicate name followed by the list of parameters enclosed within brackets:

\[ \text{PredicateName( } \ldots \text{ )} \]

This works as long as the call is performed inside a predicate living within the scope of a class, and \texttt{PredicateName} is a valid predicate declared in such a class. The caller and callee predicates will both work on the same object instance in this case. Whenever a predicate needs to call a predicate for a specific object instance, the predicate call must be in the form:

\[ \text{ClassName( id ).PredicateName( } \ldots \text{ )} \]

Where the current scope, \texttt{ClassName} and \texttt{id} uniquely identify the object instance we want to call \texttt{PredicateName} for; if no such object instance exists in the current scope, the call will immediately return false as if \texttt{fail} was issued.

Being conceptually similar to boolean functions, predicates can be safely called within expressions and conditional statements.

These differences aside, Alpha+ predicate bodies can perform any kind of computation using any C statement to achieve the result.

### 3.5 Advanced Alpha+ constructs

So far we examined the basic constructs the Alpha+ programming language provides; we learned how to work with predicates and classes, and how to declare new object instances. Our execution model as presented in section 3.3.4 assumes when an object is created, a piece of code is executed in order to initialize the new object instance; similarly, when an object is freed, a piece of code is executed to cleanup the instance. The Alpha+ constructs previously introduced do not foresee such possibilities; in this section we introduce new constructs to satisfy these needs and more. We will also introduce other advanced concepts, and examine yet another important object oriented aspect of Alpha+: polymorphism.
3.5.1 Class setup and cleanup

It should be clear at this point how objects get created: via constructor predicates. Constructors themselves are only a mean to declare new objects though, they have no knowledge of the object instance they create. The object they declare itself does not yet exist until the constructor exits with succeed, allowing the Alpha+ executor to actually create an object instance in the current scope with the identifier stored in the constructor out parameter. This means it is not possible to put initialization code into an object constructor, or at least any code there cannot be used to initialize a specific object instance. To suffice the need for instance initialization code, Alpha+ introduces the concept of setup function. Similarly, a cleanup function is introduced to perform object instance cleanup.

The setup function

The setup function must be declared in the scope of an Alpha+ class; this makes perfect sense as it needs to be able to access object instance variables in order to initialize them. Specifically, the setup function is declared within the class body section, in the form of the setup keyword followed by the function body enclosed within braces. Listing 3.18 shows an example of class setup.

```alpha
class GRect {
    draw () {
        // call rectangle drawing functions
        succeed;
    }
    virtual size(out width, out height);
    body:
    setup {
        if (!size(w, h)) {
            w = 10;
            h = 10;
        }
    }
    int w, h;
}
```

Listing 3.18: Class setup function

With this example we answer the question of where virtual predicates get called for new object instances. The class GRect setup function calls the
size virtual predicate to initialize the rectangle $w$ and $h$ instance variables. If the predicate exits with fail, the variables are both initialized with a default value of 10; remember the call to size in setup also fails if the size predicate is not declared in the scope where the new object instance of class GRect is being created and initialized.

**The cleanup function**

As the setup function provides a mean to initialize object instances, the cleanup function allows it to deinitialize them: any instance resource that was allocated in the setup of an object instance should be freed in the cleanup. The cleanup function has the same form as the setup one: the cleanup keyword followed by function body enclosed within braces. As setup, cleanup too needs to be declared inside the body of a class.

Both setup and cleanup functions are optional to a class.

### 3.5.2 The foreach statement

An important Alpha+ feature is provided by the foreach statement whose purpose, like the classical C for statement, is to repeatedly execute a block of instructions as long as a condition is met. The syntax for this statement is in the form of the foreach keyword followed by a predicate call enclosed within brackets, followed by a statement or a list of statements enclosed within braces:

```alpha+
foreach( PredicateName( ... ) ) statement;
foreach( PredicateName( ... ) ) { statement list }
```

The purpose of foreach is to call PredicateName and execute the statement or statement list at the condition that the predicate has exited with succeed. Once the statements are executed a first time, foreach iterates and PredicateName is executed again. However, its execution does not restart from the beginning of its body as with a normal predicate call; it restarts right off the instruction next to the last succeed. This allows the foreach statements to be executed as many times as succeed is issued by the PredicateName predicate, until it issues a fail or the predicate code ends, in which case execution continues normally on the instructions next to the foreach block. Let us explain how foreach works with the help of the example shown in listing 3.19.

The example declares an object instance of class GCanvas in the global scope with ID 1. When the object gets created, the class setup function
class GCanvas {
    virtual line(out x1, out y1, out x2, out y2);

    body:
    setup {
        int x1, y1, x2, y2;
        foreach (line(x1, y1, x2, y2)) {
            // call line drawing function
        }
    }
}

GCanvas(1).line(out 0, out 0, out 30, out 0);
GCanvas(1).line(out 30, out 0, out 30, out 15);
GCanvas(1).line(out 30, out 15, out 0, out 15);
GCanvas(1).line(out 0, out 15, out 0, out 0);
GCanvas(out 1);

Listing 3.19: Foreach example

is called for the new object instance; this calls the line virtual predicate for such an object in a foreach. This predicate is supposed to declare a line by setting its endpoint coordinates; multiple instances of the predicate as found in the example define 4 lines. When the foreach calls line for the first time, the Alpha+ language executor finds the line predicate is declared and supports the case of object ID 1, which is the identifier for the object instance being initialized. So it calls it, and this sets the endpoints to be (0,0) – (30,0). The predicate exits at this point due to the implicit succeed, and thus execution restarts inside the foreach; since the predicate succeeded, the setup can now call the system line drawing routine for the line just declared and iterate. On the second iteration, the line predicate gets called again, but this time its execution restarts next to where the last succeed was issued, that is, on the second case that sets the endpoint out parameters to (30,0) – (30,15) and exits again due to an implicit succeed. The new line gets drawn and the foreach iterates again. This is repeated for the 4 line cases due to the fact the predicate has 4 implicit succeeds; after the fourth succeed the predicate has no more code to be executed and thus the foreach ends, resuming execution on the instruction next to the foreach block; in this case the setup function has no more instructions so execution ends. We expand the 4 line cases in listing 3.20 for clarity: this is equivalent to what seen in listing 3.19.
The argument of a **foreach** can be any valid predicate call; it can be extremely useful also to find out the list of object identifiers declared for a specific class in the current scope: to achieve this, it is sufficient to call a constructor from the **foreach**, and this thanks to the design of constructors which issue a **succeed** for every object to be created.

### 3.5.3 Generic virtual predicates

We have seen how virtual predicate declarations are associated to specific object instances; a virtual predicate declared as:

```plaintext
GRect(7).size(out 25, out 15);
```

will only work on object instance with ID 7 of the **GRect** class, into the current scope. Calls to the **size** predicate from other object instances will return **fail** unless the virtual predicate is somewhere declared for these instance cases as well. As we recall from section 3.4.1, the signature for virtual predicates must be in the form:

```
ClassName( id ).PredicateName( [parameters list] )
```

where we assumed so far **id** to be a numerical expression identifying the object instance of class **ClassName** onto which **PredicateName** operates.

Alpha+ provides a mean for a predicate to be valid for any object instance identifier in the same scope; it is sufficient that **id** is replaced by a variable identifier, which also becomes available for use in the predicate body as if it was a parameter passed to it. A virtual predicate in this form is called *generic virtual predicate*, as it can also express the same functionality of
normal virtual predicates seen so far; as a proof of this ability, listing 3.22 features a generic version of the virtual predicate seen in listing 3.21.

```
GRect(id).size(out w, out h) {
    if(id == 7) {
        w = 25;
        h = 15;
        succeed;
    }
    fail;
}
```

Listing 3.22: Generic virtual predicate

Generic virtual predicates can be useful in quite a number of situations. An interesting usage is to obtain the identifier number of an existing object instance; objects instances in fact do not have knowledge of their own identifier number. Getting them to know it can be easily achieved via a generic virtual predicate, as shown in the following listing:

```
class GWindow {
    virtual getID(out id);
    body:
        setup {
            getID(my_object_id);
        }
    int my_object_id;
}
GWindow(x).getID(id) {
    id = x;
    succeed;
}
```

Listing 3.23: Obtaining object id via a generic virtual predicate

### 3.5.4 Extern classes

By partially supporting the OOP model via classes, Alpha+ can be an easily extendable language. One of the advantages of the OOP paradigm is in fact
code reusing; in our context this means it is a reasonable goal is to have a library of intercorrelated Alpha+ classes performing different tasks. The programmer can then just use an existing class, extending it to suit his needs. There is a problem with this vision though: classes as introduced in section 3.4.3 encapsulate all the code for their (instance) predicates, so whenever the programmer has to use a class, its full specification has to be given. This greatly affects modularity favoring code duplication instead, as each Alpha+ source that uses an object of a class has to redefine the entire class for the object to be declared. Put in other terms, in our view of Alpha+ so far there is no mean for a source file to share the declaration of a class with another source file. To address this problem, Alpha+ introduces the concept of *extern classes*: an extern class $A$ assumes the full class $A$ has already been declare somewhere else, and it can contain just the interface of that class. Extern classes are declared in the same form of normal classes, but prefixing the declaration signature with the `extern` keyword:

```c
extern class ClassName {
    // interface class declarations
}
```

Opposed to normal class declarations, an extern class declaration cannot contain a body section. Additionally, instance predicates declared in the interface can only contain the predicate prototype, the same way as we already seen for virtual predicates: the predicate code must be omitted, as it is assumed to be specified in the full class declaration somewhere else, and the braces and the predicate body enclosed by them must be replaced by a semicolon. The class of listing 3.17 for example can have an external version as shown in listing 3.24:

```c
extern class GRect {
    draw();
    virtual size(out width, out height);
}
```

**Listing 3.24:** Extern class example

Note the absence of the class body, and the semicolon replacing the `draw` instance predicate body. External classes must provide the same interface provided by their full class counterpart; in addition, an external class can be declared at most once per Alpha+ source code listing, and the full class
declaration must be available in one of the listings that compose the Alpha+ project to avoid linking errors.

3.5.5 Polymorphism in Alpha+

As already stated, one of the most important aspects of Alpha+ is the support for a kind of polymorphism, partially borrowed from other object-oriented languages. In the literature [4], a polymorphic function \( f \) in an object-oriented language is a function that has multiple definitions; the function version to be executed when calling \( f \) depends upon the context into which it is called, and the binding is usually resolved at runtime (late binding). In many object-oriented languages the context in which a polymorphic function is called is determined by the object onto which the function operates; inheritance plays a central role in this view, as inherited objects can implement their own definition of function \( f \).

The Alpha+ approach to polymorphism is slightly different. In Alpha+, polymorphism applies to virtual predicates; as we recall from section 3.4.1, a virtual predicate for an object of class \( A \) must be declared outside the scope of class \( A \). When issuing a call to a virtual predicate, the Alpha+ executor applies a late binding by searching for this predicate in the same scope where the object instance which issued the call is declared. If the virtual predicate has not been declared in such a scope, the call immediately returns false as if \texttt{fail} was issued.

We will illustrate this behavior with the example in listing 3.25, whose output is shown in figure 3.2. In this example we have two object instances declared in the global scope, both with identifier 1, but one of class \( B \) and the other of class \( C \). These classes both provide one sub object instance of class \( A \) identified by object IDs 2 and 3; we chose to use different identification numbers for clarity reasons, but we could have chosen the ID 1 again as well for both them, as they are declared in different scopes, so they refer to completely different object instances. Classes \( B \) and \( C \) do not only declare one object instance of class \( A \) in their bodies; they also declare a different version of the virtual predicate \( X \) of \( A \) there. Now let us examine our program execution: when the program declares \texttt{B(out 1)}; and \texttt{C(out 1)}; in the global scope, these two objects get created in the order they are declared. Consider the case of the creation of the object instance of class \( B \); since the class does not provide a setup function, there is no setup code to be executed. Instead, the class declares a sub object instance of class \( A \) in its body via the \texttt{A(out 2)}; constructor, and this causes the creation of this instance in the context of the object instance of class \( B \). Class \( A \) provides a setup function, so when the object is created, this code gets executed and
3. The Alpha+ declarative programming language

```cpp
class A {
    virtual X(out x);
    body:
        setup {
            int x;
            X(x);
            printf("x is %d\n", x);
        }
}

class B {
    body:
        A(out 2);
        A(2).X(out 333);
}

class C {
    body:
        A(out 3);
        A(3).X(out 777);
}

B(out 1);
C(out 1);

Listing 3.25: An example of polymorphism
```

**Figure 3.2:** Output for program in listing 3.25

```
x is 333
x is 777
```
the \( x \) virtual predicate is called. Now, since we are executing the setup of object instance \( A(2) \) declared in the context of \( B(1) \), the language executor looks for virtual predicate \( X \) into the scope of class \( B \); such a predicate is declared there, so it gets bound and executed, returning the value 333 into the \( x \) variable of the class \( A \) setup function. The \texttt{C printf} routine is used to print out a formatted text; the value 333 is shown as in figure 3.2. The same applies for object instance \( C(1) \) creation, but this time the \( X \) predicate call in \( A \) setup is bound to the \( X \) implementation in class \( C \), as it is where the sub object instance \( A(3) \) is declared, and the value 777 is returned.

The availability in Alpha+ of this scope-related kind of polymorphism with virtual predicates, coupled with the \texttt{foreach} statement is a very powerful declarative mechanism. We have already partially seen this in the example of listing 3.19, where the programmer provides a series of virtual predicates to define 4 lines. The \texttt{GCanvas} class could have been shipped within a library of Alpha+ classes, leaving to the programmer just the need to declare the \texttt{line} predicate in order to display a line. And both a \texttt{GCanvas} object instance declaration and the \texttt{line} predicates defining its lines could be declared inside another object, effectively hiding the implementation and allowing the creation of custom objects reusing \texttt{GCanvas} functionalities.

### 3.5.6 Execution model revised

We have now a better knowledge of the Alpha+ declarative programming language, and it is time to refine the execution model as previously seen in section 3.3.4, in light of the concepts introduced thus far. In algorithm 1 we introduced the concept of live context tree and we roughly examined how the Alpha+ executor is supposed to create it when a program is executed. The new execution model presented here expands that algorithm, and can be considered final; it is this model that will be followed in the implementation.

**Algorithm 2 Execution model: program initialization**

for each object of class \( C \) declared in scope \( S \) associated with current context \( X \), do:

1) call constructor to determine the set of object identifiers \( L \)
2) for each identifier \( I \) of \( L \), do:

2a) create new object instance \( O \) of class \( C \) in context \( X \),

2b) call class \( C \) setup function for object instance \( O \)
2c) let \( S' \) be the scope associated with class \( C \) and \( X' \) be the context of object instance \( O \)
2d) recurs whole procedure with \( S \rightarrow S' \) and \( X \rightarrow X' \)
Algorithm 2 shows the full procedure used to initialize Alpha+ programs. The initial scope $S$ and context $X$ are the program global scope and context, respectively. The global context is the root of the live context tree; objects declared in the main program scope are in fact declared into this context. We will examine the execution of program listing 3.26. When the program starts, the Alpha+ executor finds two object instances into the global scope: $A(1)$ and $B(1)$. These are then created into the global context. The algorithm recurs into the context associated with $A(1)$, and it finds class $A$ declares three objects of class $B$. $B(1)$, $B(2)$ and $B(5)$ are then created into the context of $A(1)$. For each of these three new object instances, the algorithm recurs again, examining class $B$ and finding it declares two sub-objects of class $C$. The procedure continues until all objects declared at global scope and any of their sub objects are created; the result for program 3.26 is a live context tree as shown in figure 3.3.

![Figure 3.3: Live contexts tree for listing 3.26](image)

The introduction of the concept of objects cleanup as explained in section 3.5.1 assumes the cleanup function of a class to be called on program exit for every object instance of that class. It makes sense then to introduce an algorithm for program exit, alongside algorithm 2, whose purpose is to call cleanup functions and to free any memory occupied by the object instance contexts.
class D {
    body:
        int d;
}

class C {
    body:
        D(out 7);
        int c1, c2;
}

class B {
    body:
        C(out 3);
        C(out 1);
        int b;
}

class A {
    body:
        B(out 1);
        B(out 2);
        B(out 5);
}

A(out 1);
B(out 1);

Listing 3.26: A simple Alpha+ program with multiple sub-object declarations

Algorithm 3 Execution model: program exit
for each object instance O of class C declared in current context X, do:
1) call class C cleanup function for object instance O
2) let X' be the context of object instance O
3) recurs procedure with X \rightarrow X'
4) free any resource occupied by object instance O

Whereas algorithm 2 gets called by the language executor at program startup to build the live context tree, algorithm 3 is called when a program ends, and deconstructs it.
3.5.7 A programming example

To summarize and show the capabilities of Alpha+, in this section we present a more advanced and complete example of program; we will also show the result of the program execution in figure 3.4. This generic example features a fairly complete \texttt{GCanvas} class in listing 3.27, and uses it to display some graphical scenes. The class is declared into a separated file for modularity reasons; ideally it should be placed in a library of classes so the programmers could link to it. This separation greatly helps reducing the complexity of final programs; the example shown in listing 3.29 is a proof of how easy is to set up a graphical scene providing there is a proper class that accomplishes most of the work. The program just includes the \texttt{GCanvas.h} header file (shown in listing 3.28) declaring the extern class \texttt{GCanvas}, then it declares object instances and predicates to create the scene. A detailed study of the sources is omitted for brevity.

```c
// these two headers are supposed to contain prototypes for
// functions dealing with OS-level windows and canvases
#include <KWindow.h>
#include <KCanvas.h>

class GCanvas {

// interface section

// virtual predicates
virtual window (out KWindow* win);
virtual title (out i1* title);
virtual style (out style);
virtual origin (out x, out y);
virtual pos (out x, out y);
virtual size (out width, out height);
virtual hasFrame ();
virtual backColor(out color);
virtual line (out x1, out y1, out x2, out y2, out color);
virtual rect (out x, out y, out width, out height,
              out frameColor, out fillColor);
virtual ellipse (out x, out y, out width, out height,
                out frameColor, out fillColor);
virtual init ();

// instance predicates
getSize(out width, out height) {
    KCanvas_GetSize(mCanv, (ui4*)&width, (ui4*)&height);
    succeed;
}
```
body:
  // body section

  // instance variables
  KCanvas* mCanv;
  KWindow* mWin;
  Bool mHasExternalWin;
  ui4 mX, mY, mW, mH;

setup {
  ui4 theX, theY, theX2, theY2, theW, theH;
  ui4 theFrameColor, theFillColor;
  i1* theTitle;

  // no position specified: set default
  if (!pos(theX, theY)) theX = theY = 20;

  // no size specified: set default
  if (!size(theW, theH)) { theW = 640; theH = 480; }

  // get parent window (if any specified)
  mHasExternalWin = window(mWin);

  // first execution of setup
  if (mCanv == NULL) {
    // no window specified: create one to contain canvas
    if (!mHasExternalWin) {
      ui4 theStyle;
      if (!style(theStyle)) theStyle = KWindow_CAPTION;
      mWin = KWindow_New(theStyle);
      if (title(theTitle))
        KWindow_SetTitle(mWin, theTitle);
      KWindow_SetPos(mWin, theX, theY);
      KWindow_SetClientAreaSize(mWin, theW, theH);
      KWindow_Show(mWin);
      theX = theY = 0;
    }

    // create canvas and call user init predicate if any
    mCanv = KCanvas_New(mWin, theX, theY, theW, theH,
                         1024, 768, hasFrame());
    init();
  } else {
    if (!mHasExternalWin) {
      if (theX != mX || theY != mY)
        KWindow_SetPos(mWin, theX, theY);
      if (theW != mW || theH != mH)
3. The Alpha+ declarative programming language

KWindow_SetClientAreaSize(mWin, theW, theH);
}
if (theW != mW || theH != mH)
    KCanvas_SetSize(mCanv, theW, theH);
}

mX = theX; mY = theY; mW = theW; mH = theH;

// set background color
if (!backColor(theFillColor)) theFillColor = 0xFFFFFFFF;
KCanvas_SetBrush(mCanv, theFillColor);

// clean canvas
KCanvas_SaturateBuffer(mCanv);

// draw lines
foreach (line(theX, theY, theX2, theY2, theFrameColor)) {
    KCanvas_SetPen(mCanv, theFrameColor, 1);
    KCanvas_SetPenPos(mCanv, theX, theY);
    KCanvas_LineTo(mCanv, theX2, theY2);
}

// draw rectangles
foreach (rect(theX, theY, theW, theH,
    theFrameColor, theFillColor)) {
    KCanvas_SetPen(mCanv, theFrameColor, 1);
    KCanvas_SetBrush(mCanv, theFillColor);
    KCanvas_Rectangle(mCanv, theX, theY, theX+theW, theY+theH);
}

// draw ellipses
foreach (ellipse(theX, theY, theW, theH,
    theFrameColor, theFillColor)) {
    KCanvas_SetPen(mCanv, theFrameColor, 1);
    KCanvas_SetBrush(mCanv, theFillColor);
    KCanvas_Ellipse(mCanv, theX, theY, theX+theW, theY+theH);
}

// update canvas
KCanvas_Update(mCanv);
}

cleanup {
    if (mCanv) KCanvas_Delete(mCanv);
    if (mWin && !mHasExternalWin) KWindow_Delete(mWin);
}
}

Listing 3.27: The GCanvas Alpha+ class
#include <KWindow.h>

extern class GCanvas {
    virtual window (out KWindow* win);
    virtual title (out i1* title);
    virtual style (out style);
    virtual origin (out x, out y);
    virtual pos (out x, out y);
    virtual size (out width, out height);
    virtual hasFrame ();
    virtual backColor(out color);
    virtual line (out x1, out y1, out x2, out y2, out color);
    virtual rect (out x, out y, out width, out height,
                   out frameColor, out fillColor);
    virtual ellipse (out x, out y, out width, out height,
                     out frameColor, out fillColor);
    virtual init ();

    getSize(out width, out height);
}

Listing 3.28: The GCanvas.h class header file

#include <GCanvas.h>

// declare 4 GCanvases objects
GCanvas(out id) {
    for(id = 0; id < 4; ++id) succeed;
}

// declare common features
GCanvas(id).pos(out x=(id*70), out y=(id*70));
GCanvas(id).size(out width=200, out height=200);
GCanvas(id).title(out i1* title="GCanvas test");

// GCanvas(0) features an ellipse
GCanvas(0).ellipse(out 0,out 0,out 200,out 200,
                   out 0x000000,out 0x0000FF);

// GCanvas(1) features two rects
GCanvas(1).rect(out 0,out 0,out 180,out 180,
                out 0xFF0000,out 0xFF0000);
GCanvas(1).rect(out 20,out 20,out 200,out 200,
                out 0x00FF00,out 0x00FF00);

// GCanvas(2) features a gradient
3. The Alpha+ declarative programming language

GCanvas(2).line(out x1,out y1,out x2,out y2,out color) {
    x1 = 0;
    x2 = 200;
    for (y1 = y2 = 0; y1 < 200; y1++, y2++) {
        color = (y1*0xFF)/200;
        color = color | (color << 8);
        succeed;
    }
}

// GCanvas(3) features a graph
GCanvas(3).backColor(out 0x000000);
GCanvas(3).line(out x1,out y1,out x2,out y2,out color) {
    x1 = 0;
    y2 = 200;
    color = 0x00FF00;
    for (y1 = 0; y1 < 200; y1 += 10) {
        x2 = y1;
        succeed;
    }
}

Listing 3.29: An Alpha+ example program using the GCanvas class

Figure 3.4: Output of the Alpha+ generic example of listing 3.29
3.6 Dynamic Alpha+

The careful reader could have noticed a major problem with our design as discussed thus far. The execution model as introduced in section 3.3.4 and revised in section 3.5.6 can be successfully employed to start and stop programs, but once the program starts and object instances are created, nothing can change the status of the instance contexts during the whole program execution as there are no triggers or other mechanism that may cause conditional execution of specialized predicates. This status-quo can only be interrupted when the program ends, when all the object instances get freed.

3.6.1 Alpha+ and reactive callbacks

It is evident the inherited static execution model of Alpha+ is a huge limit for our own design goals, as a GUI or graphical scene is usually interactive, and assumes the possibility to trigger events like the simple push of a button. To overcome this limit we let Alpha+ programs run on top of a reactive environment, using the same principles and our own approach as detailed in chapter 2.

The language executor will ensure constructor predicates as well as setup functions are executed as reactive callbacks; this has the side effect of making the whole live contexts tree dynamic and reactive to state changes. A change in the internal state of an instance object for example may trigger the re-execution of the setup function if this references the object instance variables. A change in a variable used by a constructor may trigger its re-execution, with the consequence of rebuilding a branch of the live contexts tree, possibly with different objects since the constructor was last called; as an example, consider listing 3.30.

```plaintext
GRect (out id) {
    if (x <= 10) {
        id = x;
        succeed;
    }
    else fail;
}
```

**Listing 3.30:** An example of dynamic constructor

In this example the live contexts tree will be rebuilt each time the `x` variable changes, and if `x` gets bigger than 10, the `GRect` constructor will fail.
and no objects will be created. Each time an object does not exist anymore after a constructor re-execution, the language executor ensures the object instance is freed and its `cleanup` function is called.

The reactive environment so applied guarantees dynamism, but one last problem remains. From a predicate we can only reference object instance variables (in the case of instance predicates), thus only sub-objects can be made reactive as their constructors can reference an instance variable of the parent object and establish a reaction chain with it. Objects living in the global Alpha+ scope however have no mean to reference any variable to create reaction chains, as Alpha+ has no support for global program variables these constructors could access. With no reaction of the global objects, the whole system would be static as no reactive callback could be triggered.

To address this limit, we decided to allow the Alpha+ language to be embedded into the C programming language: a special `/*` tag marks the start and end of the Alpha+ code blocks of an otherwise normal C program. The Alpha+ code embedded into the C source can reference any global variable of the underlying C program, whereas the C program has no knowledge of the Alpha+ code embedded into it, as the Alpha+ block delimiter makes the C compiler to treat the block as a standard comment.

This solution allows Alpha+ code to be reactive to the variables of the underlying C program, solving the dependency problem for global Alpha+ objects, and opening a whole lot of possibilities for reactive animation and algorithms visualization.

### 3.6.2 The example of an animation

We will now demonstrate how the availability of a reactive environment onto which Alpha+ programs can be executed allows to create graphical animations. The same concept can be easily extended to support more complex interactive applications.

Consider the Alpha+ program in listing 3.31. The first noticeable thing is that the Alpha+ code is embedded inside a C program, as explained above: the `/*` tag marks the start of the Alpha+ block, and again the same tag marks its end. The underlying C program simply declares three global variables, `phase`, `freq` and `amp`, and enters an infinite loop; during each loop cycle, it just sets the contents of these variables, that are supposed to be the phase, frequency and amplitude of a sinusoidal wave, which depend upon the `x` variable, increased at each cycle. Purpose of our Alpha+ program is to visualize the graph of this wave, and animate it as `phase`, `freq` and `amp` are changed by the C program.
```c
#include <stdio.h>
#include <math.h>
#include <GCanvas.h>

#define W 400
#define H 200
#define PI 3.14159236

float phase, freq, amp;

/*
GCanvas(out 1);
GCanvas(1).size(out W, out H);
GCanvas(1).line(out x1, out y1, out x2, out y2, 
    out color = 0xFF) {
    float angle, inc;

    angle = phase;
    inc = freq / W;
    for (x1 = 0; x1 < W; x1++) {
        x2 = x1 + 1;
        y1 = (H/2) + (sin(angle)*amp);
        y2 = (H/2) + (sin(angle+inc)*amp);
        angle += inc;
        succeed;
    }
}
/*

void main()
{
    float x = 0;

    for (;;) {
        freq = sin(x) * 10 * PI;
        amp = cos(x / 2) * (H / 3);
        phase = tan(x);

        x += PI / 50;
    }
}

Listing 3.31: Alpha+ animation example
```
We reused the GCanvas class defined in the listing 3.27 to produce the graphical output. In our Alpha+ program we just declare an object of class GCanvas describing the graphical context onto which we are going to display the animation, its size via the size virtual predicate, and the lines to be drawn inside it with the line virtual predicate. The implementation of this last predicate holds a cycle that allows the GCanvas object to plot the lines that constitute the sinusoidal wave graph; the endpoints of these lines depend upon the phase, freq and amp variables of the underlying C programs. Being the line predicate called from the GCanvas setup function, and being the GCanvas setup a reactive callback as explained in the previous section, the whole setup function is re-executed in the context of the reactive callback whenever one of phase, freq and amp is changed, with the consequence of updating the screen with a new graph plot. The continuous change of these variables in the C program automatically causes the GCanvas object setup to be executed continuously as well, effectively producing an animation of the graph; a frame of this animation is shown in figure 3.5.

![Image](image.png)

**Figure 3.5:** An output frame of the animation for program in listing 3.31

### 3.7 Discussion

In any programming language design there is a trade-off between the range of computations that the language can express and how well the language supports its target programming paradigm. We introduced Alpha+ as a declarative programming language, though it quickly became evident that the declarative approach is just the "outer" programming model available to Alpha+ programmers; programs are made up of declarations, but these can count on both object-oriented and imperative features and constructs. Even supporting multiple paradigms, the declarative model that Alpha+ provides
3. The Alpha+ declarative programming language

to programmers enforces the use of high-level declarative abstractions that provide a safe, portable model for a focused set of applications. In this section we will discuss the scope of the language, its advantages and current limitations with respect to the target set of applications.

3.7.1 Scope

Declarative languages are best suited for descriptive problems, and Alpha+ is not different. When we first envisioned the language, our target has been easy prototyping of graphical scenes representation, animations and GUIs. In such a scope a declarative approach as followed by Alpha+ works best, as the declarative paradigm is a natural choice to describe scenes and dialogs. A scene is composed of various elements, and Alpha+ provides a way to both describe these elements – via classes – and declare them. The example in section 3.5.7 showed how it is easy to build up a new graphical scene, providing there is a class that implements the elements behavior, in that case `GCanvas`. The separation of the class declaration from the program itself as pointed out in that example also makes it easy to foresee the benefit of a possible underlying library of built-in classes that suffice most of the graphical and UI element implementations; it would be then easy for programmers to design complex scenes by reusing predefined components, by just declaring new objects and their properties via constructors and virtual predicates.

The concept of sub-objects also makes it easy to incorporate features inside custom objects; supposing the existence of a standard class library, the programmer could easily define new classes using predefined objects, effectively extending them. To a certain degree, this helps reducing the need for inheritance in the classical object-oriented point of view; think of a `GBox` class declaring four `GLine`s in its body to define the box outline for example. The possibilities are countless.

While the declarative approach is very well suited for descriptive problems, it also greatly reduces the field of possible usages for Alpha+. Though possible, it is not recommended to use the language for heavy computational tasks and generally for non-interactive applications. Alpha+ in fact forces programmers to express basic computations in a declarative manner; as an example the factorial function we implemented in Alpha+ in listing 3.2 forces the programmer to declare the factorial function as a predicate inside a class, and then requires the existence of an object instance of that class for the program to work. Also note that being such example one of the first we presented in this chapter, we omitted the class setup function for clarity, but it would be needed for the program to be correct. The long and tedious work required makes it clear that it is counterproductive to use Alpha+ for computations
3. The Alpha+ declarative programming language

that are more naturally expressed in an imperative language.

### 3.7.2 Advantages

A direct advantage of the strong declarative nature of Alpha+ is it is easy to program with the language, as already pointed out by many examples. Declarations work on objects which are instances of a class; the use of classes has the side effect of increasing the modularity of programs, and it promotes data encapsulation and information hiding, well known and appreciated concepts in object oriented languages. Inside predicates it is then possible to use classical imperative constructs to have full control on predicate execution flow. From this small discussion it should appear clear that Alpha+ tries to integrate the best out of the declarative, object-oriented and imperative programming paradigms; the use of classical imperative and object oriented approaches usually require much more overhead and work by the programmer to achieve the same result achieved by single Alpha+ declarations. To our scope, this greatly reduces the development time to create graphical representations which could be quickly done in a few lines of Alpha+ code, and example 3.29 is a proof of this as a rather complex scene as shown in figure 3.4 is achieved via a very small program.

Syntax-wise, the clever use of shortcuts and the fact Alpha+ uses the same syntax of C, though with some enhancements and limitations, add to the ease of use and transition by programmers accustomed to classic imperative languages.

Thanks to late binding and the intrinsic nature of virtual predicates, Alpha+ allows the programmer to leave object properties unspecified, in which case classes can specify default values. Coupled with a library of classes, this can further reduce the programmer’s comprehensive effort.

The language model as it has been designed also hides to the programmer all the memory allocations/deallocations; objects get created and freed transparently to the user.

Reducing the workload on the programmers lets them concentrate on program design issues rather than spending time on making programs to actually work. This is yet another reason as to why the declarative approach has been chosen, and it is why Alpha+ perfectly fits its niche of utilization.

The problem of the static nature of graphical scenes constructed using a declarative approach is overcome using a reactive environment, as it was described in section 2.3. Coupled with reactive callbacks, Alpha+ can prove to be a valid solution to quickly prototype interactive graphical scenes and animations as shown in example 3.31.
3. The Alpha+ declarative programming language

3.7.3 Limitations

While Alpha+ enforced declarative approach allows concise and readable program structures, it severely limits the control the programmer has over the program flow of execution; this is an inheritance common to all declarative programming languages due to the intrinsic separation of the what a program describes from the how it gets executed. Even with the support of a reactive environment running underneath the programs, Alpha+ applications still limit the control the programmer has over the execution flow. Rather program execution follows the data flow, coherently with the definition of the dataflow architecture, basis of a reactive environment. This lack of direct control greatly affects the possible usages of Alpha+; though it can be successfully employed to produce graphical scenes, data visualization, animations and GUIs, which are indeed all design targets for the language, it is still possible thru special techniques but not recommended to use it for other applications that require more execution control.

As it has been designed, Alpha+ tries to support many object oriented paradigms like information hiding and a kind of polymorphism; despite the support for sub-objects, the language can however suffer the lack of a real inheritance support at the class level. It is not possible for example, at the current language state, to force the declaration of sub-objects for an existing object class; the whole class would have to be re-declared for the task, while with inheritance this problem could be easily solved by deriving the new class holding sub-objects from the original class not holding them. This limit could be however removed in future releases with relatively few efforts.

3.8 Conclusions

In this chapter we have introduced the Alpha+ declarative programming language. We have learned what the language has to offer in terms of features and capabilities, what is the language target audience and how it can help easing development in such a context. Finally, we have examined the advantages and limitations of the design to find the language has the needed expressive power to easily describe graphical scenes with support for interaction thru the employment of a reactive environment provided by reactive callbacks.

In the next part of the thesis we will discuss how the language and its executor providing the reactive environment support were implemented.
Part II

Implementation
Chapter 4

The compiler

4.1 Introduction

Common approaches to development languages implementations are either the interpreted or the compiled models. In the former, a program is interpreted by the language executor, also called interpreter; advantages of this model include easier debugging support, runtime errors checking, faster development cycle and easier language maintenance. Interpretation however lacks the speed of native execution due to the additional language interpretation layer needed. In the compiled model, a program is compiled to machine code and then executed directly on the underlying hardware; this guarantees the highest execution speed, at the cost of lesser control over the program execution itself. There exists an intermediate approach, which tries to get the best out of both these worlds: in this additional model, a compiler produces machine code for a virtual machine, which does not exist in the form of hardware. The virtual machine (VM) is another application that implements the functionalities of the target machine in software, and executes the executable units produced for it by the compiler. While it can be argued the VM interprets the executable units, this approach gives much better performances and the same advantages of the classical interpreted model; contrary to classic interpretation in fact, the VM does few runtime checks, as the compiler already ensures a program has valid syntax and semantics, before producing the executable unit the VM executes. Probably the most famous programming language implemented using the approach of a so called stage compiler and associated virtual machine is Java [20].

To implement the Alpha+ language, we will also use this approach. We will therefore divide the implementation in two distinct phases: in a first phase, we will concentrate on the development of a stage compiler which cre-
4. The compiler

ates executable units out of Alpha+ source files. In a second phase, we will
develop a virtual machine whose purpose is to interpret the machine code
loaded from the executable units generated by the compiler. This virtual
machine assumes the role of the language executor we often encountered in
previous chapters, and is responsible for the program output. This develop-
ment process separation divides the workflow as explained in figure 4.1.

As we recall from chapter 2, we want to let Alpha+ programs to be
embedded inside host C programs to allow reactive programming. This would
require us to implement a full C compiler, which is out of scope for this thesis.
We chose instead to add the Alpha+ compiler as a subsystem of an existing C
compiler, and we used the one from the Leonardo Computing Environment
(LCE). The LCE open-source project provides an integrated development
framework consisting in a full ANSI-C compatible compiler, a virtual machine
and a support library which provides common data management components
and multiplatform GUI handling. The choice of LCE as a basis for the
Alpha+ implementation perfectly fits our language design targets and eases
development; the project is completely modular, the virtual machine is easily
extendible for our purposes and the support library (called Leonardo Library
– LL for short) allows it to easily output complex graphical scenes. The whole
LCE project is written in ANSI-C, guaranteeing portability and speed.

In this chapter we will focus on the development of the Alpha+ stage
compiler as a subsystem of the Leonardo C compiler.
4.2 Compiler architecture

The Leonardo C compiler uses a modular architecture that divides the various stages of the compile process and uses several components to operate. Each module is handed by one or more components; a component is an independent entity that interacts with other components while hiding its internal state, much like an object instance in object-oriented programming. Components are implemented as sets of C functions and provide a standardized interface; all components have in fact at least these two functions:

```
ComponentName *ComponentName_New();
void ComponentName_Delete(ComponentName **);
```

The first function allocates the `ComponentName` structure holding component internal data, and performs any initialization needed. The second function frees any previously allocated resources and deletes the internal data structure. Components can then provide other interface functions performing specialized tasks, following the `ComponentName_FuncName` naming convention.

4.2.1 Modules and components

Figure 4.2 shows the various compiler modules and their connections. We will now examine the purpose of all the modules and their related components, to better understand where the Alpha+ programming language support subsystem has been added.

**C preprocessor** The preprocessor reads the input C source code passed in a memory buffer and incorporates into it included files, expands macros, removes unreachable code cut off by conditional compiling directives and removes comments. If no errors occur, it outputs the resulting preprocessed source code. The preprocessor is handled by the `CCPreprocessor` component, which makes use of three sub components to perform its tasks: `CMacroList` deals with macros expansion, `CParsing` is used to parse the input source code, `CMappingTable` keeps track of original source code lines mapping after include files inclusion.

**Lexer** This module analyzes the input source code passed to it by the preprocessor and divides it into tokens suitable for C language parsing. It is handled by the `CCLexicalAnalyzer` component.
4. The compiler

Figure 4.2: Compiler architecture layout

**Parser** Probably the most important compiler module, this implements a simple top-down recursive descent parser for the C programming language. It fetches tokens from the lexer and builds up unbound object code via the assembler module functionalities. It also identifies useful symbols and stores them using the symbol table module; these symbols will also be used by the linker module for address binding and unreachable code removal. The parser does not perform any kind of optimization when generating object code at the moment; the module however can be easily expanded in the future to apply them. **CCCompiler** is the only component of the module, and its name derives from its task which is effectively to compile an input C source code into object code.

**Linker** Role of the linker is to merge one or more object codes, resolving symbols addresses and removing unused symbols. The linker takes the last step in the compile process; it outputs an executable unit, which can then be executed by the virtual machine. The **CLinker** component handles the link process.

**Assembler** This module, used by both parser and linker, helps on creating and managing blocks of target machine code. It consists in three components: **CCodeBlock** handles plain blocks of machine code, **CObjectCode** is used to store a list of code blocks and **CExecUnit** helps creating executable units. We will learn more about the target machine code in section 4.2.2.
4. The compiler

Symbol table The symbol table manages the list of symbols found in a program, sharing it between the parser and the linker. Each symbol is part of a scope, and has an associated type node that determines its data type. Symbols, scopes and type nodes are all managed by the unique CSymTab component. The symbol table also manages program labels via the CLabelTab component.

Messages handler Preprocessor, lexer, parser and linker can all generate error and warning messages; the messages handler module provides an unified interface to store these messages, so they can be later showed to the user when the compile process ends. The messages handler provides its functionalities via the CMessage component.

4.2.2 Target machine code

Before discussing Alpha+ language implementation and integration in the above architecture, it is necessary to introduce the target machine code the compiler will output, and how the virtual machine interprets it. The Leonardo C compiler and virtual machine share an instructions set we will reuse when generating Alpha+ compiled code; for this the CCodeBlock component will be used. The instructions set has a total of 108 instructions, and supports a traps mechanism, similar to system calls in operating systems; there can be any number of traps, and they are all accessed via the same virtual CPU instruction. Traps have complete control over the virtual CPU once executed, and thus they are a powerful mean to expand the CPU functionalities without adding new hardcoded instructions. As we will see, while developing Alpha+ support we will need extended functionalities not found in the virtual CPU, and we will use traps in order to implement them.

The Leonardo VM is a stack based machine; instructions can only load and store values on the stack, and any operation is performed on data available on the stack itself at any given time.

The virtual CPU provides five registers with specialized meaning, as reported in table 4.1. The PC and SP registers are used internally by the VM and cannot be accessed directly by any instruction; their contents are implicitly accessed and modified by particular instructions however. R0, R1 and R2 can only be accessed for reading; there exist specialized instructions that load on the stack the value held them with an optional offset added. The address so loaded on top of the stack can then be manipulated to read or write global or local variables, and parameters passed to a function.
4. The compiler

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>Program counter. Holds the address of the next instruction to be executed.</td>
</tr>
<tr>
<td>SP</td>
<td>Stack pointer. Holds the address next to the topmost stack element.</td>
</tr>
<tr>
<td>R0</td>
<td>Global data pointer. Used to store the address of the global program static data. When loading/storing global variables on the stack, their address will be derived as offset from the value held in R0.</td>
</tr>
<tr>
<td>R1</td>
<td>Parameters pointer. This holds the address of the first parameter on the stack passed to the current function. Parameters can be accessed by dereferencing the address stored in R1 with a proper offset added.</td>
</tr>
<tr>
<td>R2</td>
<td>Local data pointer. Stores the address of the first local variable for the current function. As R1 can be used to access function parameters, R2 can be used to access any local variable.</td>
</tr>
</tbody>
</table>

Table 4.1: Virtual CPU registers

4.2.3 Alpha+ integration

Now that we have introduced the general architecture of the Leonardo C compiler, it is time to start integrating Alpha+ support in it. Since we are building onto an existing compiler codebase, it is natural to reuse as much functionalities from it as possible; we will reuse all the modules we previously showed in figure 4.2 – with some modifications where needed – except for the parser and linker modules. The first, as a recursive descent parser, is too much specialized to parse the C programming language alone; the second lacks the addition in output executable units of the necessary glue code performing Alpha+ programs execution as explained in algorithms 2 and 3 at pages 63 and 65.

These holes can be filled by introducing two Alpha+ subsystems, one for the parser and the other for the linker. These are considered subsystems as they are really extensions to the Compiler and Linker components, reusing functionalities from them if possible. We will examine in detail the Alpha+ parser and linker in sections 4.4 and 4.5; figure shows an updated compiler architecture layout considering the needed additions.

Both parser and linker use the symbol table and assembler modules to generate an executable unit. We said we can reuse the same instruction set and thus the same functionalities as provided by the assembler module
for our purpose, but the addition of the Alpha+ language support requires modifications to the symbol table though, as new data structures are needed. In the next section we will start examining these modifications.

4.3 The symbol table

The symbol table has a central role in the compiler design, as it manages all symbols, the scope they are declared into and information on the data type associated to each of them. The Alpha+ notions of classes and predicates require the addition of specialized symbol types as we will see. To better understand how this was achieved though, it is necessary to introduce a general overview of the inner workings of this important module. It is important in fact to have a good knowledge of the data structures used in this module, as these will be shared both by the parser and linker modules.

4.3.1 Overview

With the only exception of program labels handling, all work of the symbol table is performed by the CSymTab component. Basically, CSymTab acts as a container for scopes, symbols and their types, and its interface functions provide a mean to add, delete, scan and generally manage them.

A scope entry has almost the same form as reported in listing 4.1; unuseful fields to our discussion have been omitted here for clarity.
4. The compiler

typedef struct TScopeNode {
    ...
    TSymbolNodeIdx mFunction;
    TScopeNodeIdx mSuperScope;
    TScopeNodeIdx mNextScope;
    TSymbolNodeIdx mSymbolListTail;
    TSymbolNodeIdx mDeclarations;
} TScopeNode;

Listing 4.1: Structure of the generic scope node entry

Before examining the meaning of the different fields, it is recommended to
discuss the concept of scope in the compiler. A scope defines a space where
unique symbols can exist; each scope can have a parent scope which can access
the contents of the child scope, while the opposite is not allowed. Usually a
new scope is created in C when a new function is defined; the function is said
to define a new scope. Now, how these concepts reflect in our scope structure
is straightforward: mFunction is a reference to the symbol of the function
defining the scope, if any; mSuperScope holds a reference to the parent scope
node and mDeclarations points to a linked list of symbols declared inside
the scope. mSymbolListTail is used to identify the end of such a list, and
mNextScope points to the next scope node in a linked list fashion.

As scopes, symbols are also stored in a linked list of symbol nodes. The
symbol node structure is reported in listing 4.2, again omitting unneeded
fields for clarity.

typedef struct TSymbolNode {
    ...
    TIdentifierCategory mCategory;
    ui4 mName;
    TTypeNodeIdx mTypeIdx;
    union {
        struct { ... } mVariable;
        struct { ... } mFunction;
        struct { ... } mConstant;
    }
    mVal;
    ui4 mNextSymbol;
} TSymbolNode;

Listing 4.2: Structure of the generic symbol node entry
The **mName** identifies the symbol identifier name; it is an integer because it is really just an index into a pool of identifiers. This pool stores all unique symbol names so that for checking if two identifiers have the same name it is sufficient to compare their indices. **mCategory** holds the symbol category; valid categories are **Ide_TYPE**, **Ide_STRUCTUNICATION_ENUM_TAG**, **Ide_VARIABLE**, **Ide_FUNCTION** and **Ide_CONSTANT**. **mTypeIdx** specifies the index of the type node associated to the symbol; more about this below. The **mVal** union holds sub-structures defining symbol data pertaining to specific categories; we have **mVal.mVariable** defining variables related information, **mVal.mFunction** for function symbols data and **mVal.mConstant** for constants data. At last we have a **mNextSymbol** field pointing to the next symbol in the current scope.

The last important structure to introduce in our symbol table discussion is the type node structure. Types in the compiler are stored in a direct acyclic graph (DAG); each DAG node acts as a basic type or as a container for other types. Every time a new symbol is found, its type is searched for in the types DAG; if found, the node index is associated to the symbol in its **mTypeIdx** structure field, otherwise a new type node is constructed. The types DAG greatly helps in dealing with types handling; its uses however also span over **structs** and **unions**, **enums**, and also cover function parameters list (as a function parameter can be uniquely identified by its position and type). The type node structure is reported in listing 4.3.

```c
typedef struct {
    ... 
    TTypeSpecifier TypeSpecifier;
    TTypeSize Size;
    union {
        struct { ... } Base;
        struct { ... } Enum;
        struct { ... } EnumField;
        struct { ... } StructUnion;
        struct { ... } StructUnionField;
        struct { ... } Pointer;
        struct { ... } Array;
        struct { ... } Function;
        struct { ... } Parameter;
            Val;
    }
} TTypeNode;
Listing 4.3: Structure of the generic type node entry
```
The `TypeSpecifier` field identifies the type node kind, such as `int`, `float`, `double`, `char`, `void`, `struct`, `union`, `enum`, pointer, array or function. The type size in bytes is specified in the `Size` field. As seen for the symbols structure, the `Val` union of `TTypeNode` holds sub-structures defining data pertaining to specific kind of types.

These three data structures are extremely important all over the compiler; to better understand how they interact with each other, we will introduce a small example. Listing 4.4 shows a small C snippet defining a function; the corresponding data structures will look like in figure 4.4.

```c
struct test {
    int a;
    float b;
};
int globalvar;

int func(struct test *param1, float param2) {
    float temp = 10;
    param1->a = param2 * param1->b;
    return param1->a + (temp * globalvar);
}
```

Listing 4.4: A small C function

![Figure 4.4: Data structures layout for example in listing 4.4](image-url)
From figure 4.4 we can note a few things. First, each symbol lives in an unique scope; some symbols can be associated with a new scope, like in the case of the `func` function. Each symbol has then an associated type node; a type node can be of many kinds, and each node kind can have references to one or multiple other nodes, forming a direct acyclic graph. Note that a node can be referenced by an unlimited number of other nodes; the node identifying the `float` type in figure 4.4 for example is referenced four times: directly by the `param1` and `param2` symbols, by a field node attached to a struct node and by a function parameter node.

Other than holding information on existing symbols, scopes and types, the advantages of these data structures also include easy type checking – it is sufficient to compare sub-trees of the types node DAG – and reduced memory footprint. We will now examine how our design of the Alpha+ programming language can be fitted into this view, possibly with the inclusion of additional structures.

4.3.2 Extending symbol table for Alpha+ support

In chapter 3 we introduced the language design, and we stated Alpha+ programs are basically made up of declaration of classes and predicates. So it makes sense to add new structures in the symbol table to handle them. About predicates, we have also seen that they are similar in concept to C functions, and this effectively affects the implementation. The Leonardo C compiler already provides an `Ide_FUNCTION` category for function symbols, and a `TTypeNode_FUNCTION` specifier for function type nodes. While the last one can be reused also for predicates as we will see, we need to introduce a new category for predicate symbols as these need to store additional information pertaining the predicate they are associated with. About classes, at the data level these act as containers for both instance variables, but they also need to store references for predicates declared within them. The existing structure that most closely matches our need in the symbol table is the `TTypeNode_STRUCT_UNION` type node, which can have sub nodes of kind `TTypeNode_FIELD` holding references to any other type node. This look promising for our needs and in fact we will use the same structure used by this type node, but we will tag it differently as `TTypeNode_CLASS` to distinguish classes from normal structs or unions. About class symbols we will need a completely new category as there is no existing category that we can reuse. We will now discuss in detail the implementation of both predicates and classes data structures.
Predicates

Internally a predicate is basically a C function, even though with some differences. So it makes sense to use the `TTypeNode_FUNCTION` type specifier when constructing the type node associated with a new predicate symbol. This type node has the following structure for its `Val` `TTypeNode` field:

```c
typedef struct {
    ...
    TTypeSpecifier TypeSpecifier;
    TTypeSize Size;
    union {
        ...
        struct {
            ...
            TTypeNodeIdx TypeIdx;
            TTypeSize ParamSize;
            TTypeNodeIdx Arguments;
            TTypeFuncKind Kind;
        } Function;
    } Val;
} TTypeNode;
```

Listing 4.5: Function type node also used by Alpha+ predicates

`TypeId` holds a reference to the type node describing the function return result. As predicates are always boolean functions, we will always link in this field a reference to a `long int` type node; the predicate will return 0 to indicate a false result, and any other value will indicate a true result.

`ParamSize` and `Arguments` deal with the function parameters list, and can be reused as they are for predicates. `ParamSize` returns the total size in bytes of the required parameters, while `Arguments` is a link to a linked list of type nodes of `TTypeNode_PARAMETER` kind. Each of these will have a link to another type node describing the $n^{th}$ argument.

Parameters however require a little more attention: contrary to standard ANSI-C, Alpha+ introduces `out` parameters, which are parameters passed by reference to a predicate. Internally, we implemented `out` parameters adding the new `TTypeNode_REFERENCE` type node kind; this is attached to each `out` parameter type node, and basically acts like a pointer, linking to another type node specifying the referenced type. When a variable associated to a `TTypeNode_REFERENCE` type node is accessed, the compiler automatically dereferences it.
The fields just described can already supply all the information needed to describe a function. A predicate however, can be of three different kinds: a constructor, an instance predicate or a virtual predicate. Since we use the same type node to identify C functions as well as all the predicate types, it makes sense to add an additional field to distinguish between them; this is the purpose of the Kind field, defined as an integer value serving as a bitmask. Normal C functions have a TTypeFuncKind_C_FUNCTION value, instance predicate get TTypeFuncKind_INSTANCE_PREDICATE and virtual predicates TTypeFuncKind_VIRTUAL_PREDICATE. Constructors will assume the same value of instance predicates, as it is not necessary to distinguish between the two at the type node level; the distinction will be made inside the predicate symbol node rather than in the type node.

Function type nodes identifying predicates are associated to predicate symbol nodes. A predicate symbol node uniquely identifies a predicate in the scope where it is declared, and uses a new symbol category identified by the Ide_PREDICATE constant. We added the new mPredicate structure to the mVal union field of structure TSymbolNode; this looks like the following listing:

```c
typedef struct TSymbolNode {
    ...
    union {
        ...
        struct {
            Bool misVirtual;
            Bool misConstructor;
            ui4 mExprSourceIdx;
            ui4 mCodeIdx;
            ui4 mMaxForeachCount;
            TScopeNodeIdx mScopeIdx;
            Bool misAlphaScope;
        } mPredicate;
    }
    mVal;
} TSymbolNode;
```

**Listing 4.6:** mPredicate structure in the symbol node entry

misVirtual and misConstructor are used to identify whether a predicate is virtual or not, and if it is a constructor or not. Any combination of these two boolean variables, with the only exception of both them being true, fully describes the kind of predicate the symbol refers to. This may look redundant as a symbol as an associated type node that already specifies similar
information, however having the distinction in the symbol node as well eased development and offers more efficiency as there is no need to dereference the type node to obtain the predicate kind.

\texttt{mCodeIdx} stores a reference to the block of machine code generated for the predicate; this will be generated during the parse process. The meaning of the other fields will be explained later.

Similarly to what seen for a C function in figure 4.4, figure 4.5 shows a graphical representation of how information on a predicate are stored internally in the compiler; a new predicate symbol node is created in the scope where the predicate is declared, and the symbol name is mangled so that it always assumes the form \texttt{Class:Predicate}. A series of type nodes are then attached to the symbol node, further describing the predicate.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure45.png}
\caption{Predicate data structure representation}
\end{figure}

\textbf{Classes}

We said Alpha+ classes act as containers for objects instance data, so it makes sense to reuse the same layout as used by the \texttt{TTypeNode\_STRUCT\_UNION} type node. To avoid confusion, we introduced a new tag for the type node: \texttt{TTypeNode\_CLASS}. Like its sibling, this just stores a reference to a linked list of \texttt{TTypeNode\_FIELD} nodes; each \texttt{TTypeNode\_FIELD} node has a reference to the type node describing the \textit{n}th field type; listing 4.7 reports this type node internal structure. \texttt{Name} is the field name index in the identifiers pool, \texttt{Offset} is the field offset in the memory representation of an object instance data expressed in bytes, \texttt{NextStructField} is a reference to the next field in the class fields linked list, and \texttt{TypeId} is the reference to the type node describing the field type. Contrary to structs and unions, a type node referenced by an Alpha+ class field node can be a function type node; this serves to describe all kinds of predicates declared within the class, and has two main roles: it is
used to compare virtual predicate prototypes with their actual declaration, to ensure the signatures match, and it is used to ensure calls to a predicate provide the right arguments for it. Instance variables declared in the class body all have a valid Offset value, as the VM will have to know where in memory each of them is located; Offset in fields referencing function type nodes for predicates however is not meaningful.

```c
typedef struct {
    ...
    union {
        ...
        struct {
            ...
            TIdex IdexName;
            ui4 Offset;
            TTypeNodeIdx TypeIdx;
            TTypeNodeIdx NextStructField;
        } StructUnionField;
    } Val;
} TTypeNode;
```

Listing 4.7: Field type node also used by Alpha+ classes

As with predicates, a new symbol category has been dedicated to class identifiers, named Ide_CLASS. The class symbol structure is reported in listing 4.8. The only class symbol information the parser has to keep track of is the scope associated with the class, and this is achieved via the mScopeIdx field. The mClassRecordIdx is be used internally by the linker instead.

Like C functions, Alpha+ classes have an associated scope. A class scope contains constructor and instance predicate symbols declared within the class; virtual predicates supported by the class are an exception though as they are declared outside the class and are thus not part of the class scope. As an example of how class data is handled internally, assume listing 4.9 shows a simple class declaration implementing a GUI alert dialog box (the class is far from being complete, and much functionalities are omitted for brevity); its internal data representation is showed in figure 4.6.

We have omitted the portion of type nodes DAG relative to the class predicate nodes for brevity. There are a few observations we can do by looking at the figure:

- The class symbol is declared in the global Alpha+ scope. This is always
4. The compiler

typedef struct TSymbolNode {
    ...
    union {
        ...
        struct {
            TScopeNodeIdx mScopeIdx;
            ui4 mClassRecordIdx;
        } mClass;
    } mVal;
} TSymbolNode;

Listing 4.8: mClass structure in the symbol node entry

class GAlertDialog {
    virtual text(out i1 *text);
    virtual pos(out x, out y);
    getText(out i1 *text);
    getPos(out x, out y);
    body:
        GButton(out 1);
        GButton(1).text(out "Ok");
        int x, y;
        i1 *text;
}

Listing 4.9: A simple class implementing a GUI alert dialog

ture for all classes, and ensures their availability is spread over the whole program.

- Virtual predicate symbols of class GAlertDialog do not appear inside the class scope; the class type nodes DAG has type nodes for the virtual predicates though. This is coherent with the Alpha+ language design.

- Symbol names for instance predicates of GAlertDialog are mangled so that they are stored as Class:Name. Symbols for constructor predicates like GButton are not mangled instead.

- The GButton::text symbol appears in the class scope because the text virtual predicate of class GButton is declared in the GAlertDialog class body.
Figure 4.6: Symbols, scopes and type nodes for class in listing 4.9

- The GButton: text virtual predicate and the GButton constructor predicate, though being symbols declared in the class scope, do not have a field type node in the class types DAG, as they are not part of the class in a strict sense.

The symbols name mangling is necessary to avoid conflicts with symbol names in the same scope; as the example showed in fact, in the same class scope could exist symbols for the class instance predicate, as well as for virtual predicates of sub-objects declared in the class body. If a sub-object virtual predicate has the same name of a class instance predicate, the two symbols would conflict if no name mangling is applied.

4.4 The Alpha+ parser

Now that we have introduced the needed internal data structures, it is time to start discussing the actual Alpha+ programs parsing and code generation, which are handled by the Alpha+ parser subsystem. The Alpha+ parser, like its sibling C parser, is a recursive descent parser. Since Alpha+ is meant to be a hosted language as introduced in section 3.6, the first required step is to detect whenever to switch control from the C parser to the Alpha+ parser.
We decided to use the special /*/ token as separator; any text enclosed by a pair of /*/ is parsed by the Alpha+ parser, and we call this text an Alpha+ block. Multiple Alpha+ blocks can be inserted in a C program, with one restriction: all blocks must be placed outside any C function, at the global program scope. This technique has the advantage that other C compilers will detect the embedded Alpha+ program as a standard C comment, and thus they will simply skip it. Our compiler instead will make good use of that program via the integrated Alpha+ parser and code generator.

4.4.1 About scopes

The first problem we have to face is about scopes. We said Alpha+ programs are to be embedded into C programs inside Alpha+ blocks; the sum of all blocks found embedded in a C program constitutes an Alpha+ program. It is however favorable that all these blocks share the same root scope, so that an object of class A can be declared in an Alpha+ block even if class A itself was declared into another block. All symbols found by the parser are in fact attached to the current scope, so to make symbols declared in an Alpha+ block visible to all the Alpha+ blocks, we need all them to share the same scope. This makes sense as all blocks are part of the same Alpha+ program; we call this root scope the global Alpha+ scope.

```
/*
class A {
    virtual pred(out x);
}
/*
...
/*
A(out 1);
/*
...
/*
A(1).pred(out x = 10);
/*
```

Figure 4.7: The global Alpha+ scope

Our problem stems from the fact the Leonardo C compiler allows it to create and enter a new scope and exit it, but does not allow to re-enter an old scope once this has been exited. We solved this issue by adding
new functionalities to the CSymTab component to enter and exit the global Alpha+ scope at any time; this unique scope is created only once the first time the CSymTab_EnterAlphaScope function is called, and reused on subsequent calls.

4.4.2 The parse process

The Leonardo compiler architecture does not make use of intermediate code generation. This means the output of the parser is not an abstract syntax tree or another intermediate structure holding parse information; instead, the parser directly outputs unbound machine object code. While this is actually a simplistic approach – it does not allow for almost any kind of optimizations – it has the advantage of greatly simplifying the whole compile process; the compiler codebase is young, so further architectural changes are possible in the future, though out of scope in this thesis. For the Alpha+ parser, being it integrated as a subsystem of the existing C parser, we used the same approach.

The whole Alpha+ parser and code generator is part of the CCCompiler component and is located into the CCCompiler_Alpha.c source file; the _AlphaUnit function is the parser entry point, and gets called by the C parser when it detects the /*/ token outside the scope of any C function. The role of this function is to temporarily enter the global Alpha+ scope and call other functions implementing the Alpha+ grammar rules (as reported in appendix A) until another */ token is found, causing the exit of the global Alpha+ scope and the control to be relinquished to the C parser. Since Alpha+ programs are made up of declaration of classes and predicates, it makes sense to have two functions, _AlphaClassDeclaration and _AlphaPredicateDeclaration, implementing the parsing rules for them. Each function implementing a grammar rule returns a boolean result; it returns true if the corresponding rule can be applied and false otherwise. Note that even if the function returns true, it does not mean the parser fully succeeded; the first portion of a grammar rule could have been satisfied, while a syntax or semantic error is encountered, causing the parser to skip the rest of the rule, still returning true though as the rule was considered as applied.

The Alpha+ parser, like its C sibling, makes use of the CCodeBlock component to aid in the generation of target machine code. The component allows to create new machine codeblocks – which merely act as machine code containers – put instructions in them, and link them together to form the final program object code. Each codeblock can be attached to a symbol, as in the case of C function symbols; the linker will then be able to remove code associated to a symbol which is never referenced. We will attach codeblocks
to Alpha+ predicate symbols, but as it will become clear later, we will also use them for other tasks.

We will now examine the core of the Alpha+ parser by analyzing the inner workings of class and predicate parser functions in detail.

### 4.4.3 Parsing classes

The `_AlphaClassDeclaration` function deals with Alpha+ class declarations. The algorithm below briefly explains the steps performed to parse a class:

**Algorithm 4 Alpha+ class parsing**

1. detect whether we are dealing with an `extern` or a real class declaration
2. ensure the class identifier is valid
3. create new class type DAG node tagged as `TTypeNode_CLASS`
4. create new class symbol in current scope (which must be the global
   Alpha+ scope) for given class identifier, tagging it as `Ide_CLASS`.
5. create and enter new class scope
6. scan the whole class contents, building the class type nodes DAG and
   creating new symbols
7. for each non-virtual predicate symbol found at step 6, parse predicate
   and generate code for it
8. exit class scope

Steps 1-5 can be considered initialization steps; a class identifier is considered valid if it was not been previously used, except if it was used to declare and `extern` class and we are now declaring the real version of such class. Our goal, remembering what discussed in section 4.3.2 with particular reference to figure 4.6, is to save the type nodes DAG associated with the class, save the interesting symbols in the new class private scope and generate machine code for the non-virtual predicates found in it. This last goal forces us to divide the parse process in two passes, highlighted by steps 6 and 7 of algorithm 4; some instance or constructor predicate declared inside the class could in fact reference an instance variable of the class being processed, or call one of its predicates. For these to be successfully parsed, their symbol must have been previously encountered and declared into the symbol table, otherwise they would not be recognized. A one pass parse process would in fact generate an error while parsing a simple class as in listing 4.10.

In such an example in fact, the `getx` instance predicate tries to access the `x` instance variable for reading, but the `x` symbol has not been declared as the parser is on its first pass and has not encountered it yet – it will be declared
4. The compiler

```java
class A {
    getX(out result) {
        result = x;
    }
    body:
        int x;
}
```

Listing 4.10: Small class highlighting the need for a two-pass parse process

in the body section below. The problem is solved by skipping predicate body parsing on a first pass, and parse them all on a second pass, after the whole class has been scanned a first time.

The first pass is accomplished by the private `ClassDeclaration` function. This implements the pass as a very simple two-state machine; the state machine behavior is reported in table 4.2, execution starts in state $A$.

<table>
<thead>
<tr>
<th>State</th>
<th>Parsed element</th>
<th>Action</th>
<th>New state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>&quot;body:&quot;</td>
<td>start body section parsing</td>
<td>$B$</td>
</tr>
<tr>
<td>$A$</td>
<td>&quot;}&quot;</td>
<td>end of class found: exit state machine</td>
<td><code>End</code></td>
</tr>
<tr>
<td>$A$</td>
<td>[&quot;virtual&quot;] predicate</td>
<td>declare new predicate, skip predicate body</td>
<td>$A$</td>
</tr>
<tr>
<td>$B$</td>
<td>&quot;setup&quot;</td>
<td>declare new <code>#setup</code> symbol, skip setup body</td>
<td>$B$</td>
</tr>
<tr>
<td>$B$</td>
<td>&quot;cleanup&quot;</td>
<td>declare new <code>#cleanup</code> symbol, skip cleanup body</td>
<td>$B$</td>
</tr>
<tr>
<td>$B$</td>
<td>instance variable</td>
<td>declare new type node for instance variable and attach it to a class field node, increase class object instance context data size</td>
<td>$B$</td>
</tr>
<tr>
<td>$B$</td>
<td>predicate</td>
<td>declare new predicate, skip predicate body</td>
<td>$B$</td>
</tr>
<tr>
<td>$B$</td>
<td>&quot;}&quot;</td>
<td>end of class found: exit state machine</td>
<td><code>End</code></td>
</tr>
</tbody>
</table>

Table 4.2: Class declaration parsing state machine

In the case of an `extern` class, the parsing of `body:` causes an error, as `extern` classes are allowed to provide an interface only, with the consequence
that state $B$ can never be reached. Class setup and cleanup symbol names are mangled as \#setup and \#cleanup; this is enough as in the scope of a class only one occurrence of the setup and cleanup functions is allowed. Type nodes associated to their symbol are simple function type nodes accepting only one parameter, the context parameter. The context parameter is always implicitly passed as first hidden parameter to all kinds of predicates and it contains the memory address of the object instance context (see section 3.3.3) onto which the predicate is supposed to operate. Parsed instance variable declarations do not cause the creation of new symbols in the class scope; they cause however the creation of corresponding type nodes in the class types DAG. The size of the object instance data context is increased accordingly, as instance variables affect it, and the new size is saved in the class type node. Each predicate declaration, being it in the interface or in the body of the class, is handled by the \_PredicateDeclaration private function; its purpose is to create a new symbol of type Ide\_PREDICATE for the predicate, mangling its identifier accordingly, and to create and attach to it a function type node with the attached parameters list for the predicate. The identifier name is mangled only if the predicate is an instance or virtual one; in this case it assumes the form Class:Name, otherwise it stays unchanged. The symbol is declared in the global Alpha+ scope if the function is dealing with a virtual predicate prototype declared in the class interface, or in the class scope otherwise. The predicate parameters list is parsed using the \_ParameterList function already provided by the compiler module, but with some needed enhancements applied: Alpha+ predicate parameters lists in fact differ from standard C function parameters lists in that the former also support out parameters (see section 3.4.1) and embedded assignments (see section 3.4.2). The \_ParameterList function knows when to detect and allow Alpha+ special parameter features by checking if it is being called while using the Alpha+ parser; this is accomplished by using the new mInsideAlpha state variable of the CCCompiler component internal data structure. out parameters are handled by simply inserting a type node of kind TTypeNode\_REFERENCE between the TTypeNode\_PARAMETER node and the type node associated with the parameter itself. Embedded assignment are handled by saving the lexer position into the parameter type node, so the embedded expression code can be parsed at a later time; \_ParameterList then just skips the assignment and continues on the next parameter, if any.

Back to the \_PredicateDeclaration function, once it finishes building the predicate function type nodes DAG, it attaches it to a new field type node for the current class being parsed. It then saves the lexer position so the predicate body, if specified, can be later parsed, but at this time it just skips it and exits.
Once the first pass of the class declaration process accomplished by the _ClassDeclaration function ends, the class already has a dedicated scope, it has a full associated types node DAG and all of its predicate symbols have been declared. Newly created predicate symbols however have no associated codeblock yet; the second pass addresses this issue.

Looking at algorithm 4 again, step 7 is the next one, performing what we identified as our second pass. We declared symbols in the global Alpha+ scope for virtual predicate prototypes declared in the class interface; we are not going to parse their bodies however for a simple reason: virtual predicate prototypes are, as their name suggests, just prototypes, and the real predicates with associated bodies are to be declared outside of the class declaration. So it is perfectly fine if there is a virtual predicate symbol declared at global Alpha+ scope with no associated codeblock. For all other class symbols that are not virtual predicate prototypes, a predicate body (or an implicit one via the semicolon mark) is required though, and this is excepted to be parsed by the second pass, which returns error otherwise. The predicate body parsing is performed by calling the _PredicateDefinition private function; as it is used also to parse any predicate declared outside a class, we will discuss it in the next section.

4.4.4 Parsing predicates

Predicates can be declared both inside and outside a class declaration; it is then appropriate to use the same functionalities in both cases. We have seen how a predicate declared inside a class is constructed: a new symbol for the predicate identifier is declared, and the corresponding function type nodes DAG is built and attached to it. These steps are performed by the private function _PredicateDeclaration. In the case of instance and constructor predicates, we hinted the _PredicateDefinition function is called to parse the predicate body and generate a codeblock out of it to be later attached to the predicate symbol.

Predicates declared outside a class are handled exactly the same way, with only two main differences:

- The predicate function type nodes DAG is not attached to a field node of a class node. It only gets attached to the corresponding predicate identifier symbol node.

- The predicate body is always parsed for any kind of predicate.

The _AlphaPredicateDeclaration function deals with parsing predicates declared out of a class declaration. Algorithm 5 explains how this works.
Algorithm 5  Alpha+ predicate parsing
1) ensure a class supporting the predicate has been already previously declared
2) build predicate function type nodes DAG and create symbol node in current scope
3) parse predicate body and generate code for it

We will now discuss these three steps. The _AlphaPredicateDeclaration function partly performs step 1. At it is clear, step 2 is very similar to step 6 of algorithm 4, and in fact they are both implemented by the private _PredicateDeclaration function. In the case of predicates declared outside a class, this does extra checks to ensure the predicate is valid before constructing its type nodes DAG and creating the associated symbol, effectively completing step 1 started in the caller function _AlphaPredicateDeclaration. For example, suppose the following predicate is to be parsed, assuming class GRect was previously declared as seen in listing 3.18:

GRect(1).size(out w = 50, out h = 10);

_AlphaPredicateDeclaration finds the GRect identifier to be a valid symbol identifying the GRect class; this can be a prelude to a valid predicate declaration, so the grammar rule applies and the function can continue. Control is passed to private function _PredicateDeclaration, that skips the (1), but saves the lexer position of the Alpha+ object instance ID expression, in this case 1, so it can be later used. The size identifier is found; at this point the function scans the GRect class type nodes DAG for a field holding the predicate prototype. It finds it, meaning the class supports a predicate with such name.

As previously explained, _PredicateDeclaration now calls the augmented _ParameterList function to parse the given Alpha+ predicate parameters list, building an associated type nodes DAG. This DAG is then compared with the one found for the predicate prototype stored in the GRect class type nodes DAG; if the two DAGs match, it means the parameters lists match and the predicate declaration is valid. A new symbol of type Ide_PREDICATE with mangled name identifier GRect:size is created in the current scope; the symbol node mVal.mPredicate.mExprSourceIdx field is initialized with the object instance ID expression lexer position previously saved, and mVal.mPredicate.misAlphaScope with true as the predicate lives outside a class and thus in the global Alpha+ scope.

At this point _PredicateDeclaration has finished its work, and the _AlphaPredicateDeclaration function can resume and continue with step
3 of algorithm 5: code generation. We have seen predicates code generation is handled by the \texttt{_PredicateDefinition} private function, either if the predicate is declared inside a class declaration, either otherwise.

\section*{Predicate cases}

Before examining how code is generated for a predicate, it is recommended that we discuss how a particular design choice of Alpha+ affects the implementation. In chapter 3 we introduced the concept of predicate cases; for example in listing 4.11 we have 3 cases for predicate \texttt{GRect::size}:

\begin{verbatim}
GRect(1).size(out 100, out 25);
GRect(2).size(out 80, out 40);
GRect(3).size(out 35, out 70);
\end{verbatim}

\textbf{Listing 4.11:} Multiple predicate cases

We have seen that a predicate is basically a C function, though with a few differences. In the implementation, when a new predicate is parsed a new symbol is created and set up, then a new codeblock is created and associated to it. With multiple predicate cases however there is ambiguity: in the case of listing 4.11 for example, the same \texttt{GRect::size} predicate symbol is created three times, each time with an associated unique codeblock defining the particular predicate case (the first codeblock will assign 100 and 25 to the out parameters, the second 80 and 40, etc.); multiple predicate cases can be specified only for virtual predicates. Moreover, in our discussion so far, the object instance ID expression has no role yet.

The solution to solve the ambiguity is to effectively allow multiple symbols with the same name; the linker will detect all predicate symbols declared in the same scope with the same name, and it will merge them into an unique predicate. Technically, when a virtual predicate is called, only its specific case that is targeted by the caller is executed though; to solve this second problem, the parser inserts some glue code when generating the code for each predicate case. Virtual predicates are then internally made to accept two hidden parameters: the first is the usual \texttt{context} parameter that is already passed to all kind of predicates, the second is the expression that identifies the Alpha+ object instance the predicate is supposed to operate onto. The glue code the compiler adds to a virtual predicate case just ensures the expression passed in the hidden parameter equals the predicate case
object instance identifier expression; if true, the predicate case is executed, otherwise execution is skipped.

With these considerations in mind, the example we saw in listing 4.11 would then become the predicate function as in listing 4.12; note here we assume the predicate cases to be merged in an unique one, even if this only happens during the link process, and not at code generation time.

```c
bool GRect::size(void *ctx, int expr, out int w, out int h) {
    if (expr == 1) {
        w = 100; h = 25;
        succeed;
    }

    if (expr == 2) {
        w = 80; h = 40;
        succeed;
    }

    if (expr == 3) {
        w = 35; h = 70;
        succeed;
    }
}
```

Listing 4.12: Merged predicate cases in pseudo code

The implementation of generic virtual predicates (as seen in section 3.5.3) can benefit from this approach too: it is in fact sufficient that the hidden identifier expression parameter is made visible to the predicate function body; the `if` check can be avoided as it is left up to the programmer to execute certain portions of code depending on the value of the expression identifier with which the predicate was called.

**Code generation**

The private function `_PredicateDefinition` is used to parse a given predicate body and generate code out of it. The function assumes a new symbol for the predicate has been already created and set up, and it also assumes the predicate function type nodes to be constructed. It is called to parse and generate code for all kinds of predicates, declared both in and outside a class declaration, and is also used to generate code for the setup and cleanup
functions of a class, which for the scope are considered the same as predicates accepting no arguments. Its execution model is shown in algorithm 6.

**Algorithm 6 Predicate code generation**

1) create and enter new predicate function scope  
2) declare context hidden parameter in the predicate scope  
3) if generating a virtual predicate, declare expression parameter $E$ in [the predicate scope; if not a generic virtual predicate make it hidden  
4) declare parameters identifiers in the predicate scope  
5) if generating a virtual predicate and it is not a generic one, insert [check code to ensure $E$ equals the predicate case identifier  
6) insert code for any embedded **out** parameter assignment  
7) parse and generate code for predicate body  
8) exit predicate scope

Steps 2, 3 and 4 are used to declare particular symbols into the new predicate scope created at step 1. This is necessary because parameters for example need to be visible from inside the predicate body. The identifier expression $E$ that identifies the object instance onto which the virtual predicate is supposed to operate can be parsed before the predicate body; in fact we previously saved the lexer position before skipping the expression during the predicate signature parsing exactly for this purpose. In case we are dealing with a virtual predicate that is not a generic one, the expression code is generated into a codeblock using the *Expression* function of the C parser, then this codeblock is attached in a special glue code inserted into the final predicate codeblock. Listing 4.13 reports the portion of code performing this task.

```c
... 
theEndOfPredicateLabel = CCodeBlock_NewLabel(mCurrentCode_);  
if (theExprCode) { 
    CCodeBlock_sym(mCurrentCode_, theId);  
    CCodeBlock_ldn4(mCurrentCode_);  
    CCodeBlock_AppendCode(mCurrentCode_, theExprCode);  
    CCodeBlock_eq(mCurrentCode_);  
    CCodeBlock_jz(mCurrentCode_, theEndOfPredicateLabel); 
}  
... 
```

---

**Listing 4.13:** Predicate ID expression check code
mCurrentCode represents the current predicate codeblock, theId is the symbol associated with the expression ID hidden parameter and theExprCode is the codeblock previously generated containing the expression associated with the predicate case. theEndOfPredicateLabel is an unique new label that is used to control execution flow: if the comparision between the hidden expression parameter and the expression value associated with the predicate results in a difference, a jump to such a label is executed. The label will be later placed at the end of the generated predicate case code.

Step 6 is accomplished thanks to the use of the Val.Parameter.CodeIdx field we added to the type node structure; it is in such field in fact that the _ParameterList function previously stored the lexer position for the out parameter embedded assignment, if performed, as we discussed in section 4.4.3. The _PredicateDefinition function just steps over all predicate parameter type nodes and if it finds one has an associated assigned expression, it parses it generating codeblock via the C parser _Initializer function. The assignments are serialized into the final predicate codeblock and are the first instructions that will be executed, providing execution passes the case ID test inserted in the previous step.

To generate the real predicate body codeblock in step 7, we again rely on the C parser functionalities, by calling the _StatementList function. This is already used to parse and generate code for blocks of C statements, including function blocks, so it almost fulfills our needs, since Alpha+ predicate bodies are almost the same as C function bodies. There are differences though, and these were managed by adding checks in the relevant places in the C parser to ensure they are respected. These modifications cover functions _ArgumentExpressionList, _PrimaryExpression and a few minor ones; to the former was added the capability to automatically pass an argument by reference if the corresponding parameter in a predicate parameters list is an out parameter. The latter was modified to recognize class instance variables and predicate calls, and to automatically dereference out parameters.

These modifications however are not enough to fully satisfy what discussed in section 3.4.4: to support special constructs specific to the Alpha+ language like succeed, fail and foreach inside a predicate body, a new function _AlphaStatement was added, implementing the grammar rules associated with such constructs. We will discuss how these were implemented in the next section.

4.4.5 succeed, fail and foreach

Before discussing how succeed, fail and foreach were implemented, it is better to examine how these are supposed to behave from a technical point of
4. The compiler

view. Since these instructions basically deal with predicate return values and iteration via repeated calls to a predicate, we have to discuss how a predicate stack frame works, and as a predicate is technically implemented the same way as a C function, we will introduce the way the VM handles functions and their stack frames, to later extend the concept to Alpha+ predicates.

The Leonardo VM provides five registers used by the virtual CPU for the purposes we showed in table 4.1. A function is called following the steps highlighted in algorithm 7:

**Algorithm 7 Function call**

1) the function symbol is pushed on the stack
2) the stack is expanded to make room for function return result, if any
3) all arguments are pushed on the stack
4) issue a jsr n instruction, where n is the offset into the stack where [ to get the address to jump to; executes jump
5) shrink stack to remove the space occupied by function symbol and [ its arguments

We will discuss this algorithm with reference to the example in figure 4.8, which shows how a small C function `average` is called.

Figure 4.8b shows a portion of the object code generated for the `main` function of figure 4.8a, specifically the code that executes the call to function `average`. Step 1 is performed by the `sym @average` assembler instruction; this is a meta instruction that will be later replaced by the linker, when address binding is performed and the real code address of function symbol `average` is determined. Figure 4.8c shows the bound object code. Function `average` returns an `int`, which takes 4 bytes of memory, so step 2 is achieved by the `expn4` instruction which expands the stack\(^1\). The next step is achieved by the two subsequent `pushn4` instructions; at this point the SP register holds the value 40, and the `jsr` instruction specifies to offset this by 16 to get the address of the function to be called: 40 – 16 = 24 so the address where to jump is contained at stack address 24, which holds 182 (supposed address of function `average` determined at link time). Execution branches to address 182, and when the function returns the stack is supposed to be as it was before the branch. Step 5 of algorithm 7 is performed: the `shrns8` cleans the stack of the two parameters, subtracting 8 from SP. The topmost stack value is now the function result we previously reserved space for; we still have to remove the function `average` address which is placed below the result, without removing the result itself. The Leonardo VM supports the

\(^1\)The contents of the newly expanded stack region is unknown, but in our example we assumed it to be 0. Stack grows upwards.
special rmvsn4 instruction that removes the space occupied by a symbol address (4 bytes) assuming it is placed 4 bytes below SP. Once this instruction executes, the function call is terminated and the stack contains the function result at its top. We will now examine what is the exact behavior of the jsr instruction. When jsr n is issued, it builds a new stack frame for the called function and branches to it. The following actions are performed in order:

\[
\begin{align*}
(SP) & \leftarrow PC + 4 \\
(SP + 4) & \leftarrow R1 \\
(SP + 8) & \leftarrow R2 \\
SP & \leftarrow SP + 12 \\
R2 & \leftarrow SP \\
R1 & \leftarrow SP - n - 4 \\
PC & \leftarrow (SP - n)
\end{align*}
\]
At exit, a function is supposed to restore the same stack layout as found at function entry. In its body however, the stack can expand or shrink at will; usually the first action a function performs on entry is to expand the stack to reserve space for local variables; the R2 register never changes during the execution of the function, and this enforces its supposed purpose as explained in table 4.1.

A function returns when the \textit{rts} instruction is executed; not surprisingly, this just restores the previous stack frame by performing the following actions:

\[
\begin{align*}
R2 & \leftarrow (SP - 4) \\
R1 & \leftarrow (SP - 8) \\
PC & \leftarrow (SP - 12) \\
SP & \leftarrow SP - 12
\end{align*}
\]

At first sight, it may seem this same technique could be applied to predicates calling. \textit{succeed} and \textit{fail} would just put a 1 or 0 on the top of the stack and cause an \textit{rts}. This however would actually only work if the \textit{foreach} statement was not conceived. In our design, if a predicate is called in the context of a \textit{foreach}, when it issues a \textit{succeed foreach} executes a block of statements in the caller predicate, then iterates and execution resumes next to the last \textit{succeed} in the called predicate. The procedure continues until the called predicate ends with a \textit{fail}, in which case execution continues in the caller predicate, next to the \textit{foreach} block. If a predicate is called \textit{not} in the context of a \textit{foreach} though, \textit{succeed} and \textit{fail} just cause the predicate to exit with a true or false result on top of the stack.

The double behavior of \textit{succeed} and \textit{fail} depending upon the context in which the predicate they are contained in is called, does not allow them to be implemented with the instructions provided by the Leonardo virtual CPU and the \texttt{CCodeBlock} component; to avoid adding new ones, we decided to use \textit{traps}\footnote{the \texttt{trap} virtual CPU instruction causes the execution of custom C functions in the VM, which have complete control over the virtual CPU state.} instead. Our main problem lies in the fact that when a predicate exits with \textit{succeed}, the current stack layout must be preserved, together with the activation record (R1, R2 and PC), so that if the predicate was called within a \textit{foreach}, it can be later re-entered next to the \textit{succeed}, also preserving any computation previously done in the predicate.

Our solution to the \textit{foreach} problem is shown in figure 4.9; in such a figure we present how a small predicate performing a \textit{foreach} statement is compiled. The example addresses all the situations of \textit{succeed} and \textit{fail} inside or outside a \textit{foreach}, and also shows how \textit{break} and \textit{continue} can
be used inside `foreach` statement blocks, behaving in the same fashion as within C `for` statement blocks.

Examining the predicate in figure 4.9a and the corresponding assembled codeblock in figure 4.9b-4.9c, we can do some considerations:

- There are three new instructions we introduced with this example: `ssp`, `rsp` and `bjsr`. We will examine these later, suffice to say for now that they will be implemented as traps and that they will be the only new instructions needed to manage all `succeed`, `fail` and `foreach` statements.

- Next to the push of the `callee` symbol on the stack, the `#ctx` symbol is pushed. This is the hidden context parameter all predicate require as first hidden parameter.

- After pushing the arguments to call `callee`, an additional 0 is pushed on the stack.
• **succeed** is always translated to just one instruction: **bjsr**.

• **fail** is always translated to a **jmp**, but if called within a **foreach** block the **rsp** instruction is executed first.

• At the end of **foreach** statements block, what triggers the re-execution of the **callee** predicate is an **rts** instruction.

• **break** inside the **foreach** causes a **jmp** to the instruction next to **rts**, which is an **rsp**.

• **continue** inside the **foreach** causes a **jmp** to the **rts** instruction, which in fact re-enters **callee** execution.

It may seem strange that what triggers the re-enter into the **callee** predicate next to the last **succeed** issued by it is an usual **rts** instruction. What makes this work is in fact the mechanism of the new **bjsr** instruction implementing the **succeed** statement, which performs the following actions:

\[
\begin{align*}
(SP) & \leftarrow PC + 4 \\
(SP + 4) & \leftarrow R1 \\
(SP + 8) & \leftarrow R2 \\
(SP + 12) & \leftarrow 1 \\
PC & \leftarrow (R2 - 12) \\
R1 & \leftarrow (R2 - 8) \\
R2 & \leftarrow (R2 - 4) \\
SP & \leftarrow SP + 16 \\
\end{align*}
\]

**bjsr** works almost like a normal **jsr**, but with a few differences. First, next to the activation record, on the stack is put the value 1; this will be used as return value as **succeed** is supposed to return true. The second difference is that, instead of branching to a function whose address was just put on the stack for this purpose like **jsr** does, **bjsr** branches back to the function that called the current function. This is possible as **bjsr** takes the address of the caller from the current function activation record, accessed via the R2 register. This register in fact always points to the local variables space for current function, which resides right next to the activation record in the stack frame. The **backward jump** performed (hence the mnemonic **bjsr**) does preserve the stack frame of the predicate issuing the **succeed**; the mechanism is explained in figure 4.10, where **R1**, **R2**, **SP** and **PC** are the registers before **bjsr** is executed while inside the **callee** predicate; **R1'**, **R2'**, **SP'** an **PC'** are the registers after **bjsr** is executed. Assume **callee**
Figure 4.10: Stack layout before and after a \texttt{bjsr} instruction is executed

was called by the \texttt{caller} predicate in a foreach as in figure 4.9a, and that the \texttt{bjsr} instruction issued by \texttt{callee} as \texttt{succeed} is located at address 486.

When \texttt{bjsr} is executed, the R1 and R2 registers point back to the parameters and local variables memory areas of the \texttt{caller} stack frame; the \texttt{callee} stack frame however is not destroyed, and SP is now located beyond it, effectively opening a new part of the \texttt{caller} stack frame; the \texttt{callee} stack frame is now \textit{embedded} into the \texttt{caller} stack frame. At the topmost of the stack \texttt{bjsr} has placed a 1, meaning the predicate exited with a \texttt{succeed}. Execution now resumes after the \texttt{jsr} in the foreach that caused \texttt{callee} to be executed; as reported in figure 4.9b, a \texttt{jz} checks if the predicate return result on top of the stack is true or false; in our case it is true and execution continues – note that \texttt{jz} consumes the topmost stack value, in our case 1. Instructions in the address range 240-268 constitute the foreach statements.
block; the next instruction at label L1 is the rts that we hinted causes the re-execution of the callee predicate from the instruction next to its last succeed issued. And this now should make sense: rts basically restores a stack frame, and it takes the PC, R1 and R2 values from the top of the stack, where jsr placed them. In our case, there are actually the information to restore a stack frame on the top of the stack, but these were placed there by bjsr, and describe the stack frame of the callee predicate function, with the saved PC register properly set to the instruction next to the bjsr that built this data; rts reads it and causes the re-entry into callee next to succeed.

If the callee predicate issues a fail, it is assumed 0 is returned on the stack; in this case the jz would cause a branch to address 268, where an instruction rsp 0 is executed. This ends the foreach statement handling; the next instructions describe whatever other statements are present in the caller predicate up to its end. We said a fail statement is compiled into either an rsp and a jmp, either only into a jmp, depending on if the fail is issued inside a foreach statements block or not. The jmp always causes a branch to the end of the predicate function; note that the unbound predicate object code in figure 4.9b ends with the last instruction at address 300, while the bound version has an additional rts instruction attached. This addition is performed by the linker, which first merges all the predicate cases (as discussed at page 103) and then adds an rts at the end of the merged predicate function codeblock. The execution of rts on fail issued inside callee causes the return to the jz instruction in caller, but this alone cannot guarantee that the topmost stack value is 0, signaling false. This is why before issuing the jsr to call the callee predicate function, and after pushing its parameters, we pushed a 0 into the stack. When our rts is executed, this 0 effectively is the topmost stack element, and jz can branch, ending the foreach block.

When the foreach block ends however, the stack is filled with now useless data: there are the callee address and parameters, plus a series of embedded callee stack frames, one for each succeed the callee predicate function issued. This waste of stack space occurs every time a foreach statement is executed, so is unacceptable. There is also no mean to know the amount of consumed space at compile time because we do not know how many times the callee predicate issues a succeed. To avoid this problem, the ssp and rsp instructions were added: these are used to save and restore the SP register contents respectively. The register is saved into the local variables space for the caller predicate, where space for an hidden variable is allocated before the space for the normal local variables. Saving the SP contents before the foreach and restoring it afterwards efficiently cleans the
4. The compiler

stack.

An additional problem arises however if two or more foreach statements are nested, like in listing 4.14. In this case the hidden local variable where SP is saved is overwritten, causing the old value to be lost. This has the result that only the innermost foreach leaves a clean stack.

```c
caller () {
    foreach (...) {
        foreach (...) {
            foreach (...) {
                foreach (...) {
                    ...
                }
            }
        }
    }
}
```

Listing 4.14: Nested foreach

The problem is easily solved by allocating space for as much hidden variables where to save SP as the maximum foreach nesting level in the predicate; in the case of listing 4.14 for example, space for three variables would be needed. Each foreach will use a variable chosen depending on the foreach nesting level; this explains why our rsp and ssp instructions are accepting a parameter, as this is used to select the hidden variable where to store SP. Our hidden variables are allocated as first local variables for the predicate; considering R2 always points to the predicate local variables space address, this allows us to implement the ssp n instruction as performing just the simple action:

\[(R2 + n) \leftarrow SP\]  \hspace{1cm} (ssp)

The rsp n instruction performs the opposite action:

\[SP \leftarrow (R2 + n)\]  \hspace{1cm} (rsp)

bjsr, ssp and rsp are all implemented as traps. This decision was made because traps grant complete control over the virtual CPU, and can be more easily debugged compared to real virtual CPU instructions implemented into the VM fetch-execute loop.
4.4.6 Predicate calls

We have examined how a predicate can be called as argument of a **foreach** statement, and how **succeed** and **fail** correctly behave in the called predicate in this context. But we need to ensure that if the same predicate is called in a normal expression, and not as **foreach** argument, **succeed** and **fail** must return true and false to the expression. Indeed the way we handle these statements require special treatment for this case. Figure 4.11 shows a case of a simple predicate call, and how this gets compiled.

![Figure 4.11: Standard predicate call example. (a) sample predicate; (b) corresponding unbound object code; (c) bound object code; (d) stack layout on fail after jsr; (e) stack layout on succeed after jsr; (f) stack layout before clns](image-url)
As expected, we have to push a 0 on the stack after pushing the parameters and before issuing the \texttt{jsr}, to accommodate the case in which \texttt{callee} exits with a \texttt{fail}. If \texttt{callee} exits with a \texttt{fail}, the stack layout will be as shown in figure 4.11d when execution resumes right next to the \texttt{jsr} instruction; if \texttt{callee} exits with a \texttt{succeed}, the stack layout will be as in 4.11e. We need to operate in such a way that the stack contains just either 0 or 1 at the end of the call; in our example, instructions in the address range 608-632 check if the topmost stack value is 1, and in such case, the 0 previously pushed before issuing the \texttt{jsr} is removed from the stack, replaced by 1. At this point the stack has the same layout as in figure 4.11f, where \texttt{x} can be either 0 or 1. The final step is to remove the predicate address and parameters from the stack below the result, so only the result remains. Our first thought could go to the \texttt{rmvs} instruction we previously encountered; though similar in concept, this cannot be used here however. \texttt{rmvs n} just removes the space of a symbol address located at \texttt{SP} – \texttt{n} and shifts the function result of size \texttt{n} down the stack to fill the void. We need an instruction that operates the other way around: we need to remove a block of stack elements that spans in the range \((SP – 8) – (SP – 8 – n)\) and shift the contents of \((SP – 4)\) to fill the void. This is the purpose of the new \texttt{clns} instruction we again implemented as a trap. Once this finishes execution, the predicate call terminates, leaving 0 or 1 on top of the stack.

Note from figure 4.11b that the procedure used requires the \texttt{ssp} and \texttt{rsp} instructions; this means that if a predicate performs at least once a predicate call, even though it never uses \texttt{foreach}, an hidden variable will be allocated in the local variables memory area of the predicate stack frame.

There is still one issue to discuss about predicate calls: so far we have used as example a simple \texttt{callee} predicate, assuming this to be an instance predicate declared inside a class interface. Predicate calls can be distinguished in four categories however:

1. Call to a virtual predicate of an explicit object instance.
   Example: \texttt{GRect(1).size(width, height)};

2. Call to an instance predicate of an explicit object instance.
   Example: \texttt{GRect(1).getSize(width, height)};

3. Call to a virtual predicate of the current object instance of the caller.
   Example: \texttt{size(width, height)};

4. Call to an instance predicate of the current object instance of the caller.
   Example: \texttt{getSize(width, height)};
While the bulk of a predicate call, consisting in the push of the parameters on the stack, the jsr and the mechanisms described earlier in this section as well as while discussing foreach are always implemented the same way, each of these four cases require special handling for the hidden context parameter. Also remember that virtual predicates also require a second hidden expression parameter explicitly identifying the predicate object instance for which the predicate is being called; virtual predicates in fact check this parameter in the glue code inserted at compile time to select the predicate case to be executed, as we discussed in section 4.4.4. Instance predicates do not require this second hidden parameter though.

Case 1 has no special requirements. As virtual predicates cannot access object instance data, it is safe to pass them any context as first parameter; we therefore pass the context of the caller. The expression parameter is supplied by the call itself; in the example above, 1 is passed.

Case 2 only requires to pass the context hidden parameter as it is dealing with an instance predicate. It requires to perform a lookup to find the correct object instance context to be passed though; the lookup must ensure there is an object of the specified class in the same scope of the caller object, with instance identifier equal to the supplied expression. If such an instance object exists, the lookup returns the memory address of its context, otherwise the whole predicate call cannot be performed as the target object does not exist; in this case the predicate call just returns false as if the callee issued a fail.

Case 3 also requires a lookup. We said that the context parameter is not important when calling a virtual predicate, but in this case we are not looking up the context, rather the address of the predicate function to be called. When calling a virtual predicate without specifying a target object in fact, the correct predicate function to be called depends on the scope where the caller object is declared; if an object is declared within an object instance of class A, and one of its predicates issue a call to a virtual predicate of the object like in case 3, the lookup searches for the predicate in the context of the parent object of class A. As the lookup performed in case 2, if this lookup fails the predicate call returns false as if fail was issued. Note that it is this kind of lookup that allows for the Alpha+ polymorphism we discussed in section 3.5.5.

Case 4 is handled similarly to case 1, except that in this case we only pass the context received by the caller predicate right to the callee predicate.

The two lookup kinds discussed for cases 2 and 3 are both implemented as traps inside the VM, and as we will see they both make use of hash tables to quickly perform their job. To work though, they require by the linker the setup of special data structures, named class records, we will discuss in section 4.5.2.
4. The compiler

4.5 The Alpha+ linker

When the parser finishes its task, we have the following output:

1. A series of class and predicate symbol nodes in the symbol table: each class has an associated symbol of type `Ide_CLASS`, whereas each predicate has type `Ide_PREDICATE`, and the symbol node describes the class or predicate it handles. Multiple predicate cases each have a separated symbol with the same name.

2. A series of type nodes DAGs attached to the symbol nodes above, further describing them.

3. A number of new scope nodes, each describing the scope of a class or a predicate, with reference to the symbols declared within it. Among these scope nodes, there is always one unique scope that is marked as special and describes the global Alpha+ scope.

4. A pool of codeblocks: each codeblock is attached to at most one predicate symbol and contains the generated code for that predicate. Not all predicate symbols have an attached codeblock.

Role of the linker is to get this data as input and output an executable unit, by applying a series of operations among which binding symbols. The most important task the Alpha+ linker accomplishes compared to the tasks usually performed by a traditional linker, is the addition to the final executable unit of a glue code that starts the Alpha+ program, performing the functions of algorithm 2. The glue code will install reactive callbacks to perform the task, and their target function, implemented inside the VM, will also keep track of live objects, destroying objects that are not needed anymore or initializing new ones each time the live context tree changes due to a reaction. This effectively implements the operations of both algorithms 2 and 3.

4.5.1 The link process

For normal C programs, the Leonardo C compiler calls the CLinker component to perform the linkage. Since Alpha+ programs can be embedded into C programs, it makes sense to use the same approach, and let the CLinker component detect if handling Alpha+ object code, and in such case pass control over a dedicated Alpha+ linker subsystem. We implemented such subsystem in the CLinker_Alpha.c source file.
Linking is a sequential process during which the operations described in algorithm 8 are performed. Bear in mind the linker actually receives as input a list of object units; each object unit contains the output of the parsing of a single source file, in the form of the data as discussed above.

**Algorithm 8 Standard C linking process**

1. build linkage chains
2. create new empty target executable unit codeblock \( T \)
3. find start function symbol \( S \) as this is the executable unit entry point 
   \[ \text{and append its codeblock to } T \]
4. mark all functions reachable from \( S \) and append their codeblocks to \( T \)
5. create program entry code that calls \( S \) and append it to \( T \)
6. bind symbols in \( T \)

We will discuss these steps before extending the algorithm to support Alpha+. An executable unit is basically just a container for a target codeblock that contains the bound code for all the program.

Step 1 builds linkage chains. A linkage chain is a linked list of symbol nodes, possibly defined in different object units passed to the linker, all referencing the same memory object. For each linkage chain, an associated target symbol is created, and if the symbol refers to a data object, an unique memory address is associated to it.

During steps 2 a new empty codeblock is created and attached to the executable unit. This codeblock will contain all the program instructions, as all the reachable codeblocks will be appended to it; it is this codeblock the linker works on during the whole link process.

In step 3 the linker looks for the start function symbol; for C programs this is usually the `main` function symbol. If the linker cannot find this, the `main` function is missing and an error is issued. When the `main` function symbol is found, its associated codeblock is appended to the target executable unit codeblock.

Step 4 scans the codeblock attached to the main function symbol to find all the referenced function symbols, and attach their codeblocks to the target codeblock. Then it repeats the process scanning these newly added codeblocks for referenced function symbols, and keeps adding codeblocks until all reachable codeblocks are added. This ensures codeblocks attached to unreachable functions in the C program are simply discarded from the target executable unit codeblock. While scanning a codeblock appended to the target codeblock, all symbols encountered in it are also replaced by a target symbol by looking up the associated linkage chain as created in step 1.
Step 5 is needed to add a glue code that lets the VM execute the user `main` function code upon startup of the executable unit.

Symbols binding is performed in step 6: in this process replaces all `sym` meta instructions in the target codeblock with the addresses attached to the corresponding target symbol nodes. At the end of this step, the target codeblock will have all memory references in their final form, and the executable unit will contain an unique large program codeblock ready for execution by the VM.

4.5.2 Class records

To support Alpha+ in the linker, we must first introduce the concept of class record. A class record is a data structure built at link time, holding information on each class used by the program being linked; class records are stored in the static initialized data section, that basically is a memory block attached to the executable unit, with which a program data is initialized at startup.

![Figure 4.12: Class record layout. (a) standard class record; (b) global class record](image-url)

The layout of the class record structure is shown in figure 4.12a. A class record stores the addresses of the class setup and cleanup functions and the size of the object instance data; this will later be used by the VM to know how much space to allocate in memory every time a new object instance of this
class is created. Also, the class record stores the address of a hash table that is used to keep track of all the global object instance contexts declared by the program and live at any time. More importantly, a class record contains a static hash table embedded into it; this contains information on all the virtual predicates of any class declared inside the class body, and is used by the predicate call lookup trap of case 3 as discussed in section 4.4.6. To each virtual predicate in fact, is assigned an unique key during the Alpha+ link process, and this is used as hash value. The static hash table has a size chosen at link time that is always the smallest prime number ensuring the number of virtual predicates declared in the class body can fit into the table. Upon hash collisions, the following simple algorithm is applied:

**Algorithm 9 Hash table collision solver**

1. let $N$ be the current hashed index
2. let $S$ be the hash table size (prime number)
3. let $O$ be the largest power of 2 number smaller than $S$
4. while there is a collision on the $N$-th hash table entry, do
   - let $N = (N + O) \mod S$

The algorithm is very fast and ensures all slots of the hash table to be visited in $S$ steps, with a granularity that depends on the choice of both $S$ and $O$.

Our virtual predicates hash table embedded into the class record has each of its entries composed of the predicate associated key and the corresponding predicate function address; if an entry is not used, it is marked as such.

The class record also contains the number, $m$, of sub-classes whose objects are declared within the class body; this is followed by $m$ records where each record contains the address of the class record associated to the $i$-th sub-class, followed by the address of its constructor predicate function. Constructors declared in the same scope are merged into an unique predicate, like for virtual predicates, hence the need for only one constructor function address.

Aside all class records, there is also a special global class record, which has the same layout of a normal class record, except for the fact it does not feature the first four elements. The purpose of the global class record is to store information on the virtual predicates and on the classes whose object instances are declared in the global Alpha+ scope; not describing the scope of a particular class, rather the global Alpha+ scope, the global class record has no knowledge of setup or cleanup functions, nor about object size.
4. The compiler

4.5.3 Extending the linker

We now have the basis to extend the link process to support Alpha+ linkage. Algorithm 8 can be restructured into the following Alpha+ aware algorithm:

Algorithm 10 *Generic linking process*

1) build linkage chains and predicate chains
2) create new empty target executable unit codeblock \( T \)
3) find start function symbol \( S \) as this is the executable unit entry point \[ and append its codeblock to \( T \) \]
4) mark all functions reachable from \( S \) and append their codeblocks to \( T \)
5) mark as reachable all predicates in global Alpha+ scope, the predicates \[ in their classes and, recursively, in the classes of sub-objects \]
6) append codeblocks of all reachable predicates to \( T \)
7) build class records
8) create Alpha+ glue code with initialization function \( A \)
9) create program entry code that calls \( A \) and then \( S \) and append it to \( T \)
10) bind symbols in \( T \)

Step 1 is similar to the one in algorithm 8, except for the fact that it also builds *predicate chains*. A predicate chain is a linked list of predicate symbols that are multiple cases of the same predicate that have been declared in the same scope.

Steps 5 and 6 are mostly performed by the \_StitchReachablePredicates\_ private function of the \_Clinker\_ component. This also merges all the predicates of a predicate chain into an unique predicate codeblock, that is then appended to the target executable unit codeblock.

Step 7 is performed by the \_BuildClassRecordsTable\_ function of the same component, and builds class records as seen in section 4.5.2 out of the information gathered during the previous steps.

Step 10 is almost unaltered compared to algorithm 8.

At this point the target executable unit contains all reachable predicates codeblocks and all class records, but it has no mean to perform any kind of Alpha+ code execution. Step 8 creates a codeblock holding a glue code whose purpose is to install a series of reactive callbacks that will trigger the creation of the objects declared in the global Alpha+ scope, and all their sub-objects. From there on, the reactive environment provided by the VM will manage the creation and destruction of Alpha+ objects automatically. Step 9 ensures the glue code gets called before the main function of the program being linked. In the next section we will discuss how the Alpha+ glue code behaves and gets created.
4. The compiler

4.5.4 Alpha+ glue code

As we have said, role of the glue code is to bridge the gap between the predicate functions and their execution in a reactive environment. For this task to be satisfied, all objects and their sub-objects need to be created, and any reaction chain must be established as well, to ensure the live contexts tree is kept up to date while certain predicate functions get called in reaction to specific events.

To perform this, the glue code needs to install a reactive callback on each constructor of a class declared in the global Alpha+ scope. Once called, this constructor (running as a reactive callback) needs to ask the VM to create a new context for each object identifier for which the constructor issued a `succeed`. The VM will schedule a new reactive callback for the class `setup` function, passing to it the newly created object instance context. It will also scan the object class for sub-object constructor declarations, scheduling a reactive callback for them too, repeating the process. The VM will also keep track of which new objects are created for a constructor, compared to the last time the constructor was called; objects that are not created the second time are not valid anymore and must be deleted. For these, the VM will schedule execution of the class `cleanup` function, passing the object context to it, and will then free the context memory. Being the constructor and setup function, as well as all predicates called from it, executed under the context of a reactive callback, the reactive callback system keeps track of all the memory referenced by them, establishing reaction chains that will trigger the re-execution of the whole process just described; once this occurs, the result is that the live contexts tree is effectively updated.

The class record structure was introduced to aid in this process. As a matter of facts, most of the objects management is done inside the VM for efficiency reasons, and the glue code interacts with these functionalities via traps; a pointer to a the class record will be passed around between the glue code and the VM to let them share information on a particular class, such as the `setup` and `clean` functions addresses, and the addresses of the constructors of any sub-object.

The glue code is composed of two functions: we will now examine the first, the `ObjectBuilder` function. This function gets as arguments a parent object context pointer \(P\), the address of the sub-objects hash table of the parent object \(H\), the address of a constructor \(C\) and a pointer to a class record \(R\). Each object context internally has an hash table that stores pointers to any sub-objects contexts; this is used to keep track of the live contexts tree. Objects declared in the global Alpha+ scope do not have a parent object; in this case \(P\) may be `NULL`. 
Algorithm 11 Function ObjectBuilder(P, H, C, R)
1) initialize an empty list of identifiers L
2) implement a foreach calling C; let I be the identifier successfully returned by C at each iteration, add I to L
3) issue the trap CreateAllObjects, passing to it P, H, R and the address A of function ObjectBuilder

The need to perform a foreach calling a constructor is the main reason this function is implemented in the glue code. The function is meant to just construct a list of identifiers to be considered alive when the constructor is called; it then passes control to the CreateAllObjects trap, that performs all the required operations to create the new object contexts and delete the now obsolete ones. CreateAllObjects will be discussed in the next chapter, suffice to say for now that it requires the address of the ObjectBuilder as it will schedule a new reactive callback on this function, to create sub-objects.

Algorithm 12 shows the second glue code function, AlphaInit. Purpose of this function is to create all the objects declared in the global Alpha+ scope; to perform this task, it schedules a reactive callback on the ObjectBuilder function for each class record, passing to it NULL as P, the address of the global objects hash table found in the class record as H, the address of the constructor for objects associated to the class record found during the link process as C, and the address of the class record itself as R.

Algorithm 12 Function AlphaInit
for each class record R do:
1) let C be the address of the constructor for objects of this class declared in global Alpha+ scope
2) create hash table to track global objects contexts and store its address T into the corresponding field of R
3) schedule reactive callback for ObjectBuilder(NULL, T, C, R)

Hash tables are extensively used internally to the VM to keep track of live contexts. An hash table tracking all constructed global objects of class A is attached to the class record for class A; each object context as we will see also has a series of hash tables attached, each specifying all the constructed sub-objects of a particular class contained in the class of the object. We will further discuss how the VM tracks all this data in the next chapter.

The linker forces the AlphaInit function to be called before the user main function; this ensures Alpha+ objects get created and the correct reaction chains to be established. Once glue code is generated and attached to the target executable unit codeblock, symbols binding occurs. After this, we
finally have a working executable unit containing compiled Alpha+ code the VM can run.

4.6 Conclusions

In this chapter we have examined the compiling process of Alpha+ programs, and how this occurs mainly in two steps: i) parsing and code generation, and ii) linking. During the development, the Leonardo C compiler has proven to be a valid ground onto which to build the Alpha+ compiler; its solid infrastructure has allowed us to extend it with relative ease for our needs. The declarative nature of Alpha+ and its special features have had to be broken down to an imperative language as is the assembly understood by the Leonardo Virtual Machine, and this has been challenging, requiring much efforts. In the next chapter we will look into expanding the Leonardo Virtual Machine itself, so it can handle executable units containing compiled Alpha+ code.
Chapter 5

The virtual machine

5.1 Introduction

The Alpha+ programming language assumes a reactive environment in which programs are executed; additionally the executable units produced by the compiler, as we have discussed in the previous chapter, assume the availability of certain runtime support functionalities. These factors make it impossible to support running Alpha+ programs directly on some kind of hardware, as this does not yet exist. Instead, an interpreted approach has to be undergone, as we previously discussed; a dedicated compiler parses Alpha+ source code and outputs an executable unit, containing assembly-like instructions for a virtual CPU in the form of bytecode. An associated virtual machine later interprets and executes this executable unit, producing the program output, as highlighted in figure 4.1. This is partly a forced choice mainly as no existing hardware can support reactive programming, but has the advantages that we have complete control over program execution flow, and that a virtual machine, if properly written, can execute virtual instructions rather quickly, with little speed losses hardly noticeable especially on modern computers.

The choice of the Leonardo Computing Environment as a basis for our work was not limited to the advantages it has effectively brought to the compiler development, but also as it provides a powerful, fast and modular virtual machine, which as expected has proven itself to be easily extendable for our purposes as much as the compiler. In this chapter we will discuss the extensions we developed for the LCE VM, which constitutes the second and final phase of the development process. By the end of the chapter, we will have a detailed knowledge about how the VM handles previously compiled Alpha+ programs, providing a fully reactive environment thanks to these extensions.
5.2 Virtual machine architecture

The VM uses an extremely modular design, whose architectural layout is shown in figure 5.1. The virtual CPU executes the virtual machine instructions as found in an executable unit, and for specific tasks it passes control to the traps system. The traps system calls the kernel, which divides its functionalities among several internal modules, named managers, which perform specialized tasks, possibly with side-effects on the internal virtual CPU state.

![Figure 5.1: VM architecture layout](image)

In this section we will discuss the various components of the VM architecture, to better understand where support for the Alpha+ programs execution has been added.

5.2.1 The virtual CPU

The virtual CPU is the core of the system; it is a general-purpose interpreter that executes programs written in an assembly-like language, as read from executable units previously loaded in memory. It is a stack-based execution machine with only five registers and its instruction set covers about 100 elementary control flow, data flow, and arithmetic/logic operations. We have already examined the purpose of the five registers in table 4.1; the program counter (PC) register keeps track of which instruction is to be executed at each virtual CPU cycle, and all operations are performed on the stack, thru the stack pointer (SP) register. R0, R1 and R2 are convenience registers used to store pointers to specific memory areas to access the global program data section, the parameters of a function in its stack frame, and the local
variables of a function again in its stack frame. Instructions and memory operations are always aligned to a 32bit boundary for efficiency reasons.

The virtual CPU fetch-execute loop is located in source file `CCPU_Inline.c`, in the `CCPU_Exec` function. As the file name suggests, instructions execution is performed by inlined code, guaranteeing an execution as fast as possible of the single instruction, which usually performs a very basic atomic operation.

The full state, or environment, of the virtual CPU including registers contents, used CPU heap memory and stack bounds, is stored in the `CCPU_TEnv` structure as reported in listing 5.1.

```
typedef struct {
    ui4 mPC;
    ui4 mSP;
    ui4 mR0;
    ui4 mR1;
    ui4 mR2;
    i1* mBase;
    i1* mLimit;
    ui4 mStackLimit;
    ui4 mUserData;
    ui4 mCodeBase;
    ui4 mStackBase;
} CCPU_TEnv;
```

**Listing 5.1:** The virtual CPU environment structure

In the VM, when an executable unit is loaded, a new memory block is created, which constitutes the memory associated to the program to be executed. This block, whose bounds are identified by the `mBase` and `mLimit` fields of the `CCPU_TEnv` structure, is then divided in several section associated to the program code, the global data and the work stack. Addresses referenced in the program code are always relative to this block, which automatically grows as the stack requires more space.

The advantage of this system is that it allows for a multiprogrammed environment: the VM can handle several running processes, each of which associated to a virtual CPU state comprising the state of registers and the program memory as discussed above. In this view, the memory block associated to a running process can be seen as the process address space in OS design terms, and like its OS counterpart, it is completely separated from the memory blocks of other running processes. The Leonardo VM, with its virtual CPU and as we will see with the functionalities provided by the ker-
5. The virtual machine

The virtual machine, indeed emulates an unique environment that can be seen as an OS being able to execute several processes at once.

5.2.2 The traps system

The virtual CPU is very basic by design and it limits itself to the execution of very simple arithmetical and logical operations. Higher level functionalities are provided by the kernel, which is accessed by the virtual CPU thru the traps system; traps basically act as an interface layer between them.

The traps are implemented as a function lookup table, where each slot can be associated to a trap to be executed in a certain situation. A trap handler function is coded in C inside the kernel, and has the following form:

\[
\text{typedef void (*CCPU_THandler)(CCPU_TEnv** thruEnv, ...);}]

Trap handlers receive a pointer to the virtual CPU environment, so they are free to modify in any way the internal virtual CPU state of the process that issued the trap. There are two kinds of traps: CPU traps and kernel traps, similar in concept respectively to interrupts and system calls in OS design terms; we will now discuss their differences.

- **CPU traps.** These are traps automatically invoked by the virtual CPU when specific events occur. When the VM is started, the kernel installs an handler for each of these traps to manage the related events; supported events are listed in table 5.1. Default CPU trap handlers can be overridden to supply user-defined actions, and this is especially useful for advanced applications and debuggers running inside the VM.

- **Kernel traps.** Contrary to the CPU traps, these traps are not automatically invoked upon certain events; kernel traps are directly invoked by explicit virtual CPU assembly instructions. These traps are used to access the functionalities provided by the VM kernel managers from the user program transparently. The program source code just calls C functions declared in a special way, so that the compiler translates them to trap calls. Upon execution, the virtual CPU fetches the trap instruction and executes the associated trap handler function. Kernel traps are used to provide otherwise inaccessible functionalities to the user programs, like VM process handling, memory handling, I/O handling, GUI handling, and more. Many kernel traps use the Leonardo Library to provide system-independent behaviors.
5. The virtual machine

<table>
<thead>
<tr>
<th>Trap ID</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCPU_ILLEGAL_TRAP</td>
<td>Unsupported trap invoked</td>
</tr>
<tr>
<td>CCPU_ILLEGAL_INSTRUCTION</td>
<td>Illegal virtual CPU instruction fetched</td>
</tr>
<tr>
<td>CCPU_MEMORY_PROTECTION_FAULT</td>
<td>Program performed an illegal memory access</td>
</tr>
<tr>
<td>CCPU_DIVIDE_BY_ZERO</td>
<td>Division by zero arithmetic exception</td>
</tr>
<tr>
<td>CCPU_ENTRY</td>
<td>Execution entered new routine</td>
</tr>
<tr>
<td>CCPU_HALT</td>
<td>halt end of program instruction executed</td>
</tr>
<tr>
<td>CCPU_LEAVE</td>
<td>Software breakpoint</td>
</tr>
<tr>
<td>CCPU_JSR</td>
<td>jsr instruction executed</td>
</tr>
<tr>
<td>CCPU_RTS</td>
<td>rts instruction executed</td>
</tr>
<tr>
<td>CCPU_MEM_WRITE</td>
<td>Process wrote data to memory</td>
</tr>
<tr>
<td>CCPU_MEM_READ</td>
<td>Process read data from memory</td>
</tr>
</tbody>
</table>

Table 5.1: CPU traps

5.2.3 Kernel and managers

All trap function handlers are implemented in the VM kernel. The kernel has a modular design where each module manages a specific range of traps inherent to the same topic. These modules are called managers and, with reference to figure 5.1, they comprise a process manager, a memory manager, a log manager, an event manager, a file I/O manager, a GUI manager and an utilities manager; we will now examine all of them.

**Process manager** This manager provides kernel trap handlers to create and destroy processes, start and stop them, get the current process ID, the arguments that were passed to it when it was launched, and generally other process-related functionalities. Internally it is this manager that schedules processes for execution by the virtual CPU. Other than kernel traps, this manager also provides an handler for all of the CPU traps, with the notable exclusion of the CCPU_MEM_WRITE and CCPU_MEM_READ traps.

**Memory manager** The memory manager handles kernel traps inherent to memory allocation, deallocation, copying and moving, corresponding to the C functions malloc, free, memcpy and memmove; all memory management occurs within the same process memory block. The allocator
uses a simple first-fit strategy, which has proven to be a good compromise between reduced memory allocation granularity and efficiency. It eventually enlarges the process memory block if not enough contiguous memory is available upon request. Process memory is word-aligned, and for each word of all the memory block, the manager provides a bitmask storing access rights informations on that specific word. This manager also handles the `CCPU_MEM_WRITE` and `CCPU_MEM_READ` CPU traps to ensure memory accesses on specific addresses are valid, and that the user program has the rights to perform the read/write action by examining the corresponding word bitmask.

**Log manager** The log manager keeps track of all the memory changes performed by a process, and provides kernel traps to start and stop monitoring it, and to eventually roll-back specific changes. This manager closely interacts with the memory manager to suit its needs, and its functionalities are extremely useful to develop debuggers running inside the VM.

**Event manager** An *event* is an action performed by some entity (which can be the user program as well as the system in response to an external action, like the click of the mouse button on a GUI element) that triggers the execution of specific user program routines previously installed by the program as *listeners* to that event. This manager provides kernel trap handlers to add and remove listeners, and to raise particular events. Internally the event manager works in conjunction with the process manager and the virtual CPU to provide events triggering in the form of interrupts; these are dispatched during the fetch-execute loop of the virtual CPU, but do not allow for reentrance: new interrupts generated while inside an interrupt and simply enqueued for later execution.

**File I/O manager** As the name suggests, this manager handles kernel traps dealing with file I/O so that the user program can create, delete, read and write files in the host filesystem.

**GUI manager** The GUI manager is a layer on top of the Leonardo Library multiplatform GUI management routines, providing kernel traps to allow user programs to create and manage GUI elements. GUIs are event-driven, and this manager uses the functionalities of the event manager to dispatch events due to user interaction with the GUIs created by the program.
Utilities manager  This special manager does not provide any functionality by itself, but rather delegates sub-managers to provide support for ranges of traps dealing with specialized tasks not inherent to the main managers discussed thus far. The utilities manager include:

**System support**  Sub-manager that as by now provides kernel traps to print messages to a console window and to retrieve the desktop dimensions. It will probably expanded in the future for more system-wide functionalities.

**Code generation support**  Provides kernel traps to interface with the compiler itself, which is integrated into the VM application. Having the compiler accessible from user programs via traps can be easily used to create integrated development environments running inside the VM, which coupled with a proper debugger as previously hinted can make the Leonardo platform a powerful integrated development platform.

**Math support**  Contains kernel traps to support common mathematical function for the user programs. Implements as traps most of the functions as found in the ANSI-C math.h header file.

**Strings support**  Similar to the math sub-manager, but for strings. Implements as traps most of the functions as found in the ANSI-C string.h header file.

**Time support**  Provides kernel traps to implement functions dealing with time management. Currently only allows to retrieve the value of a high-precision timer and to get the current date, but will eventually be expanded in the future to provide support for all the functions found in the ANSI-C time.h header file.

**Longjump support**  Implements kernel traps to save the virtual CPU registers state and to later restore them, effectively allowing to restore old CPU contexts. They are used to emulate the long jump interface as found in the ANSI-C l g j m p . h header file.

A desirable goal for the utilities manager with its sub-managers is to provide the backend support for a full C runtime library implementation to be used by the user programs running inside the VM; the code-base is rather young though, and very few functionalities are available at the moment, so this is still a long-term goal.
5.2.4 Extending the VM for Alpha+ support

The most natural step to integrate Alpha+ support in the architecture discussed above is to add two new managers. The first manager will be dedicated to support Alpha+ related kernel traps; while developing the stage compiler, we often used traps to support specific features, assuming these to do the bulk of the work. In the *Alpha+ manager* we will put the implementation of the related trap handlers. The second manager we are going to introduce is the reactive callbacks manager, or *RC manager* for short; this manager will be dedicated to handle reactive callbacks creation, memory monitoring and automatic triggering. To perform its task, it will work closely in conjunction with the memory manager. The new VM architecture layout with the aforementioned additions is displayed in figure 5.2.

![Figure 5.2: VM architecture layout with Alpha+ support](image)

We have decided to consider the Alpha+ manager as a sub-manager of the more general utilities manager, as its trap handlers are specific to the Alpha+ language, and do not interact much with other managers, with the exclusion of the RC manager. The RC manager is considered a main subsystem instead, as it interacts with the VM state at a lower level, hooking the memory manager. The RC manager is independent from the Alpha+ manager, and it allows to install reactive callbacks and monitor memory via traps that can be accessed also by C programs rather than from Alpha+ programs only. The Alpha+ manager on the contrary depends upon the RC manager functionalities to work, so it can be considered a secondary subsystem.

We will now discuss these two managers in detail, to end up examining how an executable unit containing Alpha+ code is executed in the modified VM, thanks to the work performed by them.
5.3 The Alpha+ manager

We will begin by discussing the Alpha+ manager, assuming the RC manager already exists and provides functionalities to install and remove reactive callbacks. How this actually works will be discussed in section 5.4, for now suffice to say that a new RC is installed by using the following function from the RC manager:

\begin{verbatim}
ui4 CRCMgr_NewRC(CCPU_TEnv** thruEnv,
ui4 inIdx, ui4 inFuncPtr,
ui4 inParm1, ui4 inParm2, ui4 inParm3, ui4 inParm4);
\end{verbatim}

The function receives the current process VM state via the `thruEnv` parameter, the process ID onto which the RC is going to be installed in `inIdx`, and will install a reactive callback monitoring the function pointed to by the `inFuncPtr` address, receiving up to four parameters in `inParm1-inParm4`. Note the function pointed to by `inFuncPtr` is a function located in the specified process memory block. `CRCMgr_NewRC` returns an ID for the new RC record created, and subsequent operations on this RC will use this to identify it. `CRCMgr_NewRC` is also available as a kernel trap, so C programs with no Alpha+ code in them running inside the VM can still install reactive callbacks. A reactive callback is removed by calling the RC manager function:

\begin{verbatim}
Bool CRCMgr_DeleteRC(ui4 inIdx, ui4 inRCRec);
\end{verbatim}

As expected, this function is also available as a kernel trap for the obvious reason of counteracting the `CRCMgr_NewRC` function job.

The RC manager interface we have briefly introduced is all that is needed by the Alpha+ manager to manage reactive callbacks; to the Alpha+ manager, RCs are like black boxes, and it does not need to know their implementation details.

5.3.1 Overview

The Alpha+ manager contains the implementation of all the trap handlers dedicated to the Alpha+ internal language support. These traps are not meant to be accessible directly from the user programs, rather it is the compiler that inserts calls to them at both compile and link time, where required to support certain language features.

During the discussion of the compiler, in section 4.4.5, we introduced the `succeed`, `fail` and `foreach` statements, and we found their implementation requires the use of three new instructions, `bjsr`, `ssp` and `rsp`, all of
5. The virtual machine

which we decided to implement as traps rather than as new dedicated virtu-
al CPU instructions; later in fact we have found trap handlers receive as pa-
ter a pointer to the VM environment, which describes the complete VM state of a process, and can freely modify it. This makes traps the perfect choice for the task, and avoid us changing the virtual CPU component, which can remain slim and unchanged. The trap handlers for these instructions are actually implemented in the Alpha+ manager in the _KAlpha_BackJSR, _KAlpha_SaveSP and _KAlpha_RestoreSP functions. The VM environment provides a snapshot of the virtual CPU registers at the moment the trap is called, so for example the trap can easily access the stack thru the SP re-
gister, as well as the parameters and the local variables of a function thru R1 and R2; the trap can modify these registers and all the memory of the current process. This is why these three traps easily implement the corresponding instructions, and their implementations closely resemble their actual definition as seen at pages 111 and 114; listing 5.2 for example reports the full implementation of the _KAlpha_BackJSR function, implementing the bjsr trap whose call replaces the succeed statement at compile time.

```c
static void _KAlpha_BackJSR ( CCPU_TEnv ** thruEnv ) {

    ENTER_TRAP ("_KAlpha_BackJSR");

    /* save current context on stack */
    SP_ (0) = (*thruEnv)->mPC ;
    SP_ (1) = (*thruEnv)->mR1 ;
    SP_ (2) = (*thruEnv)->mR2 ;

    /* put success (1) as top element */
    SP_ (3) = 1;

    /* restore old context */
    (*thruEnv)->mPC = R2_(-3);   (*thruEnv)->mR1 = R2_(-2);
    (*thruEnv)->mR2 = R2_(-1);

    /* increase stack pointer */
    (*thruEnv)->mSP += 16;

    CallOK_
}
```

Listing 5.2: The bjsr trap handler function
Note how close the implementation is to the actual bjsr instruction definition at page 111.

The Alpha+ manager comprises several other traps. During the discussion of the Alpha+ linker in section 4.5.4, we introduced the ObjectBuilder function, which calls a constructor and builds a list of live object identifiers for which the constructor returned succeed, and which hence have to be created later by the CreateAllObjects trap. The VM keeps track of the live identifiers into an internal hash table attached to each VM process and managed by the Alpha+ manager. New identifiers are added into this hash table by using a dedicated kernel trap, whose handler is the function _KAlpha_EnqueueObjectID. As we will find out also in the next section, hash tables are used in many places in the Alpha+ manager. To avoid code duplication, the manager provides an unified interface for them, using the same approach as described in section 4.5.2 and algorithm 9 for virtual predicates hash tables, but with a difference: while virtual predicates hash tables are static structures embedded into class records, the hash tables handled by the Alpha+ manager are dynamic structures, and automatically grow in size if a certain collisions threshold is passed. As for the former though, these still use open addressing with a form of linear probing to resolve collisions, and the technique used guarantees fast lookups while trying to reduce data clustering to a minimum.

5.3.2 Object instance context and live contexts tree

Before introducing the other traps supported by the Alpha+ manager, it is necessary that we discuss the memory layout of the context associated to an Alpha+ object instance. As we recall from section 3.3.3, every object instance declared in Alpha+ has an associated memory context which must hold the state of instance variables and the addresses of the contexts associated to any sub-object declared within the class the object is instance of. We also need other information to be stored in a context though: the ID associated with the instance, the address of the parent context – if the object is a sub-object of another one – and the address of the class record describing the class of the object instance; these additional data constitutes the header of the context and will be used by the other traps of the Alpha+ manager in various ways.

The memory layout of the generic object instance context is shown in figure 5.3; this requires some further explanations though, as aside from the header followed by the instance variables data section, a context holds yet other information. In section 4.5.4 we discussed on how our goal is to create reaction chains between objects, so that the live contexts tree is kept up to date automatically and predicates are re-executed in reaction to monitored
events. Since all predicate calls stem from the execution of the `setup` function of the class an object is instance of, we found how this can be achieved by installing a reactive callback on it upon object creation; this explains the need to save the ID of the RC installed on `setup` in each object instance context.

When creating an object instance though, the VM must also take care about creating the contexts of any sub-object, and establish reaction chains on them as well. Sub-objects of an object are discovered by the VM thanks to the object class record: the structure of a class record (as we recall from figure 4.12) in fact stores the addresses of the constructor functions for any sub-class whose objects are declared inside the class. A class record for class `A` stores up to `m` sub-records describing the `i`-th sub-class; the same number of records are stored in the context for an object instance of class `A`, where each record has two entries. The first entry stores a pointer to an hash table holding the addresses of the contexts of any sub-object the `i`-th sub-class constructor has instantiated; the second entry stores the ID of an RC the VM installs on the `ObjectBuilder` function as described in section 4.5.4. This is used to both cause the actual creation of the sub-objects and to establish reaction chains as we will discuss in the next section.

The use of hash tables to store the addresses of the contexts of any sub-object instance of a given class, effectively creates the live contexts tree structure we encountered many times in this paper. To better illustrate how this works, in figure 5.4 we show the live contexts tree as stored in memory, relative to the small Alpha+ program we report in listing 5.3; classes in the program omit all details except sub-objects declarations, as we are lonely interested in these to build the live contexts tree in our example.

---

**Figure 5.3:** Object instance context layout
class C {
    ...
    int x;
}

class B {
    ...
    body:
        int y1, y2;
        C(out 1);
}

class A {
    ...
    body:
        int z;
        B(out 1);
        B(out 2);
        C(out 1);
}

A(out 1);
C(out 1);

Listing 5.3: A simple Alpha+ class hierarchy

The resulting complex figure also shows the connections between the contexts and their parents, and between the contexts and their associated class record. The intricate connections allow to highlight several important aspects inherent to the memory structure of the live contexts tree. We can make these useful considerations:

- Each object instance of class \textit{A} has as many slots for hash tables as the class record for class \textit{A} has sub-classes. This is because each hash table stores the object instance addresses for all the objects of a given class being declared inside the class of the parent object.

- Each object instance has references to the parent object and the class record the object is instance of.

- Object instances declared in the global Alpha+ scope do not have a parent object, and thus their addresses cannot be stored in the corresponding hash tables of a parent object. To keep track of these objects,
the corresponding hash tables are stored in the relative class record for convenience. So all the addresses of global object instances of class A are stored in an hash table attached to the class record for class A.

- Since object instances of the same class are referenced by an independent hash table attached to their parent (or in the case of global objects, to the corresponding class record), and are mapped on it via their ID number, the result is that each class of objects has a dedicated numerical ID space, and that is also valid for objects declared within other objects. This allows as in our example to have several object instances with the same ID number 1 and still to be able to distinguish between
them; they are in fact attached to independent hash tables as they are related to objects of different classes, possibly declared inside other objects.

The above discussion should have shed further light on the need for class records; other than for the uses examined thus far in fact, these are extremely important also to determine the sub-objects of an object.

Now that we have a better understanding of the inner memory layout of the live contexts tree and its connections between contexts and sub-contexts, contexts and class records, we can discuss how other important kernel traps available in the Alpha+ manager work. The _KAlpha_InitGlobalObjects trap handler for example, is called by the AlphaInit function of the Alpha+ glue code for each class whose objects are declared in the global Alpha+ scope, and executes most of algorithm 12. In fact it creates the hash table for the global objects of the given class, and attaches it to the corresponding class record; it then schedules the execution of the ObjectBuilder glue code function as a reactive callback, passing to it the address of the corresponding constructor predicate function as found in the global class record. This will trigger the creation of all the global objects for the given class, which will then automatically trigger the creation of any sub-objects and the creation of the reaction chains.

While discussing the types of predicate calls in section 4.4.6, we found that certain types of calls require a lookup; with reference to that section, we have seen case 3 requires to lookup the virtual predicate to be called in the class record of the parent object context. With the structures introduced thus far this is an easy task for the dedicated _KAlpha_LookupVirtualMethod trap handler function: if an object instance context has a parent context, it gets the address of its attached class record, otherwise it gets the address of the global class record. It then performs an hash lookup in the static virtual predicates hash table attached to the class record, and it returns the address of the correct virtual predicate to be executed.

A different kind of lookup is required in case 2 as still highlighted in section 4.4.6: this case in fact requires to lookup a context rather than a virtual predicate. The live contexts tree memory and data structure again proves to perfectly suit the task, thanks to the hash tables attached to each context. The address of the context of a sub-object of class A with a given ID number is easily found by the _KAlpha_LookupContext trap handler, that simply performs another hash lookup in the hash table relative to sub-objects of class A attached to the current context.
5. The virtual machine

5.3.3 The CreateAllObjects trap

The last trap we will discuss is the CreateAllObjects trap, whose handler is the _KAlpha_CreateAllObjects function. This is the most important Alpha+ manager kernel trap because, as we have seen in the Alpha+ glue code discussion in section 4.5.4, it is this trap that actually creates the new object instance contexts and destroys obsolete ones, while maintaining the reaction chains. Basically it is this trap that keeps the live contexts tree updated in response to reactions; algorithm 13 explains how it works.

Algorithm 13 Trap handler _KAlpha_CreateAllObjects(P, H_p, R, A)

1) Let H_i be the hash table already filled with the IDs for objects of class C to be created
2) Clear an internal live contexts hash table H_l
3) For each ID held in H_i check if the the ID is already in H_p. If it is, the object is already alive, otherwise do:
   3a) Allocate new object instance context O with given ID
   3b) Set P as parent context of O and R as its class record
   3c) Create and attach to O a new RC with class C setup(O)
   3d) For every sub-class S_i declared in R do:
      3d1) Let R_{S_i} be the class record for S_i
      3d2) Let C_{S_i} be the constructor of S_i as found in R
      3d3) Allocate new hash table H_{S_i} for objects of class S_i and attach it to O
      3d4) Create and attach to O a new RC with function A(O, H_{S_i}, C_{S_i}, R_{S_i})
   3e) Add O to H_l
4) For each context D held in H_p but not in H_i do:
   4a) Schedule execution of trap DestroyObject(D)
5) Clear H_i

Before discussing the algorithm, it is important to examine when and how the CreateAllObjects trap gets called. In section 4.5.4 we found how the ObjectBuilder function, built in the glue code, is dedicated to call in a foreach context a constructor predicate function whose address it receives via a passed parameter. Every time the foreach iterates it means the constructor issued a succeed and placed a valid ID number into an out parameter passed by reference. ObjectBuilder then calls a trap whose handler in the Alpha+ manager is the _KAlpha_EnqueueObjectID function, whose only purpose is to store the ID of the new object to be created into an hash table. When the ObjectBuilder function terminates the foreach cycle, the
The virtual machine hash table will contain all the IDs of the objects to be built by the VM for the class whose constructor address was initially passed to the function; at this point the CreateAllObjects trap is called, and its task is to actually create these objects. CreateAllObjects receives four parameters: \( P \) is the address of the parent context, which may be NULL in the case of global objects; \( H_p \) is the address of the contexts hash table of the parent object (which in case of global objects having no parent, is the address of the global objects hash table stored in the relative class record as previously discussed); \( R \) is the address of the class record for the class we are creating instance objects of, and \( A \) is the address of the ObjectBuilder glue code function which issued the trap.

With these premises, we can now start analyzing how the CreateAllObjects trap works as highlighted in algorithm 13.

The first two steps are trivial: the Alpha+ manager keeps track of the object IDs reported to the \_KAlpha\_EnqueueObjectID trap handler, by storing them in an hash table \( H_i \) available globally to the current VM process. CreateAllObjects just assumes this to be filled with proper live object IDs when it enters execution. \( H_i \) is a secondary hash table local to the trap that is used to store the contexts created by the trap, to later compare them with the \( H_p \) table. The core of the trap work is done in the cycle initiated by step 3, executed once for every new object ID not already found in \( H_p \), and thus not already created.

Steps 3a and 3b allocate a new object instance context and assign the new object ID to it; the context also keeps track of its parent.

Step 3c is very important: here we install a new reactive callback calling the setup function for the class we are creating an instance object of. The setup function is scheduled for execution and it will receive the new context address as parameter; it will also be called in the context of a reactive callback, meaning it will monitor all the memory referenced by setup itself and any predicate called from it, effectively establishing new reaction chains.

Step 3d starts a cycle to create any sub-object of the new object; the cycle iterates for each sub-class whose objects are declared inside the class the new object is instance of. It should be no surprise that an important role in this cycle is taken by the class record associated to the class of the new object; as we discussed in section 4.5.2, class records store the addresses of other class records for the classes of any sub-object declared within the class they describe. They also store the address of the constructor predicate function dedicated to build the sub-objects of each particular sub-class.

Step 3d3 contributes creating the contexts connections thru hash tables as examined in section 5.3.2 and explicated in figure 5.4.
Step 3d4 is also of great importance: it installs a new reactive callback on the ObjectBuilder glue code function, passing to it all the proper parameters to let it restart the foreach process that will culminate with the re-issuing of the same CreateAllObjects trap with all the consequences of the case – most importantly the creation of new reaction chains – but for the creation of sub-objects instances of the current sub-class being examined in the cycle started by step 3d.

At the end of step 3, all new objects have been created, and their setup function scheduled for execution via reactive callbacks; sub-objects are also scheduled for creation, again thru the use of reactive callbacks. What the CreateAllObjects trap still has to perform is to destroy objects that were alive before its execution and whose IDs do not result among those that have been created, and that are thus to be considered the new live objects. This operation is performed in step 4, that for each of the object instance contexts to be destroyed, schedules the execution of the DestroyObject trap; its handler, implemented in the Alpha+ manager, manages to schedule the execution of the proper cleanup function for the object and all its sub-objects, before freeing the memory associated with them and removing all the relative reactive callbacks that were previously installed, effectively destroying all the reaction chains.

The CreateAllObjects trap just described constitutes the core of the Alpha+ manager work; as it is clear, it heavily depends on the reactive callbacks system to perform its task. This ends our discussion of the Alpha+ manager; we will now focus on the RC manager to find out how it provides its functionalities to let the Alpha+ manager work as intended in a fully reactive environment.

5.4 The RC manager

The Leonardo VM did not provide a reactive environment when we started the project, and making it support such an architecture has been one of the most challenging parts of our work. Our design assumes the possibility to monitor individual memory addresses, and to install one or more watches on it; each watch has an attached function we have named reactive callback. We have extended the mean of the term to indicate also the associated watches, so we often refer to reactive callbacks monitoring certain address ranges. Whenever a monitored memory cell is modified, the corresponding reactive callback is executed again, and its execution causes the re-monitoring of possibly new memory cells, different from the last time the function was called; this change can be due to function state changes that triggered the
5. The virtual machine

execution of certain conditional blocks of the function rather than others. We will examine such situations in the next section.

### 5.4.1 Monitoring addresses

A reactive callback is installed via the CRCMgr_NewRC function provided by the RC manager. This accepts a pointer to the function to be installed as callback, and up to four parameters to pass to it upon triggering; this maximum number of parameters has been chosen to satisfy the needs of the Alpha+ manager, specifically of the CreateAllObjects trap, but it can be extended if needed in the future. CRCMgr_NewRC creates a new RC record, holding informations on the callback and its state; RC records are stored in structures of type CRCMgr_TRCRec, as reported in listing 5.4.

```c
typedef struct {
    ui4 mFunPtr;
    ui4 mParam[4];
    ui4 mRCList;
    ui4 mSession;
    Bool mActive;
} CRCMgr_TRCRec;
```

Listing 5.4: The RC record structure

The mFunPtr and mParam fields correspond to the callback function to be triggered and the parameters to be passed to it; mActive is a boolean field telling whether the RC is active or not – inactive RCs stop triggering the callback function even if the monitored memory is modified. The meaning of the other fields will become clear later.

When a new RC is installed, the first task to be executed is the scheduling of the callback function for execution; scheduling is accomplished in the function CRCMgr_EnqueueRC with the help of interrupts the VM already supports via the event manager. The scheduled interrupt is particular however as it is marked as an RC interrupt; this informs the VM, and in particular the memory manager, that when the interrupt is dispatched, the function is to be executed under the context of a reactive callback, and thus it will have to start monitoring accessed memory.

The first execution of the callback function upon RC installation is of great importance as it installs the watches on the accessed memory, and these watches will later cause the automatic re-execution of the callback upon memory modifications.
Monitoring is performed by hooking the memory manager `CCPU_MEM_READ` trap. As we recall from section 5.2.2, this trap is automatically called by the virtual CPU whenever a memory cell is accessed for reading; to fulfill our design goals, every variable that is accessed for reading in the callback function must be monitored, so this trap is the perfect place where to call the RC manager to mark the address of such variables as monitored. Before discussing what actions are actually performed to monitor an address, it is recommended to recall that the memory manager, for each memory word of the block allocated for a process, also stores a dedicated bitmask; before the introduction of the RC manager, the only role of this bitmask was to store access rights for the associated memory word in order to mark it as read-only, write-only or read/write enabled. The availability of this mask has been reused by the RC manager to store additional informations on a per-word basis; specifically, two new bits have been used. The first is used to mark a word as monitored by at least one RC; this helped making the triggering process faster as we will see. The second bit is used to mark the corresponding word as already monitored by the current RC being executed during the monitoring process, as we will now examine.

The `CRCMgr_WatchAddress` function of the RC manager performs the monitoring of a given address. This function is called from the `CCPU_MEM_READ` trap of the memory manager, and it is extremely important that it performs its task as quickly as possible, as the trap is called for each memory read access the virtual CPU operates. It is also important to note though that the memory manager calls this function only if running under the context of an interrupt that is marked as RC by the event manager, that is, only during the execution of a reactive callback function. `CRCMgr_WatchAddress` basically just ensures that the address is not already monitored by the current RC being executed – simply by checking a bit thanks to the bitmask associated to the process memory – and if not, it allocates a new watch node into which it stores the monitored address, puts a link to the RC record of the reactive callback being executed, and places this node in a suitable data structure, so it can be later used by the triggering process. We will examine this data structure with figure 5.5 as a reference.

This assumes we have three reactive callbacks being installed, which are supposed to monitor 4, 3 and 3 variables respectively. With the help of the figure, we can then introduce some concepts:

- To each VM process is assigned an addresses hash table, initially capable of holding the addresses of 256 words without collisions. This hash table is used to lookup watch nodes, each of which is assigned to an address. The hash table uses chaining to resolve collisions.
Watch nodes have an associated address, and form a doubly-linked list with other nodes occupying the same hash slot. Each node is also singly-linked to other nodes, with the head of the list being located in an RC record. This connection forms the list of addresses watched by each RC installed.

Multiple nodes watching the same address can appear in the same doubly-linked list attached to the same address hash table slot; this means there is more than one RC monitoring that particular address, corresponding to the same variable.

When a new address needs to be monitored, the `CRCMgr_WatchAddress` function creates a new watch node for it, whose structure is defined by the `CRCMgr_TRCList` type as in listing 5.5; in this structure, which effectively establishes a reaction chain, `mAddr` is the address associated with the node, and `mRCRec` is a pointer to the RC record structure relative to the RC which is monitoring this address. The other fields are used to link the nodes as
5. The virtual machine

we discussed above; once created in fact, the node is attached to the tail of
the doubly-linked list for the addresses hash table slot associated with that
particular address, and it is also appended to the singly-linked list attached
to the RC record referenced by the current RC interrupt being executed, via
the mRCList field of the CRCMgr_TRCRec structure. The mRCRec field may
seem redundant as nodes are already linked to the corresponding RC record;
it is however needed to perform an \( O(1) \) access to the RC record from the
node during the RC triggering process, as we will examine in the next section.

typedef struct {
    ui4 mAddr;
    ui4 mRCRec;
    ui4 mNextRC;
    ui4 mNextHash;
    ui4 mPrevHash;
} CRCMgr_TRCList;

Listing 5.5: The RC watch node structure

There is a reason why chaining via a doubly-linked list is used to resolve
collisions instead of a more performing open addressing method. To explain
it, we must more closely examine the CRCMgr_EnqueueRC function; we have
previously seen this is used to schedule an RC interrupt to execute an RC
function and start monitoring the addresses it references. This is true, but
we omitted a detail: the mechanism to work requires that before a callback
starts monitoring, all the addresses previously monitored by this callback
have to be removed. That is, all the watch nodes associated to its RC record
must be freed. The reason for this requirement is simple, and we will explain
it with the example in listing 5.6.

Suppose the function rc_function has been installed as a reactive call-
back. Upon installation, the RC is scheduled for execution so it can start
monitoring addresses. rc_function executes, and the content of the \( k \)
variable is read. This makes the CRCMgr_WatchAddress function add a new
watch node for the address of \( k \). The if succeeds as \( k \) is 0 at program star-
tup; \( x \) is then accessed for reading as well, a corresponding watch node is
added, and the function terminates. When the RC interrupt ends, we have
two watch nodes attached to the RC record. Now when the \( k \) variable is
set to 1 later in the program execution, the VM recognizes the address is
monitored by an RC, and triggers its execution again via an RC interrupt.
The process restarts, but this time the if fails, and \( x \) is never accessed; the
result is that at the end of this second RC interrupt we have monitored one
int x = 0, k = 0;

void rc_function()
{
    if (k == 0) printf("x is %d\n", x);
}
...
k = 1;

Listing 5.6: Conditional monitoring in an RC

address only, with the consequence that the address of x should not be monitored anymore. This means that the reaction chain associated to it must be removed.

This explains why the CRCMgr.EnqueueRC function, before scheduling an RC interrupt, calls the CRCMgr.RemoveAllWatches function in the RC manager, that uses the singly-linked list attached to the RC record to find all the previously attached watch nodes and remove them. During the removal process of a node, the chain linking together nodes whose addresses lie in the same hash table slot must also be updated; to ease this process, the doubly-linked list approach has been used. As an example, figure 5.6 shows the same situation of figure 5.5, but with all the watches of a reactive callback removed; this is how the memory layout would look after the execution of CRCMgr.RemoveAllWatches on that RC, before its callback is re-executed and restarts monitoring.

An open addressing technique for the implementation of the addresses hash table would have made nodes lookup faster, but would have also caused serious slowdowns during nodes removal, so the idea has been discarded. In addition, rather than nodes lookup in the RC triggering code we simply need to traverse a list, as we will find in section 5.4.2. Nonetheless, to avoid problems inherent to excessive collisions, the hash table is automatically resized when a certain threshold ratio between nodes and table size is passed.

5.4.2 Triggering callbacks execution

Automatic triggering of reactive callback functions is performed by hooking the CPU_MEM_WRITE trap of the memory manager. By design in fact a reactive callback keeps track of memory changes, and should only be
triggered when monitored memory is modified; the aforementioned trap is automatically called by the virtual CPU whenever a memory cell is written to, so it perfectly suits our needs. We modified the trap to make it call the CRCMgr_CallRCsForAddressRange function of the RC manager; as with the case of CRCMgr_WatchAddress, this one also needs to be as fast as possible, being called every time a memory write operation occurs.

CRCMgr_CallRCsForAddressRange scans a given range of addresses, and determines whether an address is monitored by at least one RC by checking the corresponding bit in the process memory mask. If an address is marked, the function knows there is an RC that is currently monitoring it and the execution of the corresponding callback function needs to be triggered. CRCMgr_CallRCsForAddressRange just traverses the doubly-linked list associated with the hash table slot of the given address; as it is built, this list can hold many different addresses – all corresponding to the same hash table slot – as well as the same address multiple times, but associated to different RC records; figure 5.5 shows all these situations for example. The traversal in such case, assuming we want to trigger any RC callback function monitoring address 7, would be as in figure 5.7; the list scan starts from the head located in the hash table, and goes thru all the watch nodes attached to it.
Whenever a node has the address 7 associated with it, the corresponding RC callback function is triggered.

Note that during the traversal, when a node with the correct address is found, the corresponding RC record is immediately retrieved via the mRCRec field of the CRCMgr_TRCList node structure. The RC record holds the address of the callback function to be scheduled for execution, and the parameters to be passed to it; the CRCMgr_EnqueueRC function is used to schedule the callback execution. As we recall from section 5.4.1, this function removes the addresses monitored by the given RC, and then enqueues the RC function for execution as an RC interrupt. This will cause the re-monitoring of the function, rebuilding new reaction chains. Care is taken to ensure an RC is not enqueued more than once during the scan of an address range, thanks to the use of the mSession field of the CRCMgr_TRCRec structure; also, due to the considerations made in chapter 2, a callback function is not scheduled for execution if the memory write operation does not modify the content of the affected memory cell.

It is also to be noted that both monitoring and callbacks triggering only work for addresses ranging in the process heap memory that do not overlap with the process stack region; this is necessary as the stack is volatile. Monitoring a local variable of a function for example, which resides on a stack
5. The virtual machine

frame, would cause troubles when the function ends; the watch nodes would keep monitoring the address of that local variable, but that address is no longer valid as the stack frame has been destroyed. The worst-case scenario, which unfortunately can happen quite frequently, is when a new function call causes the creation of a new stack frame which overlaps with the same heap region previously occupied by the stack frame of the old function; since the watch node is still monitoring the same address now used by the new stack frame, some write operation of the new function could cause the undesired effect of triggering the reactive callback associated to the old variable.

5.5 Conclusions

In this chapter we have examined how the Leonardo virtual machine is structured and what changes we applied to make it manage the executable units containing Alpha+ programs previously compiled. We introduced new functionalities to support special Alpha+ requirements in the Alpha+ manager, then we discussed the RC manager to provide a reactive environment as described in chapter 2; our solutions for this last challenge are oriented towards adding a reactive environment for Alpha+ but are not limited to it, rather the whole VM has been made a reactive system. Efficiency has been a key goal thru the project and we believe we have achieved good results, also thanks to the extensive use of hash tables.
Appendix A

Alpha+ language productions

This appendix contains the Alpha+ language productions; many of these have been incorporated into each other during the implementation to reduce code duplication. We present them separated here for clarity reasons.

alpha_unit:
    alpha_declaration
    alpha_unit alpha_declaration

alpha_declaration:
    alpha_class_declaration
    alpha_predicate_declaration

alpha_class_declaration:
    'class' identifier '{' [alpha_class_interface]
    ['body:' alpha_class_body] '}

alpha_class_interface:
    alpha_method_declaration
    alpha_class_interface alpha_method_declaration

alpha_method_declaration:
    alpha_virtual_method
    alpha_method

alpha_virtual_method:
    'virtual' identifier '(' parameter_list\[\]
    ');'

\[\text{1}\text{Modified to support out parameters and shortcuts as described in section 3.4.2}\]

152
Appendix. A. Alpha+ language productions

\[
\begin{align*}
\text{alpha}_{-}\text{parameter:} & \quad \text{[‘out’]} \ [\text{declaration}\_\text{specifier}] \ \text{declarator [‘=’ expression]} \\
\text{alpha}_{-}\text{method:} & \quad \text{identifier } \text{‘(’ parameter\_list\textsuperscript{1} \text{’)}’} \ \text{alpha}_{-}\text{predicate}\_\text{body} \\
\text{alpha}_{-}\text{predicate}\_\text{body:} & \quad ;
\quad \text{compound\_statement}\textsuperscript{2} \\
\text{alpha}_{-}\text{class}\_\text{body:} & \quad \text{alpha}_{-}\text{class\_body}\_\text{declaration} \\
& \quad \text{alpha}_{-}\text{class}\_\text{body} \ \text{alpha}_{-}\text{class}\_\text{body}\_\text{declaration} \\
\text{alpha}_{-}\text{class}\_\text{body}\_\text{declaration:} & \quad \text{alpha}_{-}\text{predicate}\_\text{declaration} \\
& \quad \text{’setup’ compound\_statement}\textsuperscript{2} \\
& \quad \text{’cleanup’ compound\_statement}\textsuperscript{2} \\
& \quad \text{declaration} \\
\text{alpha}_{-}\text{predicate}\_\text{declaration:} & \quad \text{identifier } \text{‘(’ out\_identifier [‘=’ expression] \text{’)}’} \\
& \quad \text{alpha}_{-}\text{predicate}\_\text{body} \\
& \quad \text{identifier } \text{‘(’ identifier \text{’)}’ \text{.} \text{alpha}_{-}\text{method} \\
& \quad \text{identifier } \text{‘(’ expression \text{’)}’ \text{.} \text{alpha}_{-}\text{method} \\
\text{alpha}_{-}\text{method\_call:} & \quad \text{identifier } \text{‘(’ parameter\_list\textsuperscript{*} \text{’)}’} \\
& \quad \text{identifier } \text{‘(’ expression \text{’)}’ \text{.} \text{identifier } \text{‘(’} \\
& \quad \text{parameter\_list\textsuperscript{1} \text{’)}’} \\
\end{align*}
\]

\textsuperscript{2}Extended to include Alpha+ specific constructs as succeed, fail, foreach, predicate calls and instance variables references
Acknowledgements

Alcuni sentiti ringraziamenti, almeno questi squisitamente in italiano:

- Ai miei genitori Antonio e Maria, che con il loro costante affetto mi hanno sempre sostenuto, e grazie al cui supporto ho potuto intraprendere gli studi universitari. E' grazie a voi se oggi corono il mio sogno di laurearmi in Ingegneria.

- Al mio relatore professor Camil Demetrescu e alla moglie Irene, due persone splendide e sempre disponibili in questo lungo anno di preparazione della tesi. Mi ritengo estremamente fortunato ad avervi conosciuto.

- Al professor Daniele Nardi, i cui insegnamenti del corso di Linguaggi e Traduttori mi hanno fatto particolarmente appassionare ad una materia tanto difficile quanto bella. E' un onore averla come contro-relatore.

- A mio fratello Ernesto, che ha continuato a dispensarmi utili consigli per tutto questo tempo. Non sempre li ho seguiti, ma come ben sai la testardaggine è una caratteristica di famiglia...

- Ai miei amici ingegneri in particolare, senza il cui supporto nei momenti di difficoltà e senza le numerose giornate passate a preparare gli esami insieme, sarebbe stata molto più dura terminare gli studi.

- A tutti i miei amici, fortunatamente troppo numerosi per essere elencati di seguito. La vostra presenza è stato un supporto fondamentale in tutti questi anni di ingegneria, e mi ha regalato momenti di grande felicità senza cui andare avanti sarebbe stato molto più difficile.

Angelo Mottola

Roma, il 19 dicembre 2005
Bibliography


